Taxicoin: A Decentralised Taxi Protocol

Scott Street
159027651

April 2018

Abstract

Taxicoin is a decentralised taxi protocol, developed with openness and fairness in mind, and built on Ethereum. This dissertation details the background to the problems Taxicoin attempts to solve, as well as an explanation of the Ethereum network. It then describes the protocol itself, how it has been implemented, and what steps have been taken to ensure the security, reliability and effectiveness of the implementation.

Contents

1 Introduction

Taxicoin is an attempt at designing and building a protocol for hailing taxis, where the entire system is fully decentralised, with no single authority in control. The motivation behind this is to combat some of the issues found with existing similar traditional applications, such as *Uber* and *Lyft*.

These companies saw a way to improve the taxi industry and, by improving the user experience and ease of ordering a taxi, attracted many users. However, as was the case with existing taxi companies, they still take a significant cut of fares. Coupled with the fact that they attempt to keep fares lower for passengers, the drivers are left with very little earnings.

In contrast, Taxicoin has been designed so that the will of riders and drivers self-regulates fares. A driver decides and quotes fares on an individual basis, rather than being instructed as to what fare they should charge. Riders then only accept fares that they feel are fair – if they are in a hurry, they may be willing to pay a higher price.

1.1 Benefits of Decentralisation

In general, decentralised systems are more open. In context, this means that anybody is free to participate – one of the key precepts of Taxicoin.

Additionally, such systems, if designed well, should be not in the control of any one group of people. If we look at traditional taxi companies, the customers are at the will of the company. In many cases, one company will form the sole taxi coverage of a territory, meaning that they can decide on the price for a journey.

With a decentralised system, decisions about such issues are made transparently between all parties involved. In short, decentralisation creates a fairer system.

1.2 Why an Open Protocol

Continuing with the themes of decentralisation, it was important that Taxicoin be as open as possible. If a single entity was tasked with developing and maintaining the Taxicoin network, and decided to cease doing so, the entire system would likely collapse. It also creates a *walled garden* scenario, where the single entity has total control over the system. They could decide at any point to disregard the original decisions made behind the network, and start charging a fee for each journey.

But with an open protocol, the specification needed to implement ones own version of Taxicoin is freely available. If users in a certain location beyond that which is actively supported by the initial version wish to launch their own Taxicoin, they can. It also allows for continued critique and improvement, where potential bugs can be discovered and fixed, and if a developer external to the original project wishes to write an extension to the protocol to add further features, they may do so.

2 An Explanation of Ethereum

Ethereum is a platform consisting of three components: Swarm, a "distributed storage platform and content distribution service" [1]; Whisper, a peer-to-peer communication protocol [2]; and the Ethereum Virtual Machine (EVM) used for running smart contracts [3]. The latter is often referred to alone simply as "Ethereum", however all three should be considered part of the same platform, each one complimenting the others. The aim of this section is to explain how these three solutions are used together to develop fully decentralised applications, or dApps.

2.1 Blockchains

The EVM component of Ethereum is built on top of a "blockchain", a term coined by the anonymous creator of Bitcoin [4], which is the original, and most widely known application of such technology. At its core, a blockchain consists of transactions, grouped together into "blocks", with each group also referencing the previous one, thus forming a "chain". A blockchain can be thought of simply as a form of database, keeping a state. A transaction represents a state transition, but must be verified before being deemed to be valid and placed in a block.

The blockchain itself is distributed across all nodes in the network (except in cases where a node chooses to reference another's copy), meaning that, unless explicitly obfuscated by the user, all transaction data on the network is open. This allows the auditing of transactions by any node on the network, and eliminates the need to trust a single entity to provide accurate data - this is the concept of trustlessness.

Transactions on the Bitcoin blockchain are, for the most part, simply that - transactions. They represent a transfer of funds from one "address" to another. They can additionally contain an amount of data, representing anything from a simple message, to a method call in cases where the receiving address is a "smart contract" (or simply "contract").

In Bitcoin, contracts are a special type of address which causes nodes on the network to execute some predefined code when a transaction is sent to it. Contracts are deployed by a standard (human controlled) address, but once deployed act as independent entities. Unless their code contains functionality to do so, the deployer has no control over the contract.

However, contract execution on the Bitcoin network is not Turing complete, due to the halting problem - the inability to determine whether a section of code will complete execution without looping infinitely. If contract execution was Turing complete in the existing Bitcoin network, a malicious actor would be able to perform a denial of service attack against the network by deploying and calling contact code containing an infinite loop.

This is where the EVM differs, with the addition of "gas". This introduces a fee per instruction to be executed (paid in Ethereum's native cryptocurrency, Ether). The sender of a transaction sets the maximum amount of gas they are willing to spend for a transaction to complete, and a contract call will continue executing until either the execution completes, or the maximum amount of gas is consumed. This safely allows the use of loops within contracts, as it becomes very expensive to perform an infinite-loop attack.

The result is that the EVM is Turing complete, and thus in theory any arbitrary program can be implemented in a contract, opening the door to a wide variety of applications.

2.2 dApps

While smart contracts are well suited to taking inputs, making state changes, and producing outputs, that is all they do. It is possible to interact with them via a command line interface, through an Ethereum node, however this is obviously far from the desired experience for end users. To address this problem, several attempts at providing a user interface layer for the Ethereum network have been introduced. The most widely adopted, and officially endorsed solution, is Web3 - a browser API which allows interacting with all parts of the Ethereum platform from Javascript embedded on a web page.

This leads to the approach that many Ethereum dApp developers take: considering their application as a traditional "Single Page App", where instead of calling HTTP API endpoints, they are now interacting with a smart contract through the use of Web3. Smart contracts effectively take the place of a "backend" web server, leading to many benefits over traditional web apps, such as availability, security and integrity.

Coupling this with Whisper allows for peer-to-peer communication between instances of a dApp, which in most cases translates to between different users. For example, two parties negotiating the price of an item to be purchased - they do not necessarily want their negotiation to be public (or rather, it does not provide any value for it to be), therefore they can come to an agreement "off-chain" before publishing (sending) a transaction of the final agreed price. Whisper is also beneficial for situations where the sender of a message wishes to remain anonymous. When publishing a transaction on the blockchain, the sender is published along with it, whereas in Whisper, unless signed, it is improbable to determine the sender of a message [cite].

Additionally, the static HTML, Javascript and any additional components of a dApp can be hosted from Swarm (or a similar platform such as IPFS [cite]). When a file, or set of files, is published to Swarm, a hash is computed, and the file is split into pieces called "shards". The shards are then distributed across nodes in the network, with the intention that if one node becomes unavailable, the shards of the file should still be accessible. When a user wishes to retrieve a file at a later date, they can provide the previously computed hash to a Swarm node, which will request shards of the file from its connected peers.

In this manner, it is possible with the Ethereum platform to develop fully decentralised applications where the user interface is written as a web page and is served from Swarm, thus eliminating the requirement for a traditional web server. An application's "backend" logic is contained within a smart contract, removing the need for a backend web server such as PHP. And finally, instances of the application may communicate between each other through the use of Whisper, removing the need for a solution such as WebSockets, where a central signalling server is required.

2.3 Solidity Language

Todo.

2.4 Smart Contract System Design

As smart contract execution is only ever triggered as a result of a transaction, applications must be designed around deliberate actions. For example, where in a traditional system, a method may be set to execute at a particular date and time, in a smart contract this is not possible. Instead such a method may only have a check for if the allowed time of execution has passed, and must be manually triggered by a transaction.

As the reasons for some of these differences are unlikely to be clear to users, it is important to consider how to communicate them.

Additionally, as there is an attached "gas" fee for publishing transactions and calling contract methods, it is in the interests of the users for the contract developer to make contracts as efficient as possible, and to make a minimal number of contract calls in a dApp. One way of doing this is to avoid on-chain interaction wherever possible, through the use of peer-to-peer protocols, predominantly Whisper. In extreme cases, complex routines within contracts can be written in the underlying EVM byte-code for improved efficiency.

2.5 Why Ethereum

Todo.

3 Identification of Requirements

The first step towards developing Taxicoin was to identify the requirements for the resulting system. These were split into two categories: minimum viable and additional requirements.

3.1 Minimum Viable Requirements

These requirements are those which must be included in order for the system to correctly function.

Drivers must be required to pay a deposit in order to advertise to act as a reasonable barrier to entry. Without this in place, the network is easily open to spam and scammers posing as drivers. The deposit acts as an incentive to behave well.

Riders must advertise jobs to drivers on an individual basis in order to protect the privacy of the rider. As this is likely to contain individually identifying information, such as location, if this were published it could be used to track an individual.

The fare must be determined by quotes from driver to remove the need for a centralised fare decision. This is due to the fact that the fare depends on many factors which cannot be automatically determined in a decentralised and reliable manner, such as distance and demand. The alternative would be fixed fares, but this is highly undesirable as short trips would be overpriced, and long trips underpriced.

Riders must pay fares to a contract in advance as a security measure, due to the fact that there is no other way to guarantee riders will pay after the fact. Without this, it is likely that a subset of riders would not pay for journeys.

Riders must provide an additional deposit before starting a journey to act as an incentive to successfully and formally complete a journey in the system. Without this, riders may have paid fare and have no regard for consequences of bad acts. Additionally, they may not carry on to rate the driver, an integral part of the smooth running of the system.

Riders and drivers must both rate the other on completion of a journey to affect the reputation of the other party. This is likely to be implemented such that a user is unable to interact with the rest of the system until they have formally completed their previous journey. Without this requirement, there is no way to determine the trustworthiness of another individual on the network, which is key to preventing bad behaviour.

When a journey is completed, deposits should be returned to the respective parties, and the fare paid to the driver this ensures that riders and drivers both have a stake in formally completing a journey. If they do not, their deposits are not returned, and neither is the driver paid. Without this deposit system, there is no guarantee that either party will rate the other - potential hit-and-run scenarios could occur where a rider uses the system only once and does not care to formally complete a journey and rate their driver as it provides no benefit to them. With the deposits however, they are likely to complete the process, at stake of losing their funds.

3.2 Additional Requirements

These are requirements deemed as "nice to have" features, without which the system will continue to function, but the addition of which would improve the system in some way.

Prospective drivers and riders should be able to informally communicate before forming a contract to allow any additional requirements on either part be known. For example if a rider is wishing to take a large, bulky item on the journey with them, they may communicate this in advance. If it transpires that the driver's car is small, the journey can be cancelled (or not formally begun), and another driver arranged, before the original driver has taken the effort of travelling to pick up the rider.

Dispute resolution should be built into the system for situations where driver and rider are unable to successfully complete a journey. This would work in a similar way to negotiating a price. In a worst case scenario, the driver wants payment in full, but the rider wants to pay nothing. In this case, the two negotiate until an agreement is reached. If they do not reach such an agreement, it reflects poorly on both parties, as the fact that they have an unresolved dispute is public. The system is able to function without this, but bad disputes are likely to go unresolved which is dissatisfactory.

4 Protocol Specification

The protocol portion of Taxicoin is designed to be open. As such, anybody should be able to implement it in their own software. The following section of this document should be sufficient to do so.

4.1 Methods

Each of these methods is intended to be part of a smart contract. When one is called, it will modify the state of the contract, and/or return a value.

The specified arguments are to be supplied when calling that function of the contract, with the types representing built-in Solidity language types. The *payable* keyword indicates that a method accepts a transaction with a currency value attached. In instances where the preconditions for a method are not met, the method will revert and the state will be unmodified.

4.1.1 Driver Advertise

Description	Takes a deposit from a driver and publishes their location and public key.
Arguments	Latitude: String Longitude: String Public Key: String
Payable	Driver deposit
Preconditions	User must not be currently on a journey, either as a driver or rider. Deposit must either have been already provided, or sent with this transaction.
Postconditions	The driver's location and public key are published, and the value of the deposit provided by the driver is recorded. If any deposit over the required amount was provided with the transaction, the excess is returned.

4.1.2 Driver Advert Revoke

Description	Removes an active driver's advertisement.
Arguments	None
Payable	No
Preconditions	User must be advertised as a driver.
Postconditions	The driver is removed from the list of active drivers, indicating that riders should not send job proposals to this driver. The previously supplied deposit is not returned.

4.1.3 Rider Create Journey

Description	Accepts a quoted fare for a journey as a rider and forms the rider's part of a contract between driver and rider. Intended to be called after an off-chain negotiation with `job` and `quot` messages.
Arguments	Driver Address: address Fare: uint, value in *wei* $n > 0$ Public Key: String
Payable	Fare plus rider deposit

Preconditions The user at the provided address must be an actively advertised driver, and not currently on a journey. The user calling this method must not be an actively advertised driver, nor be part of a journey as either rider or driver. The full rider deposit, plus an amount equal to the provided `fare` must have been provided with this transaction.

Postconditions The rider's intent to travel with the specified driver at the specified price is published. At this stage, the agreement is not binding until the driver accepts, before which the journey may be cancelled, with the rider deposit and fare being returned in full.

4.1.4 Rider Cancel Journey

Description Cancels a journey which has not yet been accepted by a driver.

Arguments None

Payable No

Preconditions Rider must be part of a journey, for which the driver has not already accepted.

Postconditions The rider is removed from the journey. From this point it is no longer possible for the driver to accept the journey. The rider's deposit and fare are returned.

4.1.5 Driver Accept Journey

Description Formally accepts a job as a driver, committing both the rider and driver to its completion.

Arguments Rider Address: address
Fare: uint, value in *wei* $n > 0$

Payable No

Preconditions Driver must be actively advertised and have provided the driver deposit. Rider must have formally created a journey with the driver set to the caller of this method, and the fare of equal value to the argument provided.

Postconditions The driver is marked as being on a journey with the specified rider. From this point, the journey is considered to be in progress, and any attempt to change any aspect of the journey will require an agreement to be made between both rider and driver.

4.1.6 Complete Journey

Description Marks the current journey as completed, as either the rider or driver.

Arguments Rating: uint8, $1 \leq n \leq 255$ with 255 being the "best"

Payable No

Preconditions The caller of the method must either be a driver or rider who is currently on a journey.

Postconditions The caller of the method is marked as having completed the journey, however they are still part of this journey until the other user has also called this method. The rating for the other user is stored. If the other party has already called this method, then the ratings for both parties are applied to their overall rating, and the journey is formally completed. The rider and driver deposits are returned, and

the fare transferred to the driver. However, in cases where the fare is zero (only possible where the fare has been altered during a journey to indicate that the journey should be cancelled), the driver's deposit is not returned.

4.1.7 Driver Propose Fare Alteration

Description	Formally proposes the alteration of the fare for a journey. Intended to be called after an off-chain negotiation with `Propose Fare Alteration` messages.
Arguments	New Fare: uint, value in *wei*
Payable	No
Preconditions	The user calling the method must be a driver, and currently be on a journey.
Postconditions	The driver's proposed new fare is recorded. The new fare does not take effect until the rider calls the `Confirm Fare Alteration` method.

4.1.8 Rider Confirm Fare Alteration

Description	Confirms the alteration of the fare for a journey.
Arguments	New Fare: uint, value in *wei*
Payable	Difference between old and new fares, if new is higher
Preconditions	The user calling the method must be currently on a journey. Driver must have previously agreed the same new fare with the `Alter Fare` method. In the case that the new fare is higher, the difference must have been provided with this transaction.
Postconditions	The new value for the fare for the journey is recorded. In the case that the new fare is lower, the difference is returned to the rider. If the new fare is zero, the journey is considered to be cancelled - the journey may now be completed with the rider's deposit being returned, and no fare being paid to the driver.

4.1.9 Get User Type

Description	Returns an integer representing the type of user at the provided address.
Arguments	User Type: uint8
Payable	No
Preconditions	None
Postconditions	Returns an integer between 0 and 3, representing the enum { None, Driver, ActiveDriver, Rider }.

4.1.10 Get Driver

Description	Returns the details of the driver at the given address.
Arguments	Driver Address: address
Payable	No
Preconditions	None
Postconditions	If the address provided is of a user who has previously (or is currently) advertised as a driver, the details of the driver will be returned. Otherwise, all zero-values will be returned.

4.1.11 Get Next Driver

Description Returns the details of the driver next in the list of advertised drivers after the given address.

Arguments Driver Address: address

Payable No

Preconditions None

Postconditions If the address provided is of a user who has previously (or is currently) advertised as a driver, the details of the next driver in the list will be returned.

If the zero-address (`0x0`) is provided, the first driver in the list is returned.

If the address provided is for the last driver in the list, the zero-address is returned.

4.1.12 Get Previous Driver

Description Returns the details of the driver previous in the list of advertised drivers after the given address.

Arguments Driver Address: address

Payable No

Preconditions None

Postconditions If the address provided is of a user who has previously (or is currently) advertised as a driver, the details of the previous driver in the list will be returned.

If the zero-address (`0x0`) is provided, the last driver in the list is returned.

If the address provided is for the first driver in the list, the zero-address is returned.

4.1.13 Get Rider

Description Returns the details of the rider at the given address.

Arguments Rider Address: address

Payable No

Preconditions None

Postconditions If the address provided is of a user who has previously used (or is currently using) the system as a rider, the details of the rider will be returned.

Otherwise, all zero-values will be returned.

4.2 Messages

Driver and rider user clients should be listening for the following messages, where applicable. These messages are communicated via the Whisper protocol.

Message topics are always a length of 4 bytes (4 ASCII characters), therefore any topics listed here of a length less than 4 byes are right-padded with spaces.

4.2.1 Job Proposal

Topic `job`

Purpose This message is sent by a rider to a prospective driver, indicating that they wish to make the described journey. It is intended to be sent to advertised drivers matching a specified criteria, e.g. within a certain distance, with at least a certain reputation.

However the sending of these messages is not intended to be carried out manually by the user – rather there is an automated process which fetches the list of active drivers and determines which to propose to.

Response Should a driver be interested in a proposal, they respond with a quote message.

Payload Please see below.

```
{
  "pickup": {
    "lat": String,
    "lon": String
  },
  "dropoff": {
    "lat": String,
    "lon": String
  },
  "address": String
}
```

4.2.2 Driver Quote

Topic `quot`

Purpose This message is sent by a driver as a response to a job proposal. It contains the network address of the driver, as well as the fare for which the driver is willing to take on the job. At this point, the quote is not binding.

Quote messages with a fare of -1 are considered to be a rejection, indicating that the driver does not wish to accept this job.

Response If the rider chooses to accept the quote, they next call the create journey method, and respond with a `Journey Created` message.

Payload Please see below.

```
{
  "address": String,
  "fare": Integer
}
```

4.2.3 Journey Created

Topic `crea`

Purpose This message is sent by a rider to a driver after they have created a journey. This is an indication that the rider has accepted the driver's quote.

Response The driver should next call the `Driver Accept Journey` method, and respond with a `Journey Accepted` message.

Payload Please see below.

```
{
  "address": String,
  "fare": Integer
}
```

4.2.4 Journey Accepted

Topic `accp`

Purpose This message is sent by a driver to a rider, after they have formally accepted the rider's journey, to indicate that both parties are now on a journey.

Response None

Payload Please see below.

```
{
  "address": String,
  "fare": Integer
}
```

4.2.5 Driver Location

Topic `lctn`

Purpose Sends the location of the driver to the rider with whom they are currently on a journey with. Allows the rider's client to display how far the driver is from the pickup location.

Response None

Payload Please see below.

```
{
  "location": {
    "lat": String,
    "lon": String
  }
}
```

4.2.6 Journey Completed

Topic `cmpl`

Purpose This message is sent to the other party when either one calls the `Complete Journey` method. It indicates that the other should also (or dispute it). If the other party has already completed the journey, then this message indicates that the journey is fully complete.

Response None

Payload None

4.2.7 Propose Fare Alteration

Topic `nfar`

Purpose This message is sent to the other party to indicate that the user wishes to alter the fare for the current journey.

Response If the message is received by a driver for the first time, unprompted, they may either agree with the new proposed fare, and formally propose the new fare with the `Propose Fare Alteration` method, or reject the new fare by sending a message of this type with an alternate fare.

If the message is received by a driver for the second time, after agreeing to the proposed new fare, and the value in this message is unchanged, this indicates that the rider has called the `Rider Confirm Fare Alteration` method, and the new fare has been applied. No further response is sent. However if the fare was changed, the driver may act as if this is the first such message (see above).

If the message is received by a rider for the first time, unprompted, the rider may either agree to the new fare, and respond with a message of the same type with an unchanged value, or they may disagree and respond with their proposed new fare.

If the message is received by a rider for the second time, after agreeing to the proposed new fare, and the value is unchanged, this indicates that the driver has called the `Propose Fare Alteration` method, and that the rider should call the `Rider Confirm Fare Alteration` method. They then respond with a message of this type, with the fare value unchanged.

Payload Please see below.

```
{
  "fare": Integer
}
```

4.3 Contract Solidity Interface

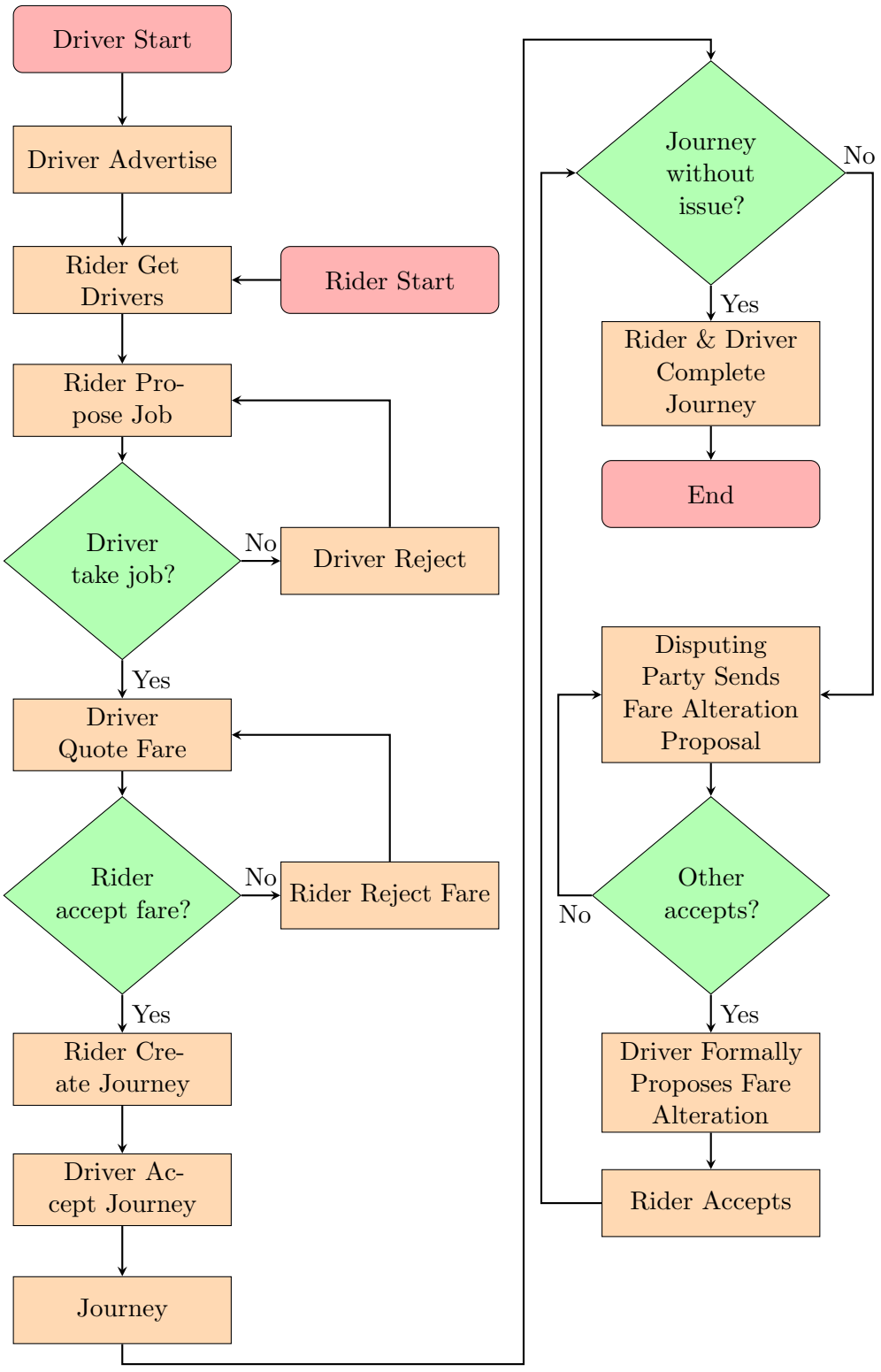
The above methods can be translated to a Solidity interface, which should be conformed to for all Ethereum-based contracts implementing the Taxicoin protocol. This aids the goal of creating an open ecosystem as, in theory, if all implementations conform to this standard, any client should be able to work with any contract implementation.

```
pragma solidity ^0.4.21;

contract ITaxicoin {

  function driverAdvertise(string lat, string lon, string pubKey)
      public payable;

  function driverRevokeAdvert() public;

  function riderCreateJourney(address driver, uint fare, string
      pubKey) public payable;

  function riderCancelJourney() public;

  function driverAcceptJourney(address rider, uint fare) public;

  function completeJourney(uint8 rating) public;

  function driverProposeFareAlteration(uint newFare) public;

  function riderConfirmFareAlteration(uint newFare) public payable
      ;

```

```
21    function getUserType(address addr) public view returns (uint8);
22
23    function getDriver(address driverAddr) public view returns (
          address addr, string lat, string lon, string pubKey, uint
          updated, address rider, uint deposit, uint8 rating, uint
          ratingCount, uint8 riderRating, uint proposedNewFare, bool
          hasProposedNewFare);
24
25    function getNextDriver(address driverAddr) public view returns (
          address addr, string lat, string lon, string pubKey, uint
          updated, address rider, uint deposit, uint8 rating, uint
          ratingCount, uint8 riderRating, uint proposedNewFare, bool
          hasProposedNewFare);
26
27    function getPreviousDriver(address driverAddr) public view
          returns (address addr, string lat, string lon, string pubKey,
           uint updated, address rider, uint deposit, uint8 rating,
          uint ratingCount, uint8 riderRating, uint proposedNewFare,
          bool hasProposedNewFare);
28
29    function getRider(address riderAddr) public view returns (
          address addr, string pubKey, address driver, uint fare, uint
          deposit, uint8 rating, uint ratingCount, uint8 driverRating);
30
31 }
```

4.4 Process Flow Diagram

This describes the possible routes of interaction through the system.



5 Implementation

My implementation is in two parts: that of the protocol described above, and additionally an example *Web3* client. The protocol itself is split between peer-to-peer communication and smart contract methods. P2P messages are handled entirely outside of the contract, and in this implementation are sent by the client library via the Whisper protocol on the Ethereum network.

The entire implementation of the Taxicoin protocol and client is contained within one git repository. The directory structure is defined by a combination of `vue-webpack`[cite] and `truffle`[cite], both of which have strong opinions.

5.1 Smart Contract

As previously discussed, Ethereum smart contracts are self-contained programs which execute on the Ethereum Virtual Machine (EVM). The core of Taxicoin is implemented as a contract, allowing riders and drivers to interact with one and other with it as an intermediary.

The contract is centred around state, which can be divided into three parts: the state of the Taxicoin network, the state of a driver, and the state of a rider. It conforms to the `TaxicoinInterface` contract interface, as defined in the protocol specification.

5.1.1 Network State

When a contract is deployed, various parts of it may be initialised, in a similar way to how the constructor of a class initialises the state of an object. In Taxicoin, only the driver and rider deposit values are set at this stage.

Two key mappings are kept: one which maps an address to a driver, and one which maps an address to a rider. There is additionally a third mapping used to implement a double-linked list (DLL), discussed below.

There is also a `UserType` enum from the contract interface, which is used as a return type for the `getUserType` helper function. This is an easy method of determining the current "mode" of a user based on the state of their `Driver` and `Rider` objects (described in more detail below).

```
1  enum UserType {
2    None,
3    Driver,
4    ActiveDriver,
5    Rider
6  }
```

5.1.2 Driver State

An individual driver is represented by a `Driver` struct:

```
1  struct Driver {
2    address addr;
3    string lat;
4    string lon;
5    string pubKey;
6    uint updated;
7    address rider;
8    uint deposit;
9    uint8 rating;
10   uint ratingCount;
11   uint8 riderRating;
```

```
12    uint proposedNewFare;
13    bool hasProposedNewFare;
14  }
```

When a user first wishes to become a driver, a `Driver` object will not exist for them[1], and they will have to call the `driverAdvertise` method, providing their current latitude, longitude, Whisper public key (used for peer-to-peer communication), and a deposit. If the deposit provided is sufficient, then the driver's state will be updated.

This consists of setting the address of the driver on the object (used as an integrity check - the address of an advertised driver should map to a `Driver` object with the same address), the latitude and longitude, the time at which the driver was last updated (used to detect stale advertisements), the deposit provided by the driver (for cases where the global driver deposit value may change, the amount provided when the driver initially advertised is what will be returned), and the driver's public key (used for contacting this driver via Whisper). Additionally, if the driver is not already advertised, their address is added to the DLL.

Just as the address of a `Driver` object is used to check integrity, it is also used to indicate whether a driver is currently advertised or not. Any user should be able to view information about a driver at any time, and the overall rating of a driver needs to be stored even while a driver is not advertised. Therefore, this data is kept, and can be accessed via the `drivers` map. However, if the address does not match, this indicates that the driver is not currently advertised.

To mark a driver as not active, they are removed from the DLL and their address set to zero. To mark a driver as on a journey, they are removed from the DLL and their rider is set to a non-zero address. The advantage of this state-based approach to determining the mode of a driver is that we do not have to explicitly store and update a separate indicator.

5.1.3 Rider State

An individual rider is represented by a `Rider` struct:

```
1  struct Rider {
2    address addr;
3    string pubKey;
4    address driver;
5    uint fare;
6    uint deposit;
7    uint8 rating;
8    uint ratingCount;
9    uint8 driverRating;
10 }
```

A rider's rating and rating count are kept between journeys, but otherwise, all remaining fields are empty when the rider is not part of a journey. As with a driver, the `addr` field is used to determine whether a rider is active. To determine whether the driver is locked into a journey, we look up the driver in the `drivers` mapping, using the given address. If their rider field is set to the address of this rider, then both users are locked into a rider together.

5.1.4 Double Linked List

As Solidity is still a relatively young language, some features which one would expect from a more mature language are missing. This includes the ability to return dynamic-length lists from a method, which posed an issue early in the development of Taxicoin. Although

[1] Rather, the mapping will return a Driver object with all zero values.

it's possible to return a single element at a time, this requires keeping a separate cursor for which element should be next. Therefore we instead use a novel implementation of a double-linked-list, which allows us to use the address of the current driver as the cursor to fetch the next or previous[2].

Modifying operations on the list are likely be performed on only a single element at a time, therefore to link to the next and previous item in the list is not much more of a cost. This provides the benefit of not having to scan forward to move backwards in the list, particularly useful for pagination, which will likely be needed when a large number of drivers are advertised and a user wishes to manually review potential drivers.

The contract uses a mapping which maps driver addresses to another map, which in turn maps a boolean to an address. The boolean represents whether we want to look up the next or previous element in the list - false is the previous, and true is the next. This then returns the address to use to look up the `Driver` object in the drivers mapping.

This list is easily interfaced with using the `getNextDriver` and `getPreviousDriver` methods, as defined in the contract interface. These features of the protocol were in fact partly designed this way due to the limitation as described above, that solidity does not support returning dynamic-length lists from methods.

5.1.5 Journeys

Building on the idea of rider and driver state, there is no concrete *journey* object in this Taxicoin implementation. Rather, the concept of a journey is inferred from the state of the participants.

When observing a rider, if the `driver` property is set, this indicates that the driver is either on a journey with that driver, or has created the journey and is waiting for the driver to accept. Whether the driver has accepted or not can be determined by observing the driver's `rider` property. If it is set the address of the rider, then the driver has accepted the journey and both are formally part of the same journey. If the driver's rider address is blank, they have not yet accepted the journey, and if it is the address of another rider, then they have effectively declined the journey with this rider and have chosen to accept another rider's journey.

At the end of a journey, both parties call the `Complete Journey` method, which will record the rating to be given to the other. This is stored in the rider/driver object for the user, and additionally acts as an indication as to whether the user has already called the method. This is then used by the method to determine whether to finalise the completion of the journey.

When both parties have completed the journey, the ratings given are applied (recalculate average rating, and increment rating count), deposits returned, and the fare paid to the driver. The state of both is then reset – their own addresses and those pointing to the other are set to zero, the ratings to be given to the other are set to zero, and the recorded values for the deposits are fare are set to zero.

5.1.6 Integrity Checks

As this implementation is heavily based on the state of objects within the contract, it is important to maintain data integrity. Coupled with the immutability of smart contracts, if we do not ensure that our contract state is kept correctly, it may be impossible to recover from a situation where the state is altered in some unexpected way.

A key feature of Solidity which allows us to check certain pre- and post-conditions is the use of `require();`. In the case that the contained statement evaluates to false, execution will halt and any changes made to the state of the contract instance are reverted.

[2]The ultimate implementation was based on [5].

For example, the protocol specification states that when a driver advertises, they must first be required to pay a deposit. Additionally, they may not already be on a journey as either a driver or rider. To check these post-conditions we can use the following Solidity code:

```
// check the driver has paid deposit
require(drivers[msg.sender].deposit >= driverDeposit || msg.value
    >= driverDeposit);

// must not be ActiveDriver or Rider
require(getUserType(msg.sender) != UserType.ActiveDriver);
require(getUserType(msg.sender) != UserType.Rider);
```

This is also useful for performing "safe math", where we want to protect against overflow or underflow. In both of these cases, integers in Solidity will simply wrap-around, which, particularly when dealing with currency as is the case much of the time in Solidity, can be catastrophically bad. Although it may not always be possible to recover from a situation where over- or underflow would occur, we can use `require` to check if the resulting value has been a result of such an occurrence, and revert the state of the contract instance.

5.2 Client Library

While the contract handles the state of Taxicoin, it can be unwieldy to interact with. It also only implements part of the Taxicoin protocol, not including any of the off-chain, peer-to-peer interactions.

The JavaScript client library is what really opens up the potential ecosystem of applications using the Taxicoin protocol. The aim was for it to be as simple to use and integrate into other people's projects as possible, and ideally should hide as much of the complexity of interacting with an Ethereum smart contract as possible.

The client library exposes a class, which when instantiated sets up all of the common requirements for using the Taxicoin protocol in most cases.

5.2.1 Web3 Interaction

The JavaScript library used for interacting with the Ethereum blockchain is called `Web3.js`[cite], and acts as a layer for sending commands to an Ethereum network node which computes transactions and relays them to the network.

This is coupled with the `truffle-contract` library, which abstracts the logic for calling methods of a smart contract, and provides each one as an asynchronous JavaScript method. These in turn interact with a Web3 library instance. The typical code[3]for calling a contract method is as follows:

```
const instance = await this.contract.deployed()
const account = await this.getAccount()
const tx = await instance.methodToCall([arguments, ]{from: account
    [, otherOptions})
```

`this.contract` is a `TruffleContract` instance which points to the Taxicoin contract code. Calling the `deployed` method on that returns a wrapper with a reference to a deployed instance of the contract on the current network. From this, we can then call the methods of the deployed contract.

The `getAccount` method fetches the Ethereum address of the user, as defined when instantiating a Taxicoin library instance. This is then required to be passed as an option with any contract method calls which modify state, as it is used to identify the user when signing the transaction.

[3]In ES6 async/await function syntax.

5.2.2 Peer to Peer Messages

The other key purpose of the client library is to facilitate the sending of peer-to-peer Taxicoin protocol messages. This is done with Whisper, one of the protocols part of the Ethereum network.

As Whisper is a fairly low-level protocol, sending messages is not entirely straightforward. The Web3 library handles much of the work, however it still requires various parameters, such as keys and the level of encryption it should use. Topics in Whisper also must be exactly 4 bytes in length, and the Web3 library expects this as a hexadecimal string, which is not something that would be straightforward for a user of the Taxicoin library.

Therefore, the library manages the keypair for use with Whisper, maps human-readable topic names to 4 byte hexadecimal string, converts JSON objects to strings for sending as the payload of a message, and handles the various other parameters used when sending Whisper messages.

For messages which are intended to be send after certain on-chain actions are taken, the library handles this, waiting for the transaction to be confirmed, and sending the appropriate message.

The library also registers filters for the various Taxicoin message topics, automatically polls for new messages sent to the user, and fires JavaScript events which the user of the library may listen for in their application.

5.3 Example Web Client

This section is yet to be written.

6 Verification and Validation

As Taxicoin is intended to be a ubiquitous, maintenance-free, and open protocol, it is extremely important that implementations correctly conform to the standard, and that they are without issues. If, for example, one widely used implementation featured an incorrectly implemented method with unexpected side-effects, it could impact on the whole network of Taxicoin clients and contracts.

6.1 Validation

In order to ensure this implementation of Taxicoin meets the protocol specification, various methods of validation have been used.

A series of functional integration tests validate the overall ability of the client library to interact with the smart contract instance, while conforming to the protocol specification.

Unit testing has also been performed on the smart contract itself to check the various pre- and post-conditions of each protocol method, as well as any additional internal methods not defined in the protocol.

Finally, a static analysis tool has been applied on the smart contract to check for common Solidity implementation errors.

6.1.1 Functional Tests

The functional integration tests use the *Ganache* [cite] tool to emulate a blockchain, on which we deploy the Taxicoin contract. The framework used for testing is Mocha [cite], along with the Chai [cite] assertion library. The tests are orchestrated by Karma [cite], which transpiles the JavaScript library and launches a Google Chrome browser instance in which to run the tests. Before each test, the state of the Taxicoin contract is reset, and a new Taxicoin library instance is constructed for use. These are then cleaned up in the after hook.

```
1  describe('driver accept journey', () => {
2    it('should throw an error if the provided address (rider) has
         not agreed to ride with the driver', async () => {
3      const riderAccount = await tcRider.getAccount()
4      await expect(tcDriver.driverAcceptJourney(riderAccount)).to.be
           .rejected
5    })
6
7    it('should set the driver\'s state to being on a journey', async
         () => {
8      const driverAccount = await tcDriver.getAccount()
9      const riderAccount = await tcRider.getAccount()
10     const fare = 100
11
12     await tcDriver.driverAdvertise(51.5074, 0.1278)
13     await tcRider.riderCreateJourney(driverAccount, fare)
14     await tcDriver.driverAcceptJourney(riderAccount)
15
16     const journey = await tcDriver.getJourney()
17     journey.should.not.be.null
18   })
19
20   it('should remove the driver from the list of advertised drivers
         ', async () => {
21     const driverAccount = await tcDriver.getAccount()
```

```
    const riderAccount = await tcRider.getAccount()
    const fare = 8000

    await tcDriver.driverAdvertise(51.5074, 0.1278)
    await tcRider.riderCreateJourney(driverAccount, fare)
    await tcDriver.driverAcceptJourney(riderAccount)

    const drivers = await tcRider.getDrivers()

    expect(drivers).to.be.empty
  })
})
```

Each test group is based around one of the methods or messages, as described in the protocol specification. With the methods, we can easily construct each individual test scenario based on the pre- and post-conditions.

These tests ensure the technical requirements, as defined in the protocol specification, are met.

6.1.2 Unit Tests

As mentioned previously, Solidity is still a fairly immature language, however related tooling does exist for running unit tests. The Truffle framework [cite] provides a Solidity-based testing tool, with standard assertions.

Unit tests have been used with Taxicoin to ensure that the contract (written in Solidity) performs the implemented functionality as expected. Each test case calls certain methods within the contract, given certain pre-conditions, and checks that the resulting state of the contract is as expected. An example test case is given below.

```
contract TestTaxicoin {
  ...
  function testDriverAdvertise() public {
    Taxicoin tc = Taxicoin(DeployedAddresses.Taxicoin());

    string memory lat = "1.23";
    string memory lon = "50.67";
    string memory pubKey = "<pub_key_goes_here>";

    uint driverDeposit = tc.driverDeposit();
    tc.driverAdvertise.value(driverDeposit)(lat, lon, pubKey);

    FetchedDriver memory dr = getDriver(tc, tx.origin);

    Assert.equal(dr.addr, tx.origin, "driver addr should equal
        address of sender");
    Assert.equal(dr.lat, lat, "driver lat should equal advertised
        lat");
    Assert.equal(dr.lon, lon, "driver lon should equal advertised
        lon");
    Assert.equal(dr.pubKey, pubKey, "driver pubKey should equal
        advertised pubKey");
    Assert.equal(dr.updated, block.timestamp, "driver updated
        should equal block timestamp");
    Assert.equal(dr.deposit, driverDeposit, "driver deposit should
         equal global driver deposit");
  }
  ...
}
```

6.1.3 Static Analysis

The Solidity static analysis tool *Manticore* has been used to check for common Solidity programmer faults. This is done through a series of input generation for "inputs that trigger unique code paths", as well as crash discovery of "inputs that crash programs via memory safety violations".

Additionally, it is able to detect common unexpected cases such as integer overflow. As an example, if a contract featured some method for withdrawing funds and the amount to withdraw was calculated based on the addition user input, the user may be able to cause an overflow, and cause the amount to be withdrawn to be incorrect, resulting in a loss of funds.

This static analysis is yet another method for ensuring that the Taxicoin protocol can act entirely autonomously, without fault, in what should be assumed as a hostile environment.

6.2 Verification

The above validation methods confirm that the technical specification for the project is met, however they do not provide a sense for whether the functional requirements of the project have been satisfied. The only way we can do this is to check each one individually.

Based on the below comparisons between functional requirements and implemented features, this implementation of the Taxicoin protocol satisfies all original core requirements, and also some of the additional requirements.

Drivers must be required to pay a deposit in order to advertise

When advertising, if a driver does not provide funds of an amount greater than or equal to the contract-defined driver deposit value, then the contract will revert and throw an error. Therefore, this requirement is met.

Riders must advertise jobs to drivers on an individual basis

An index of advertised drivers is available for all to see, including the public key of each, allowing a rider to send a `Job Proposal` message via Whisper. There is no public listing of proposed jobs, and therefore rider privacy is not compromised. This meets the requirement.

The fare must be determined by quotes from driver

When a driver receives a job proposal, they choose whether or not to accept it, and what fare they which to charge. This allows the driver to decide to adjust a fare based on the circumstances (e.g. congestion, distance). This requirement is met.

Riders must pay fares to a contract in advance

When a rider formally creates a journey on-chain, they must provided at least the stated fare (plus deposit, see below) with the transaction or the transaction will fail. Additionally, when a driver then formally accepts the journey on-chain, they provided a value representing what they are expecting the fare to be, and if the rider's provided fare does not match, the transaction will fail. This is a security measure to prevent riders from tricking drivers. Both of these factors satisfy the requirement.

Riders must provide an additional deposit before starting a journey

When formally creating a journey on-chain, a rider must provide funds greater than or equal to the stated fare (value passed as an argument), plus the contract-defined driver deposit, or the transaction will revert and throw an error. This meets the requirement.

Riders and drivers must both rate the other on completion of a journey

The complete journey method requires a rating between 1 and 255 to be passed as an argument, or the method will not execute. Until this is called, both parties are unable to retrieve their deposits, and the driver is not paid their fare. This mutual stake ensures that it is in both party's interests to complete the journey with ratings. Therefore this requirement is met.

When a journey is completed, deposits should be returned to the respective parties, and the fare paid to the driver

On the completion of a journey (once both parties have called the complete journey method), the deposits paid by either party are returned and the fare is paid to the driver. This satisfies the requirement.

Prospective drivers and riders should be able to informally communicate before forming a contract (additional requirement)

This additional requirement was not implemented as it was not deemed essential, but would be in a future protocol revision.

Dispute resolution should be built into the system (additional requirement)

If the driver and rider have a disagreement during the journey, they are able to propose a new fare to the other party, initiating a negotiation over the new fare, performed peer-to-peer via Whisper with `Propose Fare Alteration` messages. When an agreement is met (both parties propose the same fare), the driver formally proposed the new fare on-chain, and the rider formally accepts it. If the new fare is higher, the rider must provide the difference with the transaction, otherwise the difference is returned to the rider.

Additionally, if the rider believes the driver has not provided the promised service, they may wish to entirely cancel the journey and pay have their fare returned. This can be achieved by setting a fare of zero. The journey may then be completed. This thoroughly meets the requirement.

7 Evalutation

This final sections acts as an evaluation of the success of the Taxicoin project, both as a protocol and an implementation.

7.1 Completeness of Requirements

"How do you know that your set of requirements is complete and correct, with respect to the objective of the project?"

7.2 Impact

The premise of this entire project was that traditional taxi travel can be an unfair experience, and that riders and drivers are often at the mercy of large corporations, especially with more modern app-based services.

As far as taking the power away from a central authority goes, Taxicoin has achieved this by effectively utilising an Ethereum network based protocol, which doesn't place the trust of the service in any one participant's hands. In theory it should be nonsensical for one party to attempt to cheat the other, as they will lose their deposit, although the deposit must be significant enough for this mechanic to function correctly.

If a person wished to develop a Taxi app, their desired features should conceivably be possible to implement within the scope of a Taxicoin implementation. And in the case that this is not true, as the protocol specification is open source, they may submit proposed changes to it, or write an extension to it.

7.2.1 Negotiation Standard

Taxicoin uses its own novel method of negotiation, where two parties first agree on a price for a service off-chain (through Whisper), and then make a formal agreement on chain. This is a method which may potentially be used with a myriad of other applications.

As such, it would be sensible for a generic standard to be developed for this purpose. This would allow for the development of other contracts which can participate in negotiations which follow the standard, in a similar way to, for example, interfaces in programming languages which support polymorphism.

7.2.2 User Value

In context of aims set out at the start of the report.

7.3 Future Development

There is certainly room for improvement in both the protocol, and the implementation presented in this document.

7.3.1 Additional Features

There are several mentions throughout this report of potential improvements to the protocol. One such is the inclusion of a message type for informal "chat" between rider and driver before and during a journey. Although the protocol functions without this, it would be useful to be able to communicate special requirements, for example.

The Taxicoin JavaScript library makes heavy use of the *Web3.js* library and signing transactions on a local Ethereum node, which functions adequately, but can be inflexible. If the user does not have access to their own local node, they may wish to use a third

party node, such as the Infura service [cite]. In order for this to be secure, a user's signing key must not be transmitted to the third party. Therefore, a reasonable improvement to allow this would be to implement the signing of transactions locally (from JavaScript). This can be achieved either with Web3, or via an alternative such as *Ethers.js*.

Additionally, due to the interaction-based nature of the protocol, there are likely to be some "rough edges" around the protocol, for which certain aspects will have to be adjusted. These are most likely to be focused around unforeseen exceptional circumstances.

As the protocol has been designed around an open ecosystem of independent implementations, it is hoped that such implementations will begin to be written. The core of these will likely be interaction libraries for various languages (other than JavaScript), and indeed one such library is already under development for the Go language [cite], based on the protocol specification in this document.

7.3.2 Licensing Considerations

At present, taxi drivers are required to hold a license in order to operate in many areas. In it's current state, the Taxicoin protocol could potentially support checking whether a driver is allowed to operate. However, this would require a blockchain-based *oracle*, maintained by the body which issues taxi licenses. This would allow other smart contracts on the network to view the records of drivers (the issuing body would also have to publish an identifying address of the driver).

From a technical point of view, this is a very simple step to take. However, it would likely be blocked by local government bureaucracy, therefore it is not feasible at current, and it was not included in the project for this reason.

Identity on blockchain-based networks is still a relatively new topic. Many research projects are targeting the issue, therefore it is hoped that an agreed-upon solution and standard will be available in the near future.

7.3.3 Client Algorithms

The specification states that certain decisions are intended to be made in a semi-autonomous way. This includes the fare negotiation stage when a rider proposes a job to a set of drivers.

In the naive example client presented previously, the rider and driver fare negotiation is a manual process, however in reality it should function somewhat like the following: the driver's client determines an initial fare based on distance of journey and any other parameters. The rider's client then either approves or rejects, based on its threshold of what it believes a fair price to be. In the case of a rejection, the driver's client would incrementally decrease the fare quote until a threshold set by the driver is met, at which point the driver's client would reject the job.

There are many factors to be considered when developing this algorithm, thus why it was out of the scope of this project.

8 Conclusion

Todo.

References

[1] Viktor Trón. *What is swarm?* 2015. URL: https://github.com/ethersphere/swarm/blob/ac8b54726551d7a590f85d6d377dbcac3ae26794/README.md (visited on 03/16/2018).

[2] URL: https://github.com/ethereum/wiki/wiki/Whisper (visited on 03/16/2018).

[3] URL: http://ethdocs.org/en/latest/introduction/what-is-ethereum.html (visited on 03/16/2018).

[4] URL: https://bitcoin.org/bitcoin.pdf (visited on 03/16/2018).

[5] URL: https://ethereum.stackexchange.com/a/15341 (visited on 04/20/2018).

Appendix

Project Definition

Subject

When the Internet was in its infancy, if you wanted to use it for a specific application, you might have written a protocol. That way, anybody who wanted use the Internet for that purpose would have a common way of doing it - and if a new person came along and wanted to join in, they could just write their software to conform with the standard.

In the past 15-or-so years however, the landscape has changed. Companies now favour their own proprietary systems, where they can have complete control, and ultimately gain the most profits. Specifically companies such as Uber have taken an industry which was once fairly well distributed, and put the control in their own hands - they decide who can be a driver, they manage the fares, and how much they pay their drivers.

But recent developments with distributed networks threaten to disrupt this comfy business model. Technologies such as Ethereum allow "trustless" applications, where activities of a single node are verified by the entire network. It's an area which is yet to be explored to its full potential, but all the features needed to be able to implement feature-rich apps are there. The logic of applications running on such a system has to be rethought, but with Uber as an example, there would be no central authority to take a cut of profits. The entire system would be self-regulating.

Deliverable

A ride-sharing webapp accessible with an Ethereum network-enabled browser, designed in such a way that no single entity has control over the running of the system. Drivers will be able to advertise their location (published to blockchain), and riders will be able to send job proposals (containing pickup and drop-off locations) to these drivers on a peer-to-peer basis. This protects the privacy of the rider by ensuring that only the chosen drivers are able to see the rider's location. When a driver initially advertises their location, they are required to provide a deposit to the network, which will be returned in completion of a trip. This gives the driver a stake in wanting to complete a journey, and should reduce spam on the network.

Drivers are then able to issue a response to a proposal, by either rejecting or quoting a price for the journey. This allows drivers to choose which journeys they take, and prevents drivers from having to travel a long distance to a pickup location, compared with if the allocation was done randomly. Should the rider choose to accept the quote, then both rider and driver form an agreement via a smart contract on the Ethereum network. This includes the passenger offering up the cost of the journey, plus an additional deposit equal to the amount the driver provided previously.

At this point, the fare for the journey, a deposit from the rider, and a deposit from the driver are all held by a smart contract. This acts as an incentive for the driver and rider to successfully complete the physical journey. When this is done, and both parties are in agreement that it is completed, then the deposits can be returned and the fare paid to the driver.

All monetary transactions will be executed with cryptocurrency on the Ethereum network, so as to minimise fees and prevent the transaction from being intercepted by a third party.

As the vast majority of interactions between riders and drivers will be based on no existing knowledge of the other party, a reputation system will be used to form a layer of trust. Based on previous journeys, and the ratings given to both rider and driver on completion of each, future riders will be able to make informed decisions about which

drivers to send job proposals. And in the same fashion, drivers will be able to decide which riders' proposals to accept.

Originality

Although ride-sharing apps aren't a new thing, nearly all existing solutions are controlled by a central authority who take a cut of the profits. This means users are at the mercy of the company when it comes to fares, and drivers must be approved, potentially opening the way for discrimination.

This project eliminates these problems by taking control away from any one part of the system. All transactions take place in a peer to peer nature, with the network being the only intermediary. This ensures that the two parties involved have full control over the process, whilst at the same time preventing one from cheating the other.

Timetable

The following proposed timetable will be used to track progress over the course of the project. The work is broken down into fortnightly blocks. Through the entirety, a project diary will be kept to keep track of key decisions. This is to be used as the basis for much of the final report.

Date	Planned Activity
02/10/17	Begin writing a formal project definition. Decide on project objectives, and have an idea of what features will be included. Which features would the system not work without.
16/10/17	Finish project definition. Begin mapping out interactions of users with the system and other users as a diagram. Create protocol documentation - similar to RFC. This is to be used to test against. **Project definition due 20th October**
30/10/17	Start implementing said protocol, with aim of creating fully function implementation (not including user interface). Test against RFC-style document.
13/11/17	Finish initial protocol implementation.
27/11/17	Develop testing suite for protocol implementation.
11/12/17	Fix any issues with implementation, and complete testing. Begin TP1 progress report.
25/12/17	Continue TP1 progress report.
08/01/18	Exams scheduled in this period, therefore expecing a slow down in project work. **TP1 progress report due 19th January.**
22/01/18	Continuation of development based on progress report. Begin writing of final report.
05/02/18	Research into how to develop the user interface. Review of existing mobile Ethereum clients.
19/02/18	Development of user interface.
05/03/18	Addition of identified "stretch" features.
19/03/18	Finalising development and report writing.
02/04/18	Report writing.
16/04/18	Finalising report and considering how to present the project during demos. **Final submission due 27th April.**

Teaching Period 1 Progress Report

This is the teaching period 1 progress review for my final year project, referred to here on after as *Taxicoin*.

Current Progress

As of January 2018, significant progress has been made on the implementation. From a technical point of view, the core "must have" features are complete.

As a rider, the user is able to advertise their job to drivers on an individual basis. The intention is that eventually the advertising will be done automatically, to all available drivers which meet some criteria, e.g. minimum rating or maximum proximity from rider.

When accepting a journey with a specific driver, the rider must pay the fare for the journey up-front, as well as an additional deposit which ensures the rider has a stake in completing a journey without dispute.

At the end of a journey, the rider is able to rate the driver. The rating acts as the only form of reputation, and is currently a simple average of all ratings. Each rating is an integer between 1 and 5.

Drivers are able to advertise their location publicly as an indication that they are active and accepting job proposals. However, to do so, drivers must provide a deposit.

In the event that a driver receives a job proposal, they have the option to respond with a quote for the fare of the journey.

Significant Achievements

- Technical contract implementation is now in a working state.
- User interface with map, location search, and user geolocation is in a working state.
- Spoke at BrumJS November 2017 meetup on the subject of the Ethereum platform and its uses. Afterwards gathered informal feedback about the concept of Taxicoin.

Next Steps

In terms of the implementation itself, the user interface needs tidying up significantly. At the moment, the "user flow" is somewhat lacking, and not as fluid as it should be. This is the first priority.

The reputation is currently very simplified from what I had initially planned. I would like to develop it further, as it is an integral part of the application. I've recently read into how other Ethereum-based decentralised applications are managing their reputation systems, and will hopefully apply some of the ideas in Taxicoin.

I plan to write an "Introduction to Ethereum" section of the report, with a comprehensive explanation of how the platform functions, and why I have chosen to develop Taxicoin with it.

As discussed with Peter Lewis, a crucial part in proving the successful implementation of a complete protocol for Taxicoin will be to develop a comprehensive suite of tests. These will primarily test the functionality of the contract parts of the application, as this is where the protocol layer is implemented.

Hurdles

The Ethereum platform is still rapidly evolving. Indeed, even from when I began research into how I would develop this project, protocol specifications have been amended. As a

result, I am having to keep an eye on developments with Ethereum while developing the application.

Traditional databases for storing data do not translate well to blockchain-based systems. Therefore I will need to research distributed databases, particularly for a more advanced reputation system. This is unexplored territory for me, so I am unsure what to expect.

Project Diary

3rd October 2017 talked about the fact that this project is relevant to the interests of the ALICE group. It is effectively a self-governing application. Was decided that the focus should be on compiling a list of "must have" features and then implementing them.

16th October 2017 was suggested to write a RFC-style protocol specification, to be used later to test against to determine if the implementation is correct.

13th November 2017 no huge amount of progress was reported due to other commitments. We revisited the idea of producing an RFC-style document, focusing on the IMAP protocol as an example.

4th December 2017 we discussed that including a network architecture diagram in the report would be a good idea of explaining how various parts of the project communicate with each other (e.g. front end talks to contract, different instances of front end talk to each other). At this point, a working implementation had been completed, therefore we began talking about how to write tests. It was decided that the contract should be tested directly with unit tests, and potentially integration testing performed on the Javascript abstraction layer and contract. We discussed that it would be good to get to the point where the application could be security audited.

Ethics form for student projects

SEAS group: Computer Science

Project title: Taxicoin

Supervisor name and email: Peter Lewis <p.lewis@aston.ac.uk>

Ethics questions

Please answer Yes or No to each of the following four questions:

1 - Does the project involve participants selected because of their links with the NHS/clinical practice or because of their professional roles within the NHS/clinical practice, or does the research take place within the NHS/clinical practice, or involve the use of video footage or other materials concerning patients involved in any kind of clinical practice? ***No***

2 - Does the project involve any i) clinical procedures or ii) physical intervention or iii) penetration of the participant's body or iv) prescription of compounds additional to normal diet or other dietary manipulation/supplementation or v) collection of bodily secretions or vi) involve human tissue which comes within the Human Tissue Act? (eg surgical operations; taking body samples including blood and DNA; exposure to ionizing or other radiation; exposure to sound light or radio waves; psychophysiological procedures such as fMRI, MEG, TMS, EEG, ECG, exercise and stress procedures; administration of any chemical substances)? ***No***

3 - Having reflected upon the ethical implications of the project and/or its potential findings, do you believe that that the research could be a matter of public controversy or have a negative impact on the reputation/standing of Aston University? ***No***

4 - Does the project involve interaction with or the observation of human beings, either directly or remotely (eg via CCTV or internet), including surveys, questionnaires, interviews, blogs, etc? *Answer "no" if you are only asking adults to rate or review a product that has no upsetting or controversial content, you are not requesting any personal information, and the adults are Aston employees, students, or your own friends.* ***No***

Student's signature: Scott Street

Supervisor's signature: _____

Please submit this form as part of your Term 1 Progress Report. If any of the answers are "yes", you will need to complete an online application for ethics approval, which can be found at https://www.ethics.aston.ac.uk *. You can log in with your usual Aston user name and password.*