CSCI5673 Spring 2020 Group Project
DeRiS: Blockchain based Ride Sharing

Nirvan S P Theethira
nith5605@colorado.edu

Zachary McGrath
zamc8857@colorado.edu

Abstract

Ride sharing is a concept that has gained popularity over the past decade with drivers offering up free spaces in their vehicles to riders in need for a price. While this has helped reduce congestion and automobile pollution, current ride sharing systems use a centralized structure where a single entity orchestrates interactions between riders and drivers. This leads to a single point of failure. As a consequence, this centralized structure raises concerns about data privacy and is vulnerable to attack from malicious users. Furthermore, as the central entity takes a certain cut of the profit from drivers who bear the brunt of the costs of gas and vehicle wear and tear. The propose solution to this problem is DeRiS, (De)centralized (Ri)de (S)haring. DeRiS, as its name implies, is a decentralized ride sharing system based on the Ethereum Blockchain.

Related work

Related work in this area includes implementations such as Ridecoin (1) and LaZooz (2) that exist, but are either just a concept or attempt to build their own cryptocurrency and/or blockchain. Instead, we choose to use Ethereum (3) for its wide adoption and support from its community. Others, such as Chasyr (4), are built atop the Ethereum blockchain, however their methods are proprietary and the methods and organization that they use can be assumed to be different to that of what is proposed in B-Ride, if for no other reason, that Chasyr (4) allows for riders to choose their driver. Other blockchain based ridesharing apps, such as DASCEE (5) and TADA (6) are region locked to countries outside of the United States. The B-Ride (7) paper goes into great detail about how to implement a secure system for decentralized ride sharing based on the Ethereum blockchain. The paper emphasized security in all rider and driver interactions. Most of the inspiration and ideas for this project are taken from the B-Ride (7) paper.

Proposed Solution

The proposed solution to the problem above aims to use a public blockchain that enables drivers to offer ride-sharing services eliminating the need for a third party (7). We hope to use Ethereum for the public blockchain (3). The Ethereum block chain is an open-source block chain with plenty of documentation for developers to build applications upon. Furthering the notion that the Ethereum block chain is a viable public option is that the Ethereum cryptocurrency is one of the most popular on the market. We hope to build a basic application that would pair a rider with an available driver using Ethereum smart contracts. We plan on building an application atop the Ethereum block chain. The smart contracts themselves will be written using Solidity (8). The application to interact with smart contract will be written in HTML/JS with the web3js package being used for interfacing with the smart contract (9).

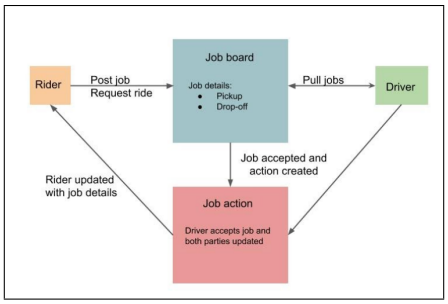

Figure 1: Rudimentary system design.

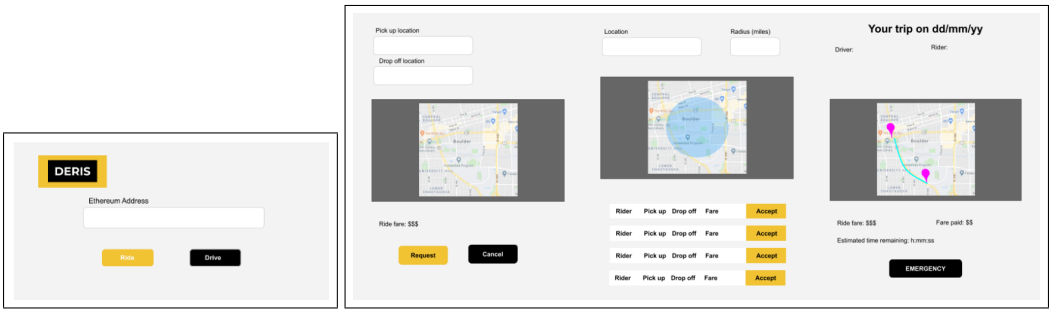

Figure 2: Rudimentary application design. On the top is the login page. On the left is the Rider's location selection page, in the middle is the driver's rider selection page and on the right is the trip in progress page.

System Design

As previously stated, DeRiS will be built upon the Ethereum blockchain. This open-source and well documented blockchain is ideal for development given the large community it has and widespread support. Many existing tools and frameworks for development already exist, making the overall development process cleaner and easier. As of any ride sharing application, the preliminary need is to have riders post there trip details and for drivers to select rides to be completed. A rudimentary design of this system can be seen in figure 1. The elementary design of the system can be outlined as follows:

- Rider post the start and end location of the trip along with his/her Ethereum account address to request a ride.
- The rider request is added to the blockchain.
- The Driver can now view all ride requests.
- Driver selects preferred rider.
- Rider pays the driver as the trip is completed.

The application to interact with smart contract will be written in HTML/JS with the web3js (9) package being used for interfacing with the smart contract (see figure 4). The first page is the login page which requires the user to input his/her/them Ethereum account address. This address will be used to identify the user and initiate transactions between users. The user has to then select if he/she/they is a rider or a driver. Depending on this selection they will be directed to the rider or the driver page. In the ride page the user has to select the start and end location of the trip to request a ride. The ride requested will show up on the driver page. The driver can view rides within a certain radius of the start location. The driver can then accept a ride which leads both the rider and driver to the trip in progress page. The in progress page outlines trip details as it completes. The pay as you ride algorithm as mentioned above deducts money from the rider and pays the driver as the trip is completed.

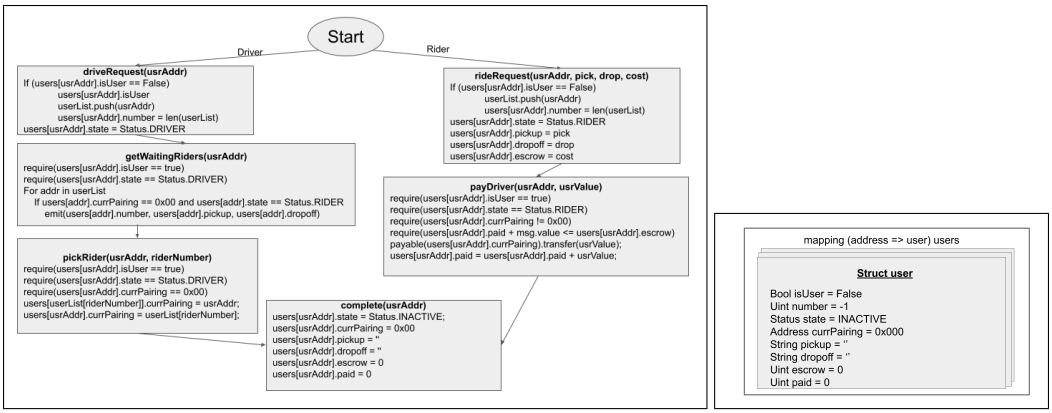

Figure 3: Left: Logical flow of execution of contract pseudo code. Right: Dictionary hash to store user address and related data struct.

Ethereum contract

As discussed in the proposed solution section, the back end blockchain portion of the application was built and deployed on an Ethereum test network. The test network was hosted by Ganache (10). All the data/transactions happening between the rider and driver was handled using the Ethereum blockchain. The Ethereum blockchain works on smart contracts which are essentially pieces of code that execute transactions between the rider and the driver. The smart contracts was written in solidity (8). All the logic between the rider and the driver discussed in the system design section is handled by the smart contract.

Data structures

The smart contract contains the following data structures to store user data:

- The **Status enum** enum is used to store the state of the user. A user can be in one of three states: INACTIVE, RIDER, DRIVER. Here an INACTIVE user is one that has either never used the app or one that has used the app in the past but is currently not using the app (aka waiting for a ride or waiting to drive someone). A RIDER is a user that is currently active and is waiting for a driver or is currently being driven. A DRIVER is a user that is currently active and is waiting to drive someone or is currently in the process of driving someone to a location.
- The **users mapping** mapping does most of the heavy lifting and stores information regarding application users (see figure 3). It is essentially a hash table that links a particular address to a particular **user Struct**. As a result of the way solidity handles mapping, ever possible address (key), even if it not yet been added (initialized), has a valid structure (value) associated with it. Therefore every address maps to a structure containing default values as seen in figure 3.
- **address [] userList** is a list of address of users that have ever used the application before. This is required to iterate through **users mapping** to find valid users. As discussed before, because of the way solidity handles mappings ever possible address (key), even if it not yet been added (initialized), has a valid default value. Therefore the a dictionary cannot be iterated to find all the keys (addresses) as every key is exists. This requires the storage of address in a separate **address [] userList**.
- The **user Struct** is what an address in the **users mapping** maps to. Each structure is used to store information regarding a single user (see figure 3).
  - **bool isUser** is a Boolean that is set to true if the user connected to an address has every used the application in the past. Its default value is false indicating the user has never used the application. This is required because of the way solidity handles mappings in which ever possible address (key), even if it not yet been added (initialized), has a valid default value. The **isUser** helps differentiate address that have used the app before vs ones that have not.
  - **Uint number** Stores the index of the user address in the **address [] userList**.

- **Status state** stores the status of the user as discussed in the first bullet point.
- **Address currPairing** store the address of the rider or driver the current driver or rider is paired with respectively. It is used to differentiate a waiting rider or driver from one that is currently in a trip. It is also used in payment to access the driver the rider has to pay.
- **String pickup** This is the pickup location in latitude and longitude for a rider. It is empty for a user who's current status in DRIVER.
- **String dropoff** This is the dropoff location in latitude and longitude for a rider. It is empty for a user who's current status in DRIVER.
- **Uint escrow** The cost of the trip calculated on the application side based on start and end location. This only applies to a rider. It is empty for a user who's current status in DRIVER.
- **Uint paid** As the driver is paid incrementally as the trip is completed, this is the amount that has been paid to the driver as the trip is completed. This only applies to a rider. It is empty for a user who's current status in DRIVER.

Program Flow

The smart contract has a set of function to operate on the above mentioned data structures. These functions are public and can be called by anyone on the blockchain in any order. The intended logical flow of function call along with the pseudo code of each function can be seen in 3. Depending on weather the user initiates a session as a driver or rider, there are two possible way the program flow could move. If the user starts the session as a driver (intends to give rides to other users) the program flows in the following direction:

- The **driveRequest** function is called. The user address is passed to function based on the account the contract was called from. The address value is stored in **msg.sender** as per solidity syntax. To make the pseudo code simpler in 3 this value is called **usrAddr**. As seen in the pseudo code, if the user already exists, the users state is changed from INACTIVE to DRIVER. If the user does not exist the address is added to the **userList** and the user number is updated along with the **isUser** status.
- The drivers application then calls the **getWaitingRiders** function to get the list of active riders. The function uses the require Solidity syntax, which is essential an assert, to check if the caller is a valid user and if the caller is currently and active driver. The function then finds all the users that are currently active riders who are not yet paired with a driver and emits the rider number along with the riders pickup and drop off location to the driver. Emit is a solidity function that is used to mimic events. The application code subscribes to the event and is asynchronously updated every time an event occurs. This way the driver application can asynchronously collect and update the list of active waiting riders.
- Once the driver picks a rider on the application the **pickRider** function. Notice a rider number is used to identify riders thereby never disclosing the rider account address. The **pickRider** function checks to see if the caller is a valid user and if the caller is currently and active driver who is not yet matched. The rider and driver structures are updated to create a pairing.
- The driver is paid as the trip is completed and once the trip is complete calls the **complete** function to reset all stats.

If the user starts the session as a rider (intents to hitch a ride) the program flows in the following direction:

- The **rideRequest** function is called. The user address is passed to function based on the account the contract was called from. If the user already exists, the users state is changed from INACTIVE to RIDER. If the user does not exist the address is added to the **userList** and the user number is updated along with the **isUser** status. The riders pickup and drop off location along with the estimated cost of the trip is also added to the structure.
- The rider keeps waiting until picked by a driver. Once picked up by the driver and as the trip is completed, the rider application initiates payment depending on the amount of the trip that

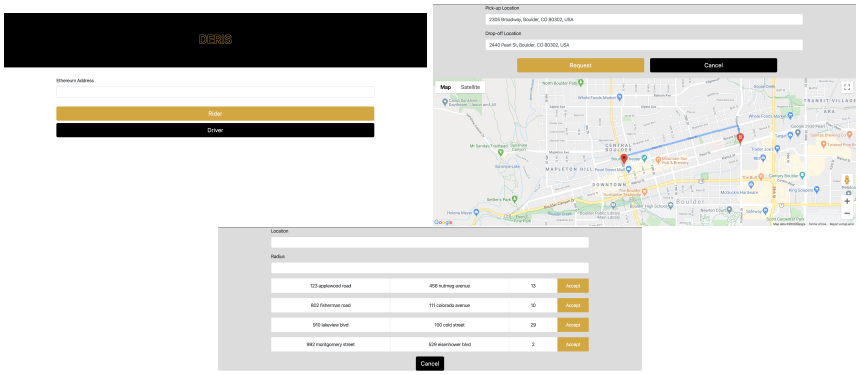

Figure 4: Current application design. On the left is the login page. Centered is the rider page. Google maps is loaded onto the page with a route chosen from the two markers. On the right is a rudimentary driver page layout. To be added to driver page is a map to allow the driver to choose a centered location.

is completed. This is done by calling the **payDriver** function. The function first checks if the caller is a valid user and if the caller is currently and active rider who is currently paired with a driver. The function also checks if the amount already paid is less than or equal to the amount owed. This is done to prevent over payment. If the checks are successful, the amount send by the rider (stored in msg.value as per solidity syntax) is sent transferred to the driver account. The amount paid to the driver is updated.

- Once the trip is complete, the rider application calls the **complete** function to reset all stats.

Application

The user interface is a single page JavaScript application. The application is built using the React framework. This framework allows for quick development of modular components to simplify front end development.

Interaction with the blockchain

The application interfaces with the Ethereum blockchain as little as possible. Due to the rather slow nature of interacting with the blockchain, much of the computation is handled on the clients device. There are, however, a few instances where interaction is required. These situations, in no particular order, are

- Pushing a rider's trip request to the blockchain
- Pulling current jobs for the driver to look through
- The communication between driver and rider during the ride to ensure payment

User interface

The current iteration of the user interface attempts to be as user friendly and as simple as possible for the user. Figure 4 shows the current status of the login page, the rider page, and the layout for the driver page. As stated previously, React framework is being used to design the application. The embedded Google Maps makes the application more user friendly as users can see and click on their desired pickup and drop off locations.

React

The user interface is built using React. React works by creating **stateful** and **stateless** components. Stateful components hold data where as stateless components merely perform functions and therefore are often implemented as functions rather than classes. For our single page web app, the highest level

class, the **App** class, holds all the relevant pages. Each page is then just a component and rendered as needed as the user interacts with the application.

For the text inputs, buttons, and other common inputs, the implementation is rather straightforward; however the Google Maps component becomes rather tricky. Currently, the module refreshes after every click on the map which re-renders not only the current page, but also the Map component. This becomes a problem for slower networks. However, the utility of being able to click on the map over typing in the desired address is worth it in the short term.

Evaluation

A fair evaluation of a prototype for this application would be the ability to set up an account as a driver or a rider, the ability to book a ride, the ability to send a location for pick and drop off and the ability for the driver to get paid by the rider as the trip is completed. These initial goals would prove the feasibility of the project. As of the second report, the contract for all block chain transactions is built and has been deployed on a test network. Baring a few modifications to the contract, the back end block chain is functional. In the front end, the login page along with the rider and driver waiting pages are ready. The next steps would be to complete the trip in progress page and to integrate the front end with the backend.

References

[1] Ride coin. [Online]. Available: https://www.fairride.com

[2] Lazooz. [Online]. Available: http://lazooz.org/

[3] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger."

[4] Chasyr. [Online]. Available: https://www.chasyr.com/

[5] Dascee. [Online]. Available: https://dacsee.com/#/

[6] Tada. [Online]. Available: https://tada.global/

[7] M. Baza, N. Lasla, M. Mahmoud, G. Srivastava, and M. Abdallah, "B-ride: Ride sharing with privacy-preservation, trust and fair payment atop public blockchain," *IEEE Transactions on Network Science and Engineering*, pp. 1–1, 2019.

[8] Á. Hajdu and D. Jovanovic, "solc-verify: A modular verifier for solidity smart contracts," *CoRR*, vol. abs/1907.04262, 2019. [Online]. Available: http://arxiv.org/abs/1907.04262

[9] web3js developer page. [Online]. Available: https://web3js.readthedocs.io/en/v1.2.6/

[10] Ganache truffle suite. [Online]. Available: https://www.trufflesuite.com/ganache