Swiftwayz Holdings (Pty) Ltd

HIGH LEVEL DESIGN DOCUMENT

goSwift Transportation Platform

Document Number: SWIFT/BRS/2802/2017

1. goSwift introduction 3

2. Developer tools 3

3. Design mantra 3

4. Modules/Sub-systems 4

5. Frameworks to be used 6

6. Datasets 6

7. Business processes 6

8. Functionality supported by each module 7

9. Testing 11

10. Technical Issues: - 11

10.1. Algorithms 12

11. Billing data model 14

# goSwift introduction

Develop a pre-paid taxi hailing platform that allows users to purchase pre-paid

Miles that can be translated into actual kilometers on the network. The miles can be sent to other users on the platform. Users who travel a certain number of miles will earn points and qualify for free miles.

* Prepaid service
* Pay-as-you-go billing
* Top-up your miles
* Send miles to friends on goSwift
* Split fare in group
* Commuter subscription (to work)

# Developer tools

* Android studio
* MongoDB
* Node.js
* Redis
* Apache thrift – services
* RabbitMQ
* Java SE Development Kit 8
* Setup CVS to GitHub

# Design mantra

* Package similar functionality into services
* Expose the service through a clean and well defined interface
* All services can be called through REST API or a more efficient RPC protocol like Thrift
* A service can be stateless or stateful
* All persistent data should be formatted in JSON as well as another binary format (messagepack)
* Design for high scalability (support for 1 million reads and 250,000 writes)
* Ensure that there is no single point of failure
* Ensure that most critical components can be replicated across machines
* Viral loop effect that enables service to spread from one user to a new user

# Modules/Sub-systems

|  |  |  |
| --- | --- | --- |
| Module | Platform | Description |
| * Rider App | Native Android | * Request taxi * Search closest and best positioned drivers * Calculate fares and trip kilometers * Make payment and load credit (miles) |
| * Driver App | Native Android | * Constantly polling/pinging dispatch service * Search for rider location * Logging GPS data every 4 seconds about driver location * Trigger a trip complete * This will record the final destination GPS and flag the trip is completed * The Dispatch service must send the trip data collected to Billing Engine using the RabbitMQ message service for processing |
| * Connection Request Processor (CRP) | Node.js/Redis | * Stores GPS logs in Redis (key-value datastore) * Great for fast searches * Use Google S2 to do geo-spatial queries using quad-cells |
| * Dispatch system | Node.js/Redis  /MongoDB | * Dispatch logic in node.js * Search closest drivers using a binary tree, redis * Stateless machine * Do all distance calculations * Push distance logs to billing engine using RabbitMQ * Log all gps data * Push to RabbitMQ * Kalmer filter (algorithm to smooth gps points along a route) * Rider/driver matching algorithms when doing car pooling * Optimal route estimates * Display driver location to riders all the time in real-time * Keep trip logs in redis hash-map * Keep driver logs in redis hash-map * Keep rider logs in redis hash-map * Use kalmer filter to “fix” incorrect gps logs |
| * Pricing engine and process queues | Java/RabbitMQ  /MongoDB | * Calculate fares for car pools and time-windows * Bill customer bank card * Asynchronous trip event queue * Send e-mail with invoice * Campaign and promotions * Store rider locations (home, work, shop, play) * Stores data in MongoDB/MySQL |
| * Driver and Partner portal | Java/MongoDB | * Create and manage driver profiles * Create and manage partner profiles * Reports of trips completed * Financial reports of revenue generated * Reports of kilometers driven by driver while logged in * Reports generated by vehicle/driver |

# Frameworks to be used

* Google’s S2 geometrical library
* Geo-hashing
* Web sockets
* JSON - message format (GeoJSON) – very slow and large footprint
* MessagePack (Thrift, Protocol Buffer)
* Google mapping
* Angular – Web application framework
* Mongoose
* RPC protocol to use? Persistent TCP
* SMPP – SMS Gateway
* OkHttp
* Rest API/HTTP for communications (Android - > Node.js)
* Google distance matrix

# Business processes

* Trip process (state change transition)
  + **request a taxi**
  + ETA state,
  + Taxi arrive
  + **begin trip**
  + on-trip,
  + **finish trip**
* bill client (invoice)
  + calculate kilometers
  + calculate fare
  + deduct from account balance
  + e-mail invoice
* Payment process
  + Confirm payment method
  + Enter amount in cash
  + Convert amount to actual kilometers (give examples JHB – Sandton)
  + Buy miles
  + Send receipt
* Request/Send miles
  + You request someone to send miles
  + Sender authorize miles
  + Sender Account is debited
  + Requester account is credit
* Create campaign
  + Enter campaign period (start/end date)
  + Campaign name

# Functionality supported by each module

|  |  |  |
| --- | --- | --- |
| Module | Platform | Description |
| Rider App | Android mobile application | Customer mobile app   * Used for 2 minutes each time * 3 clicks max and must be very simple * Create user profile * Geo-location and mapping * Discover other users around you * Flash when friends are within a certain radius from places that you normally travel to, or a place that you are currently at. * Invite friends to use goSwift (think WhatsApp invite) * Add a “travelling-to” feature that a user can broadcast * Create group-trip feature (add co-riders and split fare) * Capture departure location (current location or other location) * Capture destination * Request a taxi now (Go Now) * Estimate fare (display kilometers, price per km and total fare) – display option with pooling and waiting * Request a taxi later (once-off or recurring) * Display nearby taxis with their time to pick-up * Request a friend to send you a driver or goSwift miles? * Display estimated prices of trip based on trip, number of riders in pool, departure time-window (e.g. 15, 30, 60, 120, 180 min) * Display driver ratings and enable rider to choose between drivers? * Display notifications panel showing booking status, driver details like number plate, vehicle make and color, driver name, contact number, expected pickup time and distance, promotions and deals, busiest routes and locations, * Driver ratings * Most frequented places (e.g. work, home, shopping center, gym etc.) |
| Driver App | Android mobile application | Driver mobile app   * This app is used for 12 hours a day * Must be reliable and stable with few bugs * Driver login/authentication * Driver goes online-offline * Accept incoming trip requests * Send GPS location of driver (once every 5 seconds) * Send state information (e.g. Fetching client, Available, On-Trip) * Send arriving now notification (5 minutes?) * Navigate to customer * Navigate to destination * Driver notifications panel * Do not display customer destination or price to driver (to avoid drivers second guessing) * Highlight hotspots for drivers on the map * Display previous pick-up locations based on time to assist drivers with positioning |
| Connection Manager | Node server | * Dispatching requests to appropriate service (e.g. Dispatch, Pricing, BizLogic etc.) * Keeps GPS logs * Manage the flow of request to dispatch, pricing, partner and driver services * When customer request fare estimates, CONMAN will calculate distance estimate of trip and request billing service to estimate the fare and return this to rider * Do Google places and address geo-coding queries when rider request trips and estimate fares |
| Dispatch system | Node server application | * Vehicle GPS logging (once every 6 seconds) * Each driver updates cell ID 12 times per minute based on current location * S2 Region coverer used to search for cells with following parameters (min level = 2, max level = 20, max cells = 95) * Geo index drivers at a radius of 3,3 km * Customer GEO-location (every 1 minute) * Consider using GEO commands in Redis vs Google S2? * We must expire GPS data for drivers to limit the number of logs to traverse when doing geo queries like GEORADIUS * Demand and supply matching * Routing engine * Geo indexing of riders and supply positioning (where demand is likely to come from?) * GEO fencing and geo-spatial indexing * Riders search for drivers in real-time * Expected Time of Arrival (ETA) engine, develop a basic one initially (distance from driver to customer divided by speed to drive to customer) * Input Driver GPS, Customer GPS, speed limit, road distance, straight line distance, time-of-day, current speed, historical data – how long did it take to drive same route around same time-of-day 10,5,3,1 week ago) * Dispatch queuing and customer matching * Real-time notifications working much like WhatsApp messages (web sockets or RPC protocol) * Demand prediction (city, local, time-of-day, traffic, events, campaigns) |
| Pricing and BizLogic engine | Java application | * Create travel time windows (for specific routes, particular areas) * Create trunk routes * Create transfer stations * Create a route profile * Create price profiles * Price per kilometer * Base rate + per kilometer + per minute * Rate per person sharing * Fare split between partner and goSwift * Create payment profile (bank card, credit card, PayPal, cash) * Payments gateway integration * Schedule payment on the queue * Load goSwift miles into account * Convert goSwift miles to actual miles * Fare calculation * Billing to be implemented using RabbitMQ * Post-paid service for corporate or contract customers * Trip data creation and update |
| Partner & driver portal | Web application | * Create partner profile * Create driver profile * Add vehicles * Link vehicles to partners * Link vehicles to drivers * Link driver to partner profile * Upload partner documentation (partner info, ID, contact info etc.) * Upload driver documentation (driver license, crim-checks and assessment) * Upload vehicle documentation (vehicle docs, insurance, assessments) * Driver acceptance workflow process * Partner acceptance workflow process |
| Promotions and loyalty | Java Application | * goSwift miles service * Promotion codes * Collect driver ratings * Collect rider ratings * Add a check-in feature using goSwift into a location |

# Testing

|  |  |
| --- | --- |
| Device testing | * Different OS (Android, iOS) * Different devices (Samsung, Sony, iPhone, Huawei, LG) * Different resolutions * Test on different orientations |
| Alpha testing |  |
| Beta testing |  |
| Integration test | Python  Simulate driver  Simulate rider |
| Load testing | JMeter? |

# Technical Issues: -

* Create 3 redis hash-maps to store GPS logs
  + Customer GPS data
    - Log location of customer when they request trip
  + Driver GPS data
    - Log location of driver 12 times per minute
    - Expire gps logs older than 60 minute and persist
    - Ensure gps logs are stored in a shard storage (monthly shard)
  + Trip GPS data
    - Log gps data for trip
    - Expire and persist when trip is complete
    - Stay small as possible and not grow indefinitely
* GPS jumping too far too quickly, such as when you drive into a tunnel, bridge or parkade. You will need to correct the route (see kalman filter)
* Track user events/interaction to see how the UX is being used
* Consider handling GEO queries using an abstracted layer of GEO Indexing Providers that implement a common interface (Geo-hashing vs Google S2)
* Use a RedisGeoProvider as well as S2CellGeoProvider (implement GeoIndexInterface)
* Both providers must provide the following basic functionality: -
  + Calculate distance between 2 points/latlng (getDistance) in meters
    - Redis implemented using GEODIST command
    - S2 implemented using S2 get distance method
  + Calculate getGeoRadius given a point/latlng (getRadius) in meters
    - Redis implemented using GEORADIUS command
    - S2 implementation using
  + Implement saveLocationIndex() method to save GPS as spatial index
    - S2CellProvider implemented using a 64-bit S2 spatial index
    - RedisGeoProvider implemented using geo-hashing(52-bit) supported by redis’ GEOADD command

# Algorithms

* Kalmin filter to fix GPS errors
* ETA algorithms using time of day, historical data, date of week, current demand and time windows
* Algorithm for trip carpooling using, busiest routes, historical data, current demand, time windows
  + Log GPS of each customer *depart\_location*
  + Log GPS of each customer *arrival\_location*
  + Calculate distance of *cust\_departure* to their *cust\_arrival* (straight line or road distance)
  + Calculate distance between each *cust\_departure* with every other *customer\_departure*
  + Calculate distance between each *cust\_arrival* with every other *cust\_arrival*
  + *Use Linear regression techniques to compute the route with the least number of errors to pool the most customers.*
* Pricing algorithm based on distance, route demand, time-of-day, first-to-book pays the lowest system
* Demand prediction algorithm (answer the question, where should the taxis be holding/waiting)
* Dynamic trunk routes algorithm based on number of people travelling in a particular direction time of day etc. (regression analysis ?)

**Redis key-value store**

1. driver\_location
   1. driver\_id
   2. vehicle\_id
   3. time\_stamp
   4. gps\_latitude
   5. gps\_longitude
   6. velocity
   7. bearing
   8. gps\_accuracy
   9. charging\_bool
   10. battery\_level
2. rider\_location
   1. rider\_id
   2. datetime\_stamp
   3. gps\_latitude
   4. gps\_longitude
3. journey
   1. destination\_lat
   2. destination\_lon
   3. depart\_latitude
   4. depart\_longitude
   5. distance
   6. speed
   7. time\_stamp
4. trip\_data
   1. trip\_id
   2. trip\_gps\_latitude
   3. trip\_\_gps\_longitude
   4. trip\_time\_stamp
   5. speed
   6. bearing
   7. trip\_status (0 = start, 1 = enroute, 2 = complete)

# Billing data model

1. Trip data
2. Miles data
3. Payment data
4. Driver data
5. Passenger data
6. Vehicles
7. Documents
8. Taxi Owner
9. Address
10. Contact information
11. Payment method
12. Miles Account
13. Payment
14. Invoice
15. Campaigns

References

* Kalman filter for GPS smoothing/prediction
* Distance calculation best use Vincenty’s formulae (instead Harvesine)
* Quaternions for car pooling
* Declarative programming for optimization problems (Peter J. Stuckey)
* How to solve problems without knowing how

**Car pooling**

* Design data structure to hold a graph of the riders looking for vehicles
* This data structure will be a collection of paths and vertexes. Paths represent the rider trip (distance) and vertex represent start and end-point of each trip.
* Also ensure that the data structure is implicitly designed to deduce the directionality of the trip
* Also add timestamp and time-windows to the data structure
* One way to match passengers into a car-pool is to do the following: -
  + Calculate the individual distance of each trip (Trip A)
  + Compare this trip to the next trip buy calculating the total distance that this trip overlaps with the next trip (Trip B)
  + For trips with little or no overlaps, calculate the distance required to divert from the first passenger (Trip A) to pick-up the second passenger (Trip B) and drop them
  + A good way of doing this calculation is to calculate the two trips distance if 2 vehicles were used to fulfill these trips
  + Calculate the distance of both trips if a single vehicle was used and the trip was diverted to enable the 2 riders to be picked up in a single car
  + Compare these distances and calculate the cost to the entire system compared to using 2 vehicles
  + Use 2 data structures in redis, one set used to store the best placed trip to be pooled with Trip A based on its overall cost to the system (i.e. total distance)
  + Once complete with comparing Trip A to Trip B,C,D, proceed to compare all possible combinations by car-pooling Trip B with A, C,D etc.
  + Compare the cost to riders for sharing this trip compared to just each getting in their own vehicle
  + Investigate using minimum spanning tree to undertake these calculations