dCabS: Decentralized Cab Services Using Blockchain

Athreya Chandramouli
Roll No. 2018121002

Vijayraj Shanmugaraj
Roll No. 20171026

Devesh Vijaywargiya
Roll No. 20171132

I. Problem Statement

Existing Cab and Carpooling services rely on a central service such as OLA or Uber to connect drivers with the customers. We look to remove these central services which act as middlemen from this system with the help of a blockchain-based service run jointly by the customers and the drivers for the same, while maintaining popular features such as standardized fares, driver reputation etc.

A. Motivation

Long term drivers give Uber 20% of the total fare, whereas Medium term drivers are giving up 25% to Uber[1]. Uber was also caught spying on their users by Apple once in 2017, leaving their privacy at stake. Decentralisation of the cab-booking process would break the monopoly and control that the conglomerates possess right now.

B. Who will be the beneficiaries?

- Drivers can get the complete fee of their ride, and need not share any part of it to a parent company
- Customers would benefit too, since cab prices would significantly decrease due to the absence of a central authority and customers would have guaranteed privacy.

C. How big is the impact of your problem?

The net revenue of Uber in 2019 was $14.1bn[2]. This would get distributed among the customers, the drivers and miners involved in maintaining the decentralized application, and services get cheaper.

II. Solution

Our solution involves a decentralized application (dApp) that acts as an end to end platform for providing and availing cab services.

A. Architecture

We will be running the entire system on an Ethereum blockchain network, which will be maintained by the drivers and the customers. The location of drivers will be piggybacked along with the gossip protocol messages as a separate part of the message, so that the customer can have a look at this only when he/she tries to make a booking. The customer does not need this data until he/she makes a booking, and hence can discard the same, and save the blockchain alone.

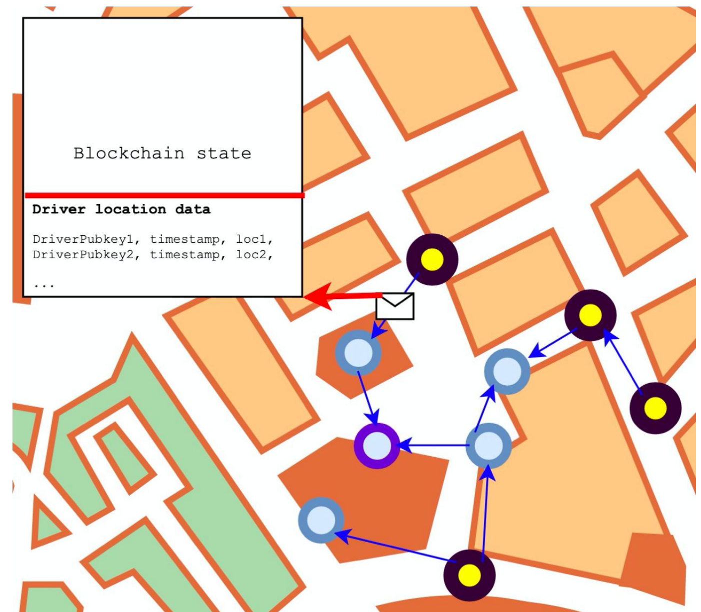

Fig. 1. Constituents of a gossip protocol packet

The flow of our dApp will be as follows-

1) While the driver interface of the dApp is active, along with the existing packets generated by the gossip protocol to sync the blockchain, the GPS coordinates of the driver is also sent.
2) When a customer wishes to book a cab, he enters his pickup and drop location on the customer interface of the dApp. The customer pays a flat fee to the contract.
3) The customer interface of the dApp will use the piggy-backed driver coordinates to determine drivers that are nearby.
4) The customer's GPS coordinates are encrypted using the public key of the nearest driver, along with details of the trip (destination and estimated time of arrival) and this is written to the blockchain.
5) In case the nearest driver wish to accept the trip, they can accept the trip by calling the relevant function in the contract. In case the nearest driver does not accept it, the user can cancel the previous request, and attempt

[1] https://www.quora.com/What-percentage-cut-does-Uber-take-from-the-total-fare-cost-of-a-ride-Do-they-subtract-a-flat-fee-for-each-dispatch-or-a-percentage-Are-there-initiation-monthly-fees-to-be-a-driver

[2] https://www.businessofapps.com/data/uber-statistics/

a request to the next nearest driver and so on. Once the trip starts,

6) The trip can be cancelled by either party, even midway. If the driver cancels it midway, he does not get any of the fee, and the customer gets a refund, whereas if the customer cancels the ride during the trip, the customer gets a refund proportional to the time period of the trip.
7) Once the destination is reached, both parties end the trip. The contract calculates the price of the trip.
8) Once the trip is complete, the contract logs the trip duration, driver rating, and source and destination locations. The locations are appended with a shared nonce, which will be unique to a particular trip, and hashed before being stored in the ledger.
9) The smart contract then forwards the payment made initially by the customer to the driver. And any remaining amount back to the customer.

B. Quality Control

Reputation: Since the blockchain stores the history of all trips made by a driver, it can be used to maintain a reputation score. However, anybody can generate multiple public keys and make fake trips to boost rating. To combat this, the dApp weights the contribution of each rating by the number of trips made with distinct drivers, and total trip duration. Hence a regular customer's rating will be much more influential on the driver's reputation than a newly created public key.

Incentive: The customer is also incentivized to stick to a single public key in order to make this reputation system effective. First of all, it is convenient to not create multiple keys. Second, the history of trips is stored such that only the customer's public key can be used to recover them. Finally, in case two customers book a cab simultaneously and they're matched with the same driver, the driver will choose the customer who has completed more trips with unique drivers in order to boost his reputation. Thus a customer who regularly uses the same public key will be given preference here.

Driver Registration: The driver will have to register himself into this dApp with a valid driver's license and a valid commercial vehicle permit. An official auditor who validates these documents will register him onto the blockchain network. In an ideal scenario, the driver's credentials are already present on the blockchain, however until that point we require some form of official verification.

C. Pricing

When the user and driver end the trip, the initial amount paid by the user can go to the driver. In the case that the user cancels the trip, the fee to be given to the driver can be calculated as :

$$rate \times (T_t - 10) + \epsilon$$

Where $rate$ is a fixed rate for travel per minute that the user will be obliged to pay if he cancels the trip, T_t is the time spent in travelling till the cancellation or trip end, and ϵ is a flat trip amount for the first ten minutes.

D. Smart Contracts

Below is the interface and a brief explanation for the smart contract and the structures and functions encapsulated by it.

```
contract dCabService{
    address public auditor;

    struct Trip, TripRequest, Driver, Customer;

    mapping(address => Customer) customers;
    mapping(address => Driver) drivers;
    mapping(address => TripRequest) activeTrips;
    mapping(address => Trip) currentTrip;

    function registerDriver public(address driverAddr, bytes_32 license){
        // Function to register a new driver into the system
    }

    function setPickup(bytes32 encryptedPickup, address driverRequested) payable{
       // Send a pickup request to a particular driver and pay a flat fee amount
    }

    function cancelRequest(address driverRequested){
        // Function for user to cancel his setPickup request
    }

    function acceptTrip(bytes32 driverPhoneNumEncrpyted){
        // Function for driver to accept the trip
    }

    function startTrip(){
        // begins the trip; the timestamp of this is used to calculate price
    }

    function getPrice(){
        // computes and returns the price based on timestamps;
    }

    function updateReputation(uint rating, address customer) {
        // updates the reputation of the driver
    }

    function updateUniqueDrivers(address customer, address driver) {
        // A helper function to calculate the reputation of the driver
        // if driver not in customers[customer].trips)
        //    customers[customer].uniqueDrivers += 1;
    }

    function endTrip(uint rating){
        // End trip function
    }

    function cancelTrip(){
        // Cancel trip function
    }
}
```

Listing 1. Interface of the smart contract being deployed for the service

1) Structures:

1) **Trip**

```
struct Trip{
        bytes32 pickupHash;
        bytes32 destHash;
        uint duration;
        address driverAddr;
        address customerAddr;

        uint price;
        uint rating;
        string comments;

        uint canEnd;
    }
```

Listing 2. The Trip Struct

The Trip struct has all the details of a completed trip. The pickup and the destination are hashed with the common nonce generated by the driver and the customer.

2) **TripRequest**

```
struct TripRequest{
        bytes32 encryptedPickup;
        address customer;
        uint driverPhoneNumEncrpyted;
        bool accepted;
    }
```

Listing 3. The TripRequest Struct

The TripRequest struct is pushed by the customer to make a request to a particular driver for a pickup. The pickup location is encrypted with the public key of the concerned driver. The accepted variable is used to denote whether the concerned driver has accepted the TripRequest or not.

3) **Driver**

```
struct Driver{
        bytes32 driverLicense;
        uint reputation;
        Trip[] driverTrips;
        bool registered;
    }
```

Listing 4. The Driver Struct

Structure for the driver details, which contains a list of trips he/she has, a verified hash of his driver's license and his reputation.

4) **Customer**

```
struct Customer{
        uint paid;
        uint uniqueDrivers;
        Trip[] customerTrips;
    }
```

Listing 5. The Customer Struct

Structure for the customer, which contains a list of trips he/she has made. Paid contains the amount paid by the customer initially. uniqueDrivers store the number of unique drivers the customer has taken a trip with.

2) Functions:

1) **registerDriver**

```
function registerDriver public(address
    driverAddr, bytes_32 license){
        require(msg.sender == auditor);
        drivers[driverAddr].driverLicense =
    license;
    }
```

Listing 6. The registerDriver function

This function can only be called by the auditor, in order to register the driver into the dApp.

2) **setPickup**

```
function setPickup(bytes32 encryptedPickup,
    address driverRequested){
        activeTrips[driverRequested].
    encryptedPickup = encyptedPickup;
    }
```

Listing 7. The setPickup function

This function is called when the user wants to send a pickup request to a particular driver. The user encrypts his location data with the concerned driver's public key and puts it up on the contract.

3) **acceptTrip**

```
function acceptTrip(bytes32
    driverPhoneNumEncrpyted){
        require(drivers[msg.sender].registered)
    ;
        activeTrips[msg.sender].accepted = true
    ;
        activeTrips[msg.sender].
    driverPhoneNumEncrpyted =
    driverPhoneNumEncrpyted
        enterTripDetails(msg.sender,
    activeTrips[msg.sender].customer,
    activeTrips[msg.sender].pickupHash,
    activeTrips[msg.sender].destHash  );
    }
```

Listing 8. The acceptTrip function

This function is called when the driver wants to accept the pickup request of a particular customer. The driver sends his phone number encrypted with the pubkey of the customer for contact. When the driver reaches the passenger, both the driver and the passenger can call the startTrip() function.

4) **cancelRequest**

```
function cancelRequest(bytes32 encryptedPickup
    , address driverRequested){
        activeTrips[driverRequested].
    encryptedPickup = encyptedPickup;
    }
```

Listing 9. The cancelRequest function

This function is called when the user wants to cancel a previous setPickup request in case he changes mind or the driver is unresponsive.

5) **endTrip**

```
function endTrip(bytes32 encryptedPickup,
    address driverRequested){
        currentTrip[msg.sender].canEnd += 1;
        if(currentTrip[msg.sender].canEnd == 2)
    {
            trip = currentTrip[msg.sender];
            trip.price = getPrice();
            trip.driverAddr.transfer(trip.price
    );
            trip.customerAddr.transfer(customer
    [trip.customerAddr].paid - trip.price);

            if(msg.sender == trip.customerAddr)
    {
                updateReputation(rating, trip.
    customerAddr);
                updateUniqueDrivers(trip.
    customerAddr, trip.driverAddr);
            }
            customers[trip.customerAddr].trips.
    push(trip);
            drivers[trip.driverAddr].trips.push
    (trip);
        }
    }
```

Listing 10. The endTrip function

This function is called by both the user and the driver to complete the trip. The fee is transferred, reputation of the driver is updated.

6) **cancelTrip**

```
function cancelTrip(bytes32 encryptedPickup,
    address driverRequested){
        trip = currentTrip[msg.sender];
        trip.price = getPrice();
        if(msg.sender is not a driver)
            trip.driverAddr.transfer(trip.price
    );
            trip.customerAddr.transfer(customer
    [trip.customerAddr].paid - trip.price);
        else transfer all fee back to customer

        customers[trip.customerAddr].trips.push
    (trip);
        drivers[trip.driverAddr].trips.push(
    trip);
        //Similar format to endTrip < Getting
    Rating etc.>
    }
```

Listing 11. The cancelTrip function

This function is called when either the user or the driver wants to cancel the ongoing trip. The appropriate fee is transferred to the driver, and the rest of the function is inline with the endTrip function.

III. Why do we need a blockchain?

Wust and Gervais provide a useful flowchart to decide whether a blockchain is required for a particular application. In our case, we do need to store state in order to record trips, maintain reputation scores, share contact information etc. There are clearly multiple writers: the various drivers and customers availing and providing the services. While we can use an always online trusted third party, we have highlighted concerns such as data privacy, service charges etc. which make decentralization very attractive. We also do not know the writers since any person should be able to use the app to avail rides. Hence the flowchart points us towards a permissionless blockchain such as Ethereum.

A. Why would the solution not be possible without a blockchain?

While a barebones, decentralized cab availing service can be developed using just peer to peer communication and end to end encrpytion without blockchain, several key features that make apps such as Ola and Uber unique would not be possible. Most importantly, there would be no way to judge whether the person responding is actually a reputed driver, or someone malicious: a crucial safety hazard! Apart from this, the blockchain provides a way to prove that each trip happened. It also provides a foolproof payment method that does not involve a third party.

1) What form of blockchain would be suitable?: Based on our requirements of identities of not every writer known beforehand, we would prefer to use a permissionless blockchain such as Ethereum to develop our use case.

IV. Analysis

We prevent the customer from overloading the system with requests by making them pay a slab amount upfront. This way, the number of requests the customer can make is constrained by the amount of money he is willing to spend. To regain his money, he must cancel his request, for which he will have to wait for a cooldown duration. We also have a cancellation fee for cancelling the trip in between, in order to avoid unnecessary cancellations.

A natural solution for being able to locate drivers in our system would be to have a central database that is updated every k seconds with the new location. This however adds an element of centralization that we wish to avoid. The logical next step would be to write these values periodically to the distributed ledger. However these values cannot be updated since the ledger can not be modified. Even if we conservatively update the location every 5 minutes, we would have to do 1,05,120 updates a year. Considering an estimate of 1.5 lakh drivers in Hyderabad[2] alone, we would have to update 15.8 billion values. This would translate to over 30 Terabytes stored in a year for Hyderabad alone. This is clearly infeasible. It also brings with it security issues since we have a complete history of the drivers' locations.

The solution therefore was a periodic broadcast message sent by each driver containing their location. However, we realized that since each user is already a client in the blockchain, it seemed natural to use the messages in the underlying gossip protocol used to sync up the blockchain. This results in a great amount of saving since we no longer have to send separate messages

[2] https://economictimes.indiatimes.com/small-biz/startups/uber-ola-drivers-hit-hard-by-falling-incentives/articleshow/56766997.cms?from=mdr

but rather simply piggyback them on the existing messages.

A major issue though, would be the amount of network bandwidth used. In order to maintain freshness of driver location, the push variant of the gossip protocol should be implemented for the system. But this would increase in increased bandwidth usage. Each message would also be slightly larger than before due to the piggybacked values.

V. Future Work

A. Dynamic Pricing

Uber and Ola have a dynamic pricing system based on which adjusts rates based on a number of variables, such as time and distance of the route, traffic, and cab availability, which can sometimes mean a temporary increase in price during particularly busy periods. As of now, the only factor being taken into account is the ride duration (which takes into account the distance and traffic). For surge pricing, a system can be implemented, where the smart contract can incorporate some form of crowd-sensing system within it to retrieve data on the number of drivers around the customer's location, and accordingly incorporate an appropriate factor into the ride fee. It would also be a good idea to somehow incorporate crucial factors such as petrol prices, road tax, and general inflation to periodically update the pricing model. An easy way would be to have an update even annually. However there can be better and more automated ways to access this data.

B. Solving network bandwidth issues

In order to solve the excessive bandwidth usage problem, we may have to devise a new data structure to compress the data of both the driver locations and the blockchain.

C. Latency

Since we are relying on the underlying gossip protocol messages, the frequency at which the driver locations are updated will be dependent on how active the underlying blockchain is. In case there are fewer transactions taking place, the driver locations will not be reflective of their actual positions.