Ca' Foscari Unversity of Venice

Science and Technology Department

Master's Degree Programme in Computer Science

Decentralized Carpooling with Algorand Blockchain

Supervisor:
Marin Andrea
Associate Professor

Graduating:
Baratella Matteo
Student ID 856097

Academic Year 2021/2022

Abstract

Nowadays, blockchains are increasingly used for the development of applications. Blockchain is, generally, a distributed ledger that is persistent and accessible to anyone. One of the big advantages of the utilization of this technology is the decentralized nature of the software, and we will take advantage of this in order to develop a car pooling management application that, in contrast to the widespread ones, does not rely on any centralized authority. In this project, we will see the logic behind the implementation of a car-pooling application with Algorand Blockchain as a case of study for a decentralized development with Algorand Smart Contracts. The main features of a blockchain are taken into review and analyzed in order to better understand how to properly develop a dApp with Smart Contracts. The application aims to make a demonstration on the pros and cons that a Blockchain can have, and which is the approach for developing in a decentralized method. Particular attention will be devoted to the Algorand project, an emerging and promising technology based on proof-of-stake.

Contents

1 Introduction

Carpooling, sometimes also improperly called carsharing, refers to a method of transportation which consists in a group of individuals sharing private cars with the primary goals of reducing transportation costs. Usually, one or more of the members make their vehicle available, potentially alternating the usage, while the others provide enough funds to cover a part of the drivers' costs.

Carpooling is environmentally friendly and sustainable, since it allows for a reduction in the number of vehicles in circulation, which has a positive impact on pollution, traffic congestion, and infrastructure needs. Also carpooling is an efficient way to reduce the costs for a trip, which will be splitted by the trip party in terms of tolls and fuel.

Normally, carpooling business takes place via a centralized application that could be either a mobile application or a web app accessible with a browser. This will require the user to make a registration or give some personal information to the business society that owns the application in question before accessing the service.

Customers and car owners may have trust issues, as well as certain security concerns, when using carpooling. Moreover, payments should be trusted and managed by a central authority. Finally, it is debatable that a company makes profit exploiting the resources of private citizens. Implementing an application based on a blockchain is one method to provide more transparency to a carpooling service while also making payments secure and convenient, as well as providing more scalability and fault tolerance.
In order to do this, we will introduce the concept of **blockchain**: a distributed ledger where nodes of the network communicate and validate new blocks to each other using a decentralized protocol. More specifically we are interested in **Algorand blockchain** and the development of a **dApp** with Algorand **Smart Contracts**.

The most famous example of carpooling systems is:

- *Blablacar*: a French carpooling service founded in 2004 by Frédéric Mazzella and available in 22 countries.

But there exists also similar ridesharing services such as:

- *Uber*: is an American mobility as a service provider which includes ridesharing services. Uber does not own any car, instead it provides a matching service between passengers and vehicle drivers.
- *Lyft*: an American company offering ride-hailing services, it works similar to Uber and it is the second largest ridesharing company in the United States.

Carpooling **has not to be confused** with car sharing as a form of short-term rental. In this kind of car sharing the cars are owned by a company and made available to the customers, who can reserve and use them for the time needed, after which they are released and made free to other users.

The contributions of this thesis are the following:

- Review of blockchains and smart contracts with particular attention to Algurand blockchain;
- Design of a purely distributed carpooling system based on blockchains and smart contracts;
- Development of a prototype of the backend and frontend of the application;
- Review and design of the development environment required to develop with a blockchain.
- Review of the security of a smart contracts based application with Algorand blockchain.

2 Blockchain Technology

Introduction

A blockchain is a distributed database that is shared among various nodes of a computer network in order to store persistent data through a consensus protocol. A blockchain collects data organized in blocks that carry information, the chain is composed by the history of all the transactions and works essentially as a **distributed ledger** that is decentralized and completely open to anyone. Each transaction is verified and approved by a consensus protocol, and once some data has been recorded inside the blockchain, it becomes nearly impossible to change it.

Each block generally contains:

- **Data**
- **Hash**
- **Hash of previous block**

Data stored inside the blockchain depends on the type of the blockchain. The hash is unique and identifies the block and all its content, once a block is created its hash is being calculated. Changing something inside the block will cause the hash to change and invalidate the whole chain.

The Blockchain concept was first described by a group of researchers in 1991. However it went mostly unused until it was adapted by Sakoshi Nakamoto in 2009 to create the digital currency of Bitcoin, as described in the original paper "Bitcoin: A Peer-to-Peer Electronic Cash System" [25].

Blockchains are mostly being used to perform coin-based transactions, but there are a lot of other useful applications nowadays, like smart contracts, that allow the development of decentralized applications through the blockchain.

2.1 Use Cases

Blockchain nowadays can be used hypothetically for many different purposes, such as:

Banking and Finance Financial institutions are only available during regular business hours, which are normally five days a week. Consumers could see their transactions executed in as little as 10 minutes by integrating blockchain into banks, which is the time it takes to add a block to the blockchain, regardless of holidays or the time of day or week.

Healthcare Healthcare providers could store their patients' medical records in a safe manner using a blockchain. When a medical record is created and signed, it can be made persistent on the blockchain, giving patients confirmation and assurance that the record cannot be altered. Moreover to guarantee privacy, a cryptographic algorithm could be used to secure information.

Smart Contracts A smart contract is a program that can be built and run on a blockchain to validate or negotiate a contract agreement. In order to work with Smart Contracts, users have to agree to a set of requirements. The terms of the agreement are automatically carried out once those compliances are met.

Supply Chains One of the most employed blockchain utilizations are the supply chains, suppliers can track the sources of the materials they buy.

Voting System Another popular usage of a blockchain could be a decentralized voting system. The usage of the blockchain for a Voting system has the ability to prevent election fraud and increase voter turnout.

2.2 Distributed Systems

A blockchain is based on distributed systems technology design in order to store immutable data validated in a peer-to-peer network. It uses a consensus protocol to achieve a common outcome on all the participants of the network.

A distributed system is composed of a set of autonomous nodes on a network that communicate and coordinate in order to achieve the same goal. These nodes can be separated by any distance and they interact by exchanging information through the network.
Some of the advantages of distributed systems design are:

- **Resource Sharing**: the capability of sharing of hardware and software resources.
- **Heterogeneity**: that means that it can be composed of different components, hardware or software.
- **Reliability**: the system is fault-tolerant up to some crash.
- **Extendibility**: refers to the possibility to include another node in the system.
- **Scalability**: refers to the possibility of including a new node of the same type. Scalability can be horizontal or vertical.
- **Performance**: the ability to offer results in an efficient way and optimum values of throughput and response time.
- **Transparency**: concealment of the separation of components so that the system appears to the customer as a single system.

2.3 Digital Signature

In blockchain we need to resolve the problem of trust of both the parties that take part in the transaction, to solve this issue, we introduce the concept of **digital signature**.
Digital signatures are used to detect **unauthorized modifications** to data and to authenticate the identity of the signer. The recipient of a signed message can use a digital signature as evidence to demonstrate to a third party that the signature was generated by the claimed signer.

The properties we would like to have are:

- **Authenticity**: the origin of the message is correctly identified
- **Non-repudiation**: the sender cannot deny the transmission of a message
- **Integrity**: the information can be modified only by the authorized entities

In order to obtain these properties a combination of **hash functions** and **public key cryptography** are used.

1. First, the message digest is created using a hash function.
2. The signature is created by encrypting the message digest using the signer's private key.
3. Anyone can use the signer's public key and the same hash function to verify the received signature.

A digital signature scheme consists of two functions:

- $SigSK(x)$ generates the signature of x using a private key SK;
- $VerPK(x, y)$ checks the validity of signature y on message x using a public key PK. Returns true if the signature is valid and false otherwise.

We require that these two functions are easy to compute knowing the keys. Moreover, it must hold $VerPK(x, SigSK(x))$.

In order to perform a blockchain-based transaction an user needs a **private and a public key**. Every transaction of the blockchain is **signed** by the sender's electronic signature using their private key and the public key of the next owner. The receiver can **verify the signatures** to prove the chain of ownership.

2.3.1 ECDSA: applied to blockchain

Elliptic Curve Digital Signature Algorithm is a simulation of Digital Signature Algorithm using Elliptic Curve Cryptography. ECC is based on **Elliptic Curves** that is a non-singular cubic equation defined by the solutions of the equation [32]:

$$E = \{(x, y)|y^2 = x^3 + ax + b\} \cup \{O\}$$

with

- O is a point at infinity.
- $a, b \in K$, where elements a and b are to the field K, together with a point at infinity.
- $4a^3 + 27b^2 \neq 0$

ECC security is based on the discrete logarithm problem (ECDLP) of the elliptic curve.

Currently Bitcoin uses secp256k1[34] with the ECDSA algorithm. secp256k1 refers to the parameters of the elliptic curve used in Bitcoin's public-key cryptography, and is defined in **Standards for Efficient Cryptography**.

ECC offers some advantages over other cryptographic systems, the significant one is that the key size of ECC is much smaller than RSA in order to achieve the same level of security [29].

This lead also to provide easier hardware realizations, compact implementations for cryptographic operations requiring smaller chips and is more suitable for machines having low computing power.

Table 1: Comparison between ECC and RSA key sizes

ECC	RSA
256 bit	3072 bit
384 bit	7680 bit
512 bit	15360 bit

2.4 Transactions

In order to see how a blockchain transaction works, we will take the Bitcoin blockchain model as a case of study. A problem of the blockchain is to guarantee the transactions to come in order in which they are generated, in fact, being a peer-to-peer network, once a node makes a transaction, it will broadcast it onto the whole network. Also this will lead to another big problem: the payee can not verify that one of the owners did not **double-submit** the transaction. For example, in a case of digital currency blockchain, we cannot verify that an user did not double-spend the coin.

We need a system in order to make sure that the entire network can agree regarding the order of transactions. A solution would be to introduce a trusted central authority that checks every transaction in order to make sure that is legit. Since we want to build a decentralized network, Introducing a central authority is not a good idea. Instead, to solve this, transactions must be publicly announced, and we need a mechanism for the nodes to agree on the same history of the order in which they were received.

Bitcoin solved this problem introducing the **Proof-of-work** method in order to validate transactions with a consensus protocol [27].

2.4.1 Proof-of-Work

All the transactions stored into a block are considered to have happened at the same time. Also these blocks are linked together in a linear way since every block contains the hash of the previous block, and that makes sure that a correct chronological history is maintained.

We introduce a **timestamp** that proves that the data must have existed at the time, in order to get into the hash. To implement a distributed timestamp server on a peer-to-peer basis, we use the proof-of-work system.

Proof-of-work is used for the next block generation election, and it works by having all the nodes of the network solving a **cryptographic puzzle**. This puzzle is solved by "**miners**", and the first one able to find the solution gets the miner rewards and can generate the next block of the chain. This miner **reward** is needed in order to incentivize miners to use their resources

to solve the puzzle and make sure to pay them back the CPU time and electricity that they have expended. The proof-of-work puzzle involves scanning for a value that when hashed, such as with SHA-256, the hash begins with a certain number of zero bits.

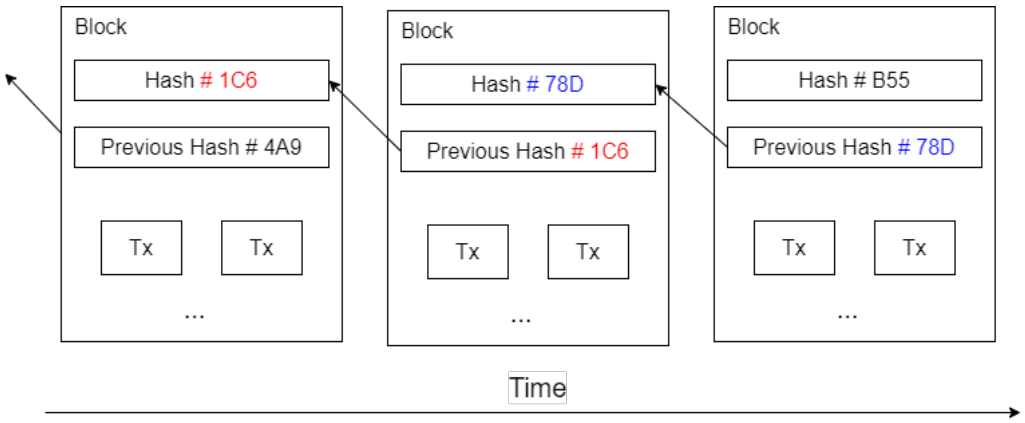


Figure 1: Blockchain Transactions

There is a probability that two users will generate a valid solution for a block simultaneously when using PoW and because of this different groups of users may have different candidates for the next block, and the blockchain **forks** into two. A fork can last for a long time, and its branches can even be extended by adding more blocks, but eventually only the longest branch will be kept, discarding all the other branches and their blocks, and the transactions on the dead branches are classified as invalid;

Security of PoW Generally, it is wasteful for a malicious user to spend resources to create malicious blocks, since the whole network will verify them and a block is validated only if the majority consensus of the miners is reached. However, the miners are not under direct control of an administrator, since everyone can join the network.
What then, if some malicious users can afford to reach **51% of the mining power** of the whole network?
In order to solve this problem, miners get proportional reward to the computational power, so to be discouraged from making fraudulent blocks, thus they are incentivated to use their power legitimately, since in this way they will earn more benefits.

2.4.2 Proof-of-Stake

Proof-of-work is the first consensus algorithm developed for Bitcoin Blockchain, but nowadays there are a lot of different existing protocols, such as Proof-of-Stake, Proof-of-Authority, Proof-of-Space and yet many others are in development. We will focus now on the second widely adopted algorithm: **Proof-of-Stake**.

The basic idea is that letting everyone compete against each other with mining is wasteful, so instead proof of stake uses an election process in which one node is randomly chosen to validate the next block.

Proof-of-stake confers decision power to minters that actually have a stake in the system and unlike the PoW in which everyone can become a miner, not everyone can join the network in a PoS system. Ownership of a currency or having a deposit in the network allows the nodes to participate in the minting process i.e. to validate transactions and create blocks.

No computational power is required to solve cryptographic puzzles like in the PoW scheme and since that, there are no rewards under the form of a money creation: validators collect fees from users and are paid from them as an usual intermediary.

Proof-of-work instead of miners is based on **validators**, that are not chosen completely randomly: to become a validator a node has to deposit a certain amount of coins into the network as **stake**. The size of the stake determines the chances of a validator to be chosen to forge the next block.

Security of PoS As for the PoW system, also the PoS system has the problem of the 51In this case, we have no miner rewards in order to discourage malicious attacks, otherwise, validators will lose a part of their stake if they approve fraudulent transactions. If this sanction is sufficiently high, an optimal control rate on the fraudulent transactions can prevent any attack of the system.

2.4.3 Transaction workflow

Finally, the steps in order to perform a transaction are the following:

- First, to make a transaction, you need a digital signature in order to verify the owner of the chain of ownership
- New transactions are broadcasted on the network, and all nodes receive the new transactions;
- A leader is selected with a consensus algorithm in order to decide who will generate the next block of the chain;
- The next block is generated and is broadcasted into the network;
- All the nodes validate the block, and if consensus is reached, the block is finally added to the chain.

2.5 Disk Storage

Once the latest transaction is buried under enough blocks, the spent transactions before it can be discarded to save disk space. To facilitate this without breaking the block's hash, transactions are hashed in a **Merkle Tree**, with only the root included in the block's hash. Old blocks can then be compacted by stubbing off branches of the tree. The interior hashes do not need to be stored.

In cryptography and computer science, Merkle tree is a tree in which every leaf node is labeled with the cryptographic hash of a data block, and every non-leaf node is labeled with the cryptographic hash of the labels of its child nodes.

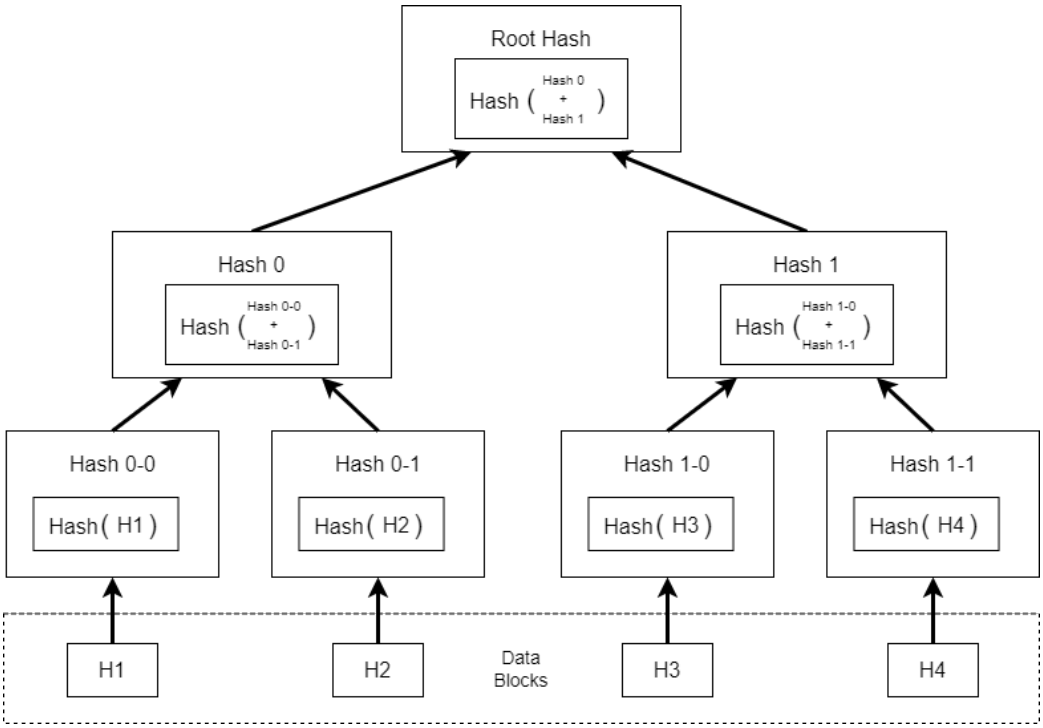


Figure 2: Merkle Tree

Conclusions

In this chapter we have acquired the basic knowledge of a blockchain structure and behavior, in particular with the transactions management and the most used consensus protocols.

3 Algorand

Introduction

Algorand is a blockchain developed in 2017 by Silvio Micali, a professor at MIT, that aims to solve the 3 main properties of a blockchain: **scalability**, **decentralization** and **security** (also known as the **blockchain trilemma**)[22]. To guarantee transparency, Algorand code for the core protocol is open source, it is open to everyone to review and contribute to. With a block size of 5,000 transactions and a block generation time of 4.5 seconds, Algorand throughput results in around 1,000 transactions per second [23].

3.1 Algo token

Each blockchain has its own native currency, which is used to perform some actions into the blockchain, as well as paying fees for transactions or to join the consensus protocol in order to validate new blocks of the chain. Algorand's native currency is called "**the Algo**". Algorand is also a permissionless, open platform and anyone who owns Algos can join the consensus protocol anywhere in the world.

3.2 Fees

Usually on a lot of blockchains, such as Etereum, when submitting transactions to the blockchain, users pay a fee. This fee is calculated in units called "gas" and is proportional to the transaction's size and complexity. These fees are required to incentivize the miners and ensure the integrity of the blockchain, and also the fee structure helps balance transaction processing supply and demand.

On Algorand, **there is no concept such as a gas fee**. When network traffic is significant and blocks are constantly full, a user can decide to increase a fee to prioritize his admission into a block. The minimum fee for a transaction is or 0.001 Algos.

3.3 Pure Proof of Stake

As mentioned before, the main goal of Algorand is to solve the blockchain trilemma: security, scalability, and decentralization. In order to solve this, Algorand uses a new version of the PoS protocol: **Pure Proof of Stake (PPoS)** consensus protocol. The PPoS strategy of Algorand relates the security of the entire economy to the honesty of the majority of the economy, instead of a reduced subset.
Also, forking on Algorand is impossible, since it is a PPoS and uses a voting algorithm to generate blocks.

Algorand uses a decentralized Byzantine Agreement protocol [36] based on pure proof of stake (Pure POS), this method for generating nodes consists of two stages: **proposal** and **vote**. Users are randomly and secretly selected to propose blocks and vote on block proposals. By staking $ALGO and generating a participation key, any member can become a participation node and join the process, but also an user must be online in order to participate in the consensus protocol. The coordination of these nodes is carried out by **Relay Nodes** which ensure communication within the network but cannot propose or vote on other nodes in the network.

The algorithm works with two phases: the **proposal phase** and the **voting phase**. In the block proposal phase, based on the quantity of tokens staked, the Verifiable Random Function mechanism selects a **block proposer** at random to propose a new block. Next, in the voting phase, a small group for the voting committee is selected randomly by the VRF, and will check the validity of the previous proposal.

3.3.1 Verifiable Random Function

For its consensus mechanism, Algoand uses the **Verifiable Random Function**. This function takes a private key and a value and generates a pseudo-random output, complete with a proof that anybody can verify. This VRF output is used to sample from a binomial distribution to simulate a call for each algo in a user's account.

3.3.2 Proposal Phase

During the proposal phase all the nodes loops over all its online accounts with a valid participation key, that is like a ticket lottery, and then runs the VRF to check if the account is selected through this lottery. If the VRF selects an account, the node propagates the proposed block along with the VRF output, to prove the validity of the proposer [1].

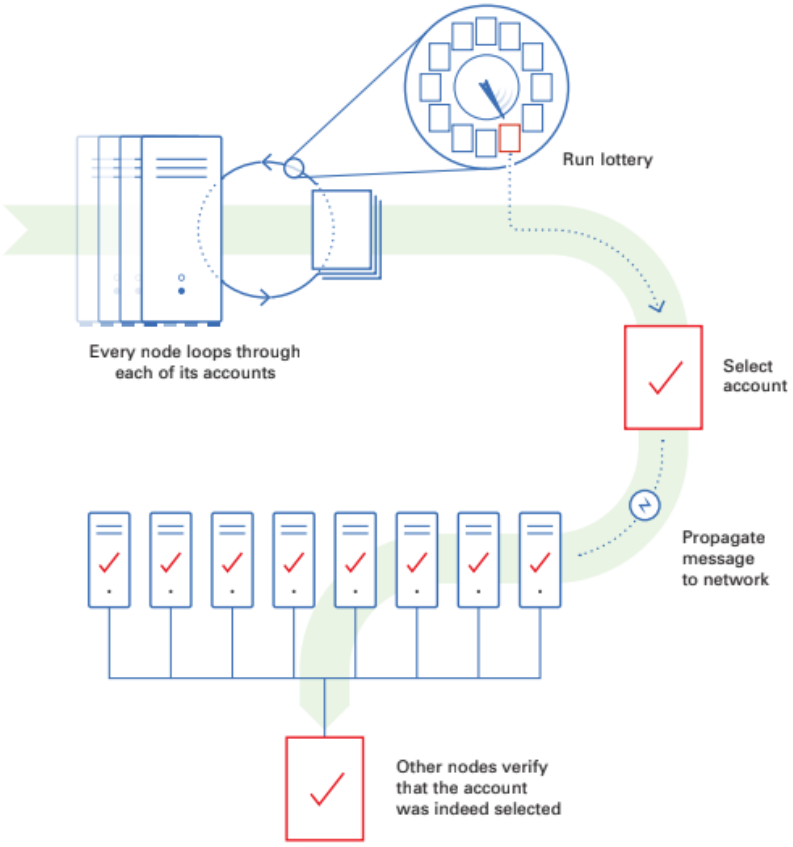


Figure 3: Block Proposal

3.3.3 Voting Phase

In the voting phase cryptographic techniques are used to verify that the vote is valid, that nobody can cheat, and that the system is completely safe. The voting phase is divided into two different stages:

- soft vote
- certify vote

Soft Vote Part 1 Multiple nodes will be selected as proposals, and each of them will go through a round of communication to verify and prove to the others that the participation key is valid using the VRF proof. Now to select a leader the lowest block protocol is used: for each validated winner's VRF proof the node will check the hash in order to discover which is the lowest, and then broadcast only the proposal with the lowest VRF hash.

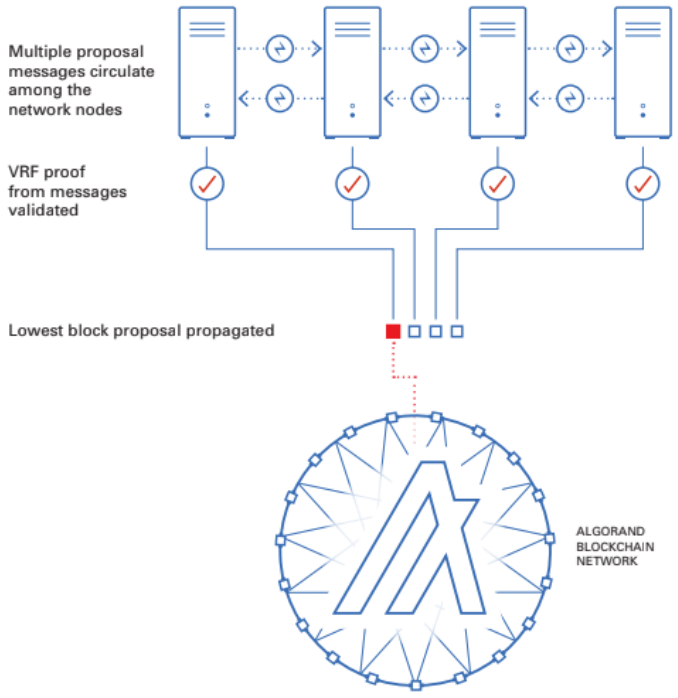


Figure 4: Soft Vote Part 1

Soft Vote Part 2 Now all the nodes will run the "lottery" with the VRF again in order to elect a smaller group for the voting committee. As before, the VRF proof of each node is verified by all the other nodes. The voting committee, once selected, will communicate to each other in order to share their votes and reach a consensus. Once a consensus is reached the process moves on to the certify vote.

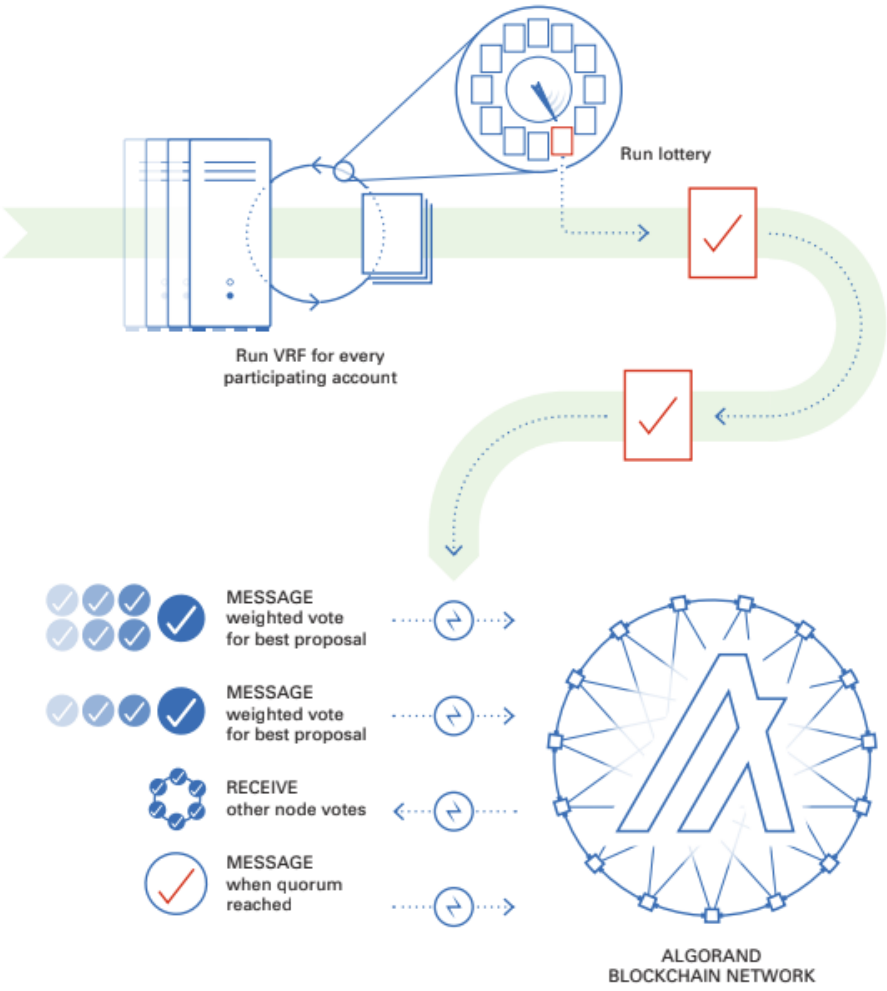


Figure 5: Soft Vote Part 2

Certify vote In this step, another voting committee is selected in the same way as the soft vote, and this new committee is in charge to check the voted proposal against some problems like overspending or double-spending. If no problems occur, the nodes will make another vote to certify the block proposal in a way similar to the soft vote.
If a consensus is reached, the block is certified and next stored into the ledger. If a consensus is not reached before a timeout, the network will enter recovery mode

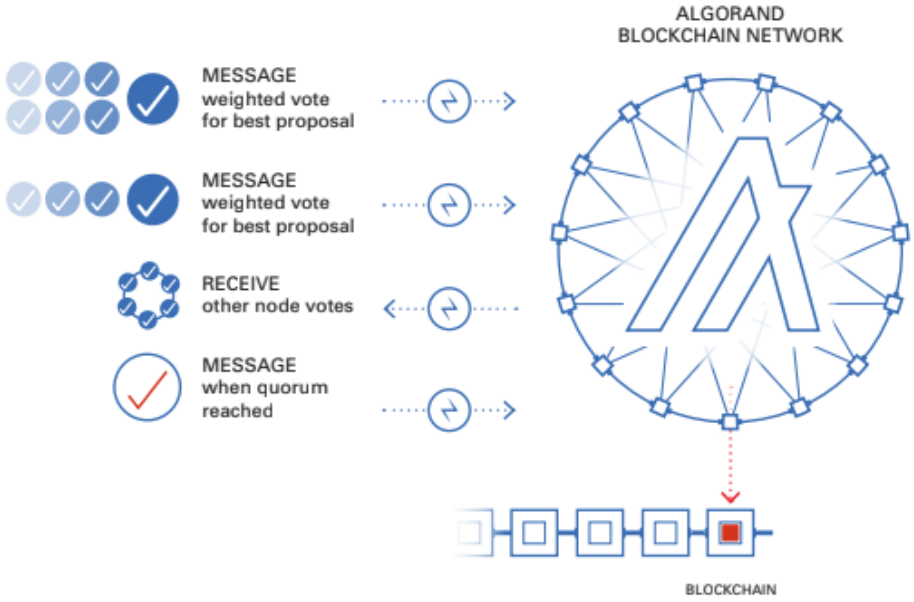


Figure 6: Certify Vote

3.4 Smart Contracts dApps

Algorand Virtual Machine (AVM) supports **Smart Contracts** with Turing-complete languages, it runs on every node of the blockchain containing a stack engine that executes smart signatures and smart contracts. Smart contracts, in Algorand, are written in TEAL, a low-level language that is interpreted by the AVM.

Smart Contracts are digital versions of real world contracts that work as simple autonomous programs that are stored on a blockchain and that can be automatically executed in order to produce powerful and sophisticated decentralized applications. Decentralized Applications, or **dApps**, are programs that run on a distributed computer system such as a blockchain.

Depending on the use case scenario we are running on, we could be interested in the usage of a decentralized application. Let's consider the trivial use case of a bidding system for an auction: the easier way to do it is by developing a centralized server that collects all the offers, and in the end it selects the winner with the highest bid, returning to the others their fundings. In this kind of system we can find some problems, like that the users will have to trust that the server owner will not run off with their bids, or to trust that the code written to hold funds is sound , etc.

In order to do an application like this, we can use a Smart Contract, that is no longer controlled by a central entity but is governed by code that is open and publicly verifiable by anyone. Also, in our case, the money is no longer controlled by a third party entity.

Since Smart Contracts are stored into a blockchain they also gain some properties:

- **immutable**: once a smart contract is created it can be no longer modified, then no one can tamper with the code of the contract.
- **distributed**: the output of the contract is validated by everyone on the network.

Smart contracts, in Algorand, are separated into two main categories:

- smart contracts
- smart signatures

The category of the contract will affect the time and the way which the logic of the program is evaluated into the AVM.

3.4.1 Smart Contracts

Stateful smart contracts form the backbone for the development of the decentralized logic of a dApp on the blockchain, they provide access to on-chain state values and can be combined with other Alogrand layer one features to develop applications that takes advantage of Algorand scalability, performance and quick transaction finality.
The primary function of a Stateful Smart Contract is to provide a mechanism for programatically and modifying an application state on the blockchain.
The application states include a global state associated with the Smart Contract and a local state for individual accounts that interact with it.

Smart contracts also contain logic that, once implemented, may be accessed remotely from any node of the ledger. They can work modifying the global state or the local state of the application and they can access on-chain values.

3.4.2 Smart Signatures

Smart Signatures are stateless contracts, their logic is submitted with the transaction.
They are primarily used for signing transactions or signature delegation.

3.5 Algorand vs Ethereum

We have talked a lot about Algorand, but why not choose a more steady and bigger blockchain, such as **Ethereum**?
Ethereum is a Proof-of-work blockchain, currently is developing a new PoS version of ETH called ETH 2.0, but could pass years until its release.

Being a Proof of Work based blockchain Ethereum have some limits, such as the transactions per second, that currently is between 10-30 TPS [14], that is far below the hypothetical 1,000 TPS of Algorand.

Another "issue" always caused by the PoW mechanism is the gas fees required in order to provide miners for the ETH blockchain. Actually gas fees for ETH are around 4-5$ per transaction, that is very high compared to the 0.001 Algos that result in less than 1$.

Also, being a multi-layered blockchain, Algorand can run smart contracts **on-chain** as well as **off-chain**. Under layer-2 (off-chain), Algorand processes smart contracts, and adds the verified transaction data to blocks on the layer-1 main chain (on-chain). This produces greater processing speeds by lightening the main chain.

Algorand blockchain has also some applications in decentralized finance (DeFi) as well, and for all these reasons Algorand is also being included in the list of "*The Ethereum Killers*".

Conclusions

In this chapter we have seen an overview of Algorand blockchain with how it works, its strengths and its weaknesses. We have talked about Smart Contracts and Smart Signatures, on which we will focus our attention later on.

4 CarPooling App Overview

Introduction

Carpooling is the sharing of car trips with more passengers in order to reduce transportation costs. Drivers can share travel costs by making empty seats in their cars available to others, in exchange for a cost contribution through the carpooling service.

The major advantages of carpooling is the economic savings in terms of fuel, oil, tires, tolls, parking costs, but also the reduction of pollution. Besides this, carpooling requires most of the time to share a vehicle with strangers, which implies trust and security concerns that could be a deterrent.

4.1 General workflow

We will take as a use case the typical car sharing scenario, provided by the most popular carpooling service such as **BlaBlaCar**. In this service, first of all you have to create a user account where some personal information is needed, and you can add another verification layer to your account to increase your credibility toward others.

After creating the account you can choose to **post a ride** or **join an existing one**.
In the first case, you are required to insert the trip information and the n° of passengers available. Also you are required to choose the join requests privacy and the payment method for the other passengers that will join the trip. Also the trip cost (per-person) can be choosed with some limits: you can not set an improperly high price. Completing the form the ride will be publicly available to all the users, and anyone can join the ride.

If you want to join a ride the service allows you to search by a start and a destination, with a date and the n° of passengers availables. The service will search if any trip fits your needs, and will show the results. At this point, if a trip is found, you can reserve your seat in the car, and optionally contact the other car members.

Each member has a rating feedback related to its account. This feedback can increase or decrease as long as other users take a positive / negative vote on their trip with you. This rating system makes sure that some users will be trusted more than others, and encourages users to maintain appropriate behavior during the journey in order to gain a positive rating.

The workflow for a car pooling app can be summarized in several different steps:

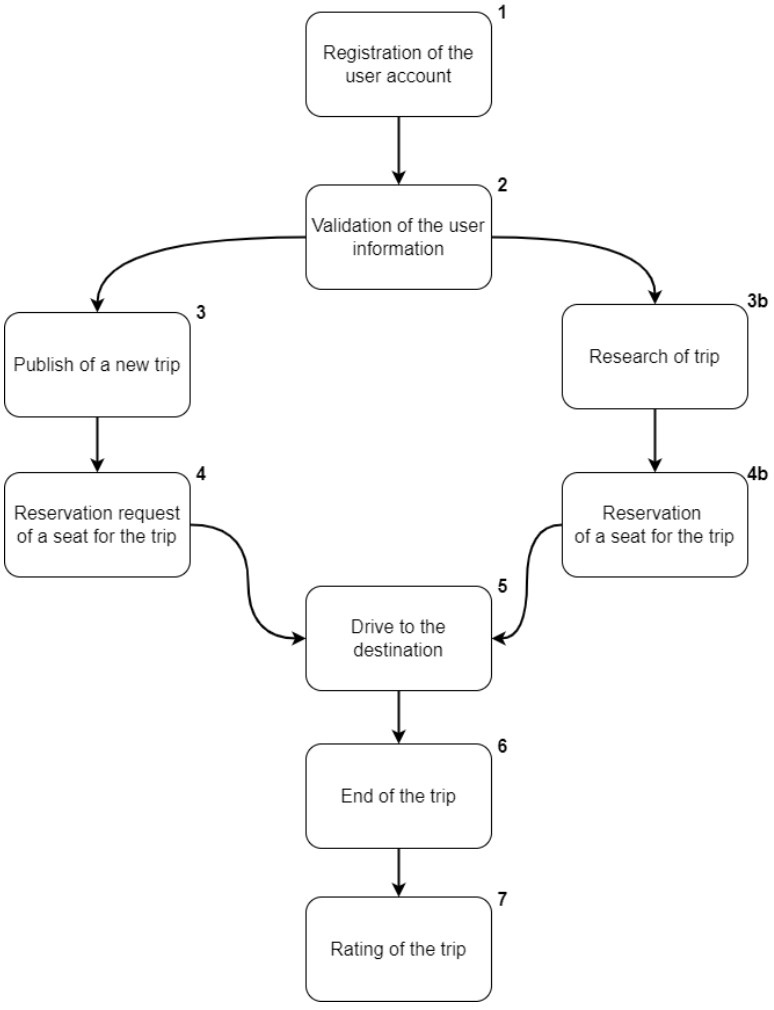


Figure 7: Carpooling general workflow

The steps are:

- **Registration**: the user is required to insert some personal information to create a new account.
- **Validation**: the personal information inserted into an account is verified, usually a valid driver license is required, this step could be optional.
- **Publish / Search a trip**: you can either publish a new trip or search for an existing one.
- **Reservation**: if you are the publisher, you will receive the notification relative to the join request of other passengers, otherwise you could join someone else's available trip.
- **Drive**: the car is driven to the designated destination.
- **End of the trip**: the trip is finished, the destination is reached.
- **Rating**: each passenger can then send feedback about the journey or the other passengers.

4.2 Redesigned workflow

Since this project aims to build an example to redesign the transaction workflow on the application some transparency assumptions and simplifications are made.
First of all we will assume that users are already legitimated, we will not validate an id card or a driver license, we will only store some basic personal information into the blockchain in order to allow an user to use the app.

We will assume that a customer can be either a **car provider** or a **car renter**.
A car provider will be able to publish a trip and its related information. A car renter will be able to join a trip published by a car renter.

Following this scheme, we define the interaction with the smart contract with 4 different entities:

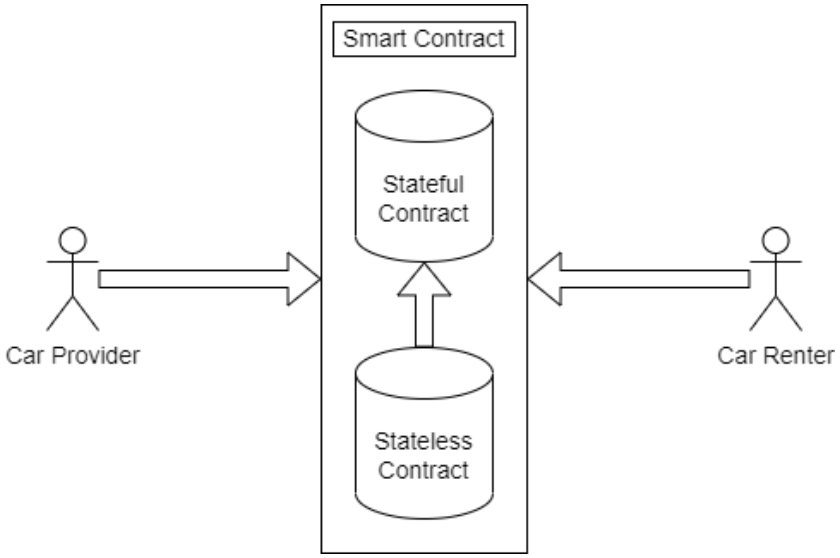


Figure 8: Contract Interactions

The smart contract will be composed by a stateful contract, our dApp, and a stateless contract that will work as an escrow. An user will interact with the application considering the "Smart Contract" as whole.

Here we can define what each entity can do:

- **Car Provider**:
  - register/login an account
  - create a trip
  - update a trip
  - delete a trip
  - terminate the trip
- **Car Renter**:
  - register/login an account
  - perform or cancel a reservation into a trip

The smart contract general interactions workflow will be:

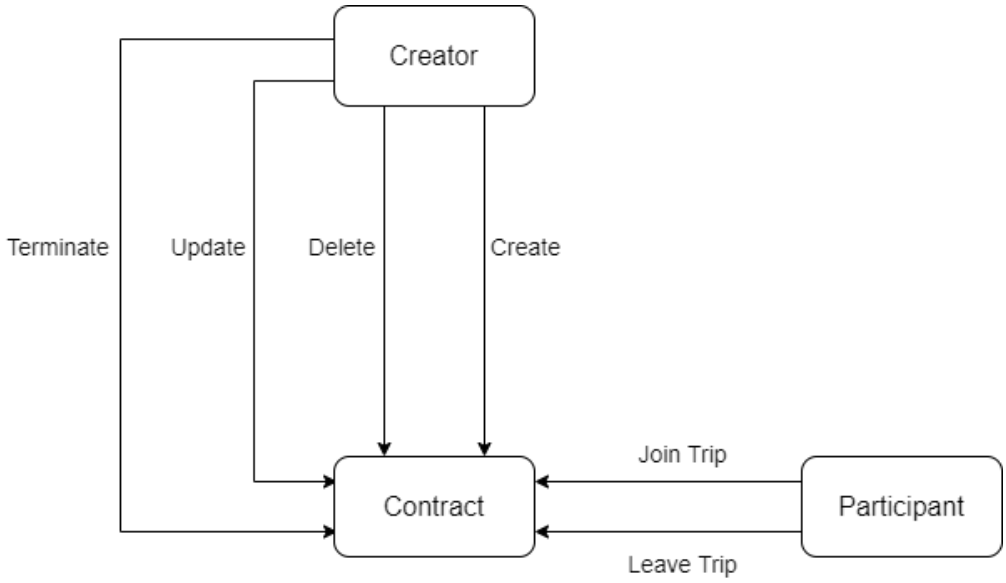


Figure 9: Contract interactions workflow

4.3 Use Cases

In this section will be defined the main use cases for the application workflow. We will take into consideration only the basic actions that the application should perform, and we will see how these actions can be managed and performed with the blockchain.

The basic actions required for the application in order to works are:

- **Account management**: the application should provide a section that allows an user to set his personal account, similar to a login.
- **Trip Creation**: the application should provide a section in order to allow a valid user to create a new trip.
- **Trip Updation**: the application should provide a section in order to allow a valid user to update his own trip.
- **Trip Deletion**: the application should provide a section in order to allow a valid user to delete his own trip.
- **Trip Termination**: the application should provide a section in order to allow a valid user to terminate his own trip when the time comes, and take the earnings from it.
- **Trip Join**: the application should provide a section in order to allow a valid user to join another trip and perform the payment.
- **Trip Leave**: the application should provide a section in order to allow a valid user to leave a currently joined trip and have a refund.

4.3.1 Escrow

Before proceeding to see how to implement the basic actions, we have to define some other statements. A fundamental concept required to perform payment in a secure way is the escrow.
An Escrow is a **legal arrangement** in which a third party holds money or assets for a period of time until a certain condition is satisfied. We will use this concept in order to manage all the payments on our application.

4.3.2 Account Management

Since for our purposes we are using the release node for the blockchain, we can exploit this to transparently manage the account requiring an user to give his Algorand account as a valid account. Some Algorand funded accounts are generated by default with the sandbox creation of a private network, otherwise with a public network an user can generate a new account with the Algorand **goal CLI** [5]. Also an user can generate a new account with the Algorand sdk if required.

For simplicity, since we are in a closed test environment, we will directly require the Account private key for the Algorand account to retrieve the Account information from the blockchain. In a real environment we should use a trusted hardware wallet, such as **perawallet** [20] to manage the accounts correctly and safely.

4.3.3 Trip Creation

This is the general workflow for a trip creation request performed by an user.

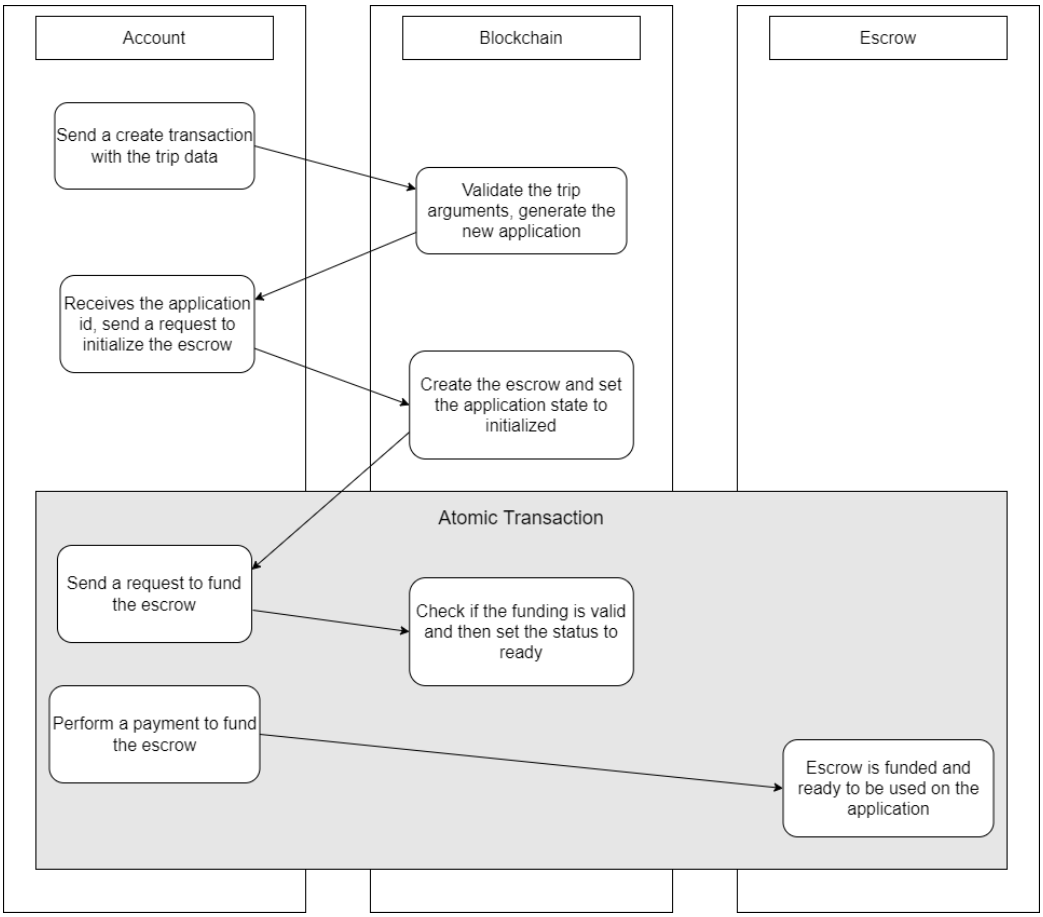


Figure 10: Trip creation workflow

This workflow is performed when a user wants to create a new Trip.
In this scheme 3 different entities are defined:

- **Account**: the Account is settled into the Android client application.
- **Blockchain**: the dApp, the stateful contract stored into the blockchain.
- **Escrow**: the stateless contract linked to the dApp.

Atomic Transfers In this workflow we will use the **atomic transfers** made available by the algorand blockchain. Atomic transfers allow total strangers to transfer assets without the use of a trusted intermediary, ensuring that each party receives exactly what they agreed to. The transactions made in this way are guaranteed to be atomic: all transactions in the request will either pass or fail [4].

The described actions are executed sequentially: first of all, the user requests to create a trip with some information related to the trip. The given information is validated by the stateful contract and, if accepted, the new dApp is created into the blockchain and an `application-id` is returned back to the client.

Next, the client has to initialize and fund the escrow, otherwise the application will not accept any other request. The client, then, sends a transaction to initialize the escrow, that will be validated and executed by the dApp, setting the new internal state to if accepted.

At this point, the client sends a request to fund the escrow with an atomic transaction. In this atomic transfer, two different transactions are performed: the first transaction is needed to transfer the funds from the creator account to the escrow. The second transaction will perform a check on the funding, that is validated, and if accepted will proceed to set the application on the status ready.

4.3.4 Trip Updation

This is the general workflow for a trip creation request performed by an user.

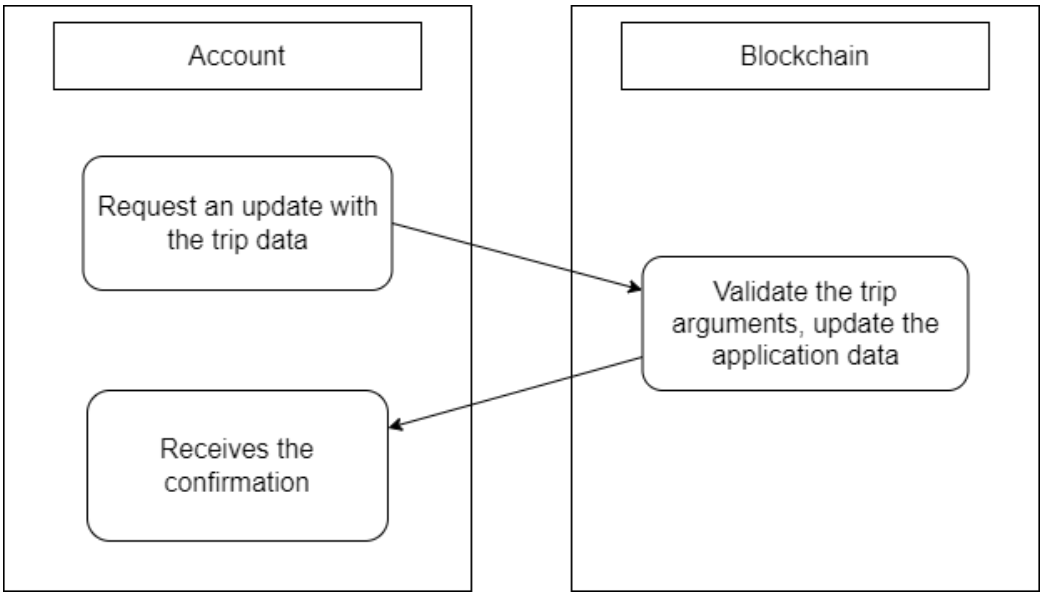


Figure 11: Trip updation workflow

This workflow is performed when a user wants to update an existing trip previously created and that is still valid.

The actions are executed sequentially: the user requests the update of the trip with some new trip information. The given information is validated by the stateful contract and, if accepted, the trip is updated with the new data.

4.3.5 Trip Deletion

This is the general workflow for a trip deletion request performed by an user.

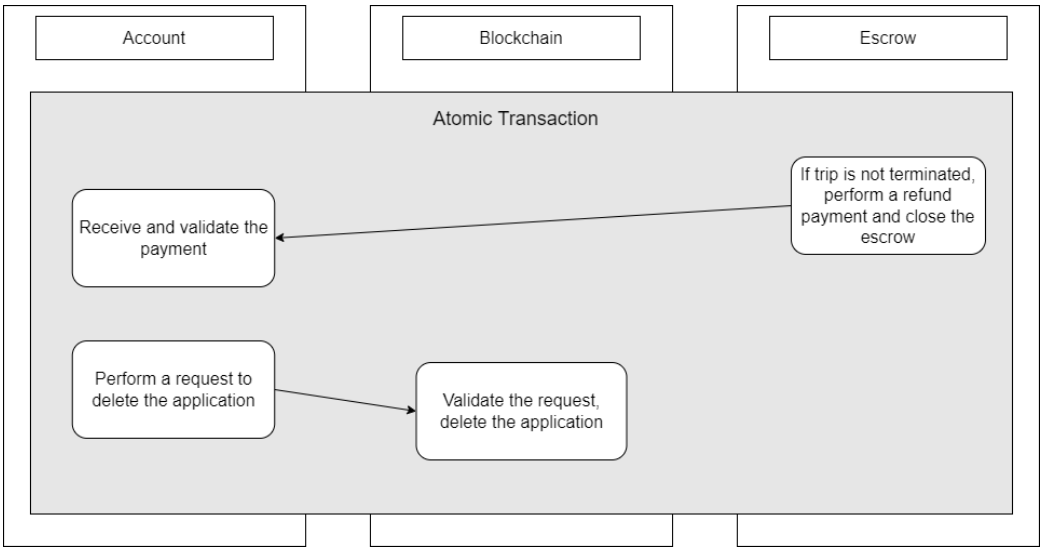


Figure 12: Trip deletion workflow

This workflow is performed when a user wants to delete an existing trip previously created and that is still valid.
This is the more complex case on the deletion workflow, in which the user requests a trip deletion before terminating it, so we need to perform an atomic transaction in order to transfer the remaining funds from the escrow to the creator, closing the escrow, before the deletion takes place.

In the case that a trip is already terminated, instead, we do not need the atomic transfer, but we just need to perform the delete request to the dApp.

In this atomic transfer, two different transactions are performed: the first transaction is needed to transfer the funds from the escrow to the creator account closing the escrow. The second transaction will perform a delete request to the dApp, that is validated, and if accepted will proceed to the app deletion. After the deletion the trip will be no more available or listed.

4.3.6 Trip Termination

This is the general workflow for a trip termination request performed by an user.

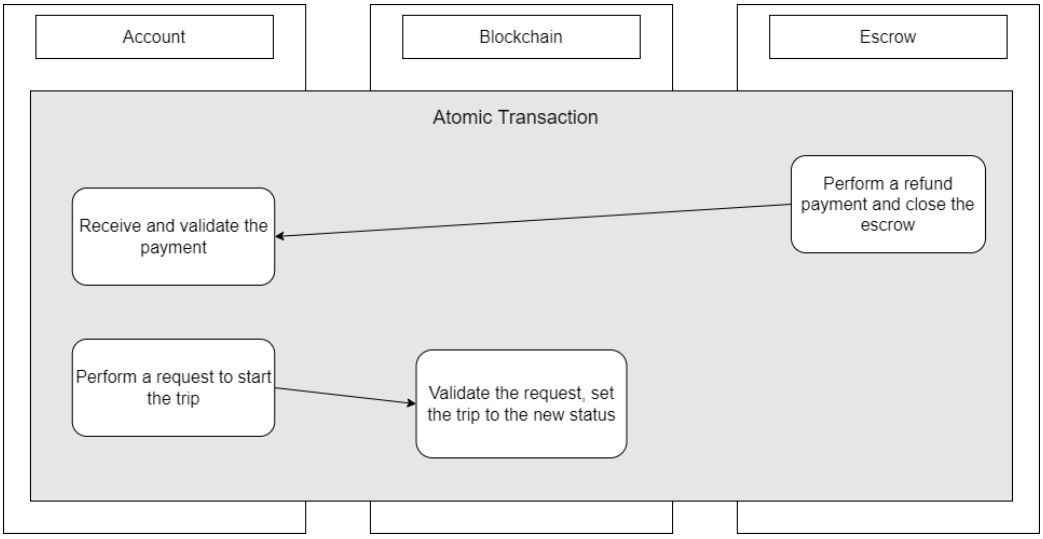


Figure 13: Trip termination workflow

This workflow is performed when a user wants to terminate an existing trip previously created and that is ready to start.

In this atomic transfer, two different transactions are performed: the first transaction is needed to transfer the funds from the escrow to the creator account closing the escrow. The second transaction will perform a request to start the trip to the dApp, that is validated, and if accepted will proceed to the change of status inside the app, that will no longer accept new requests, excluding the deletion operation.

4.3.7 Trip Join

This is the general workflow for a join request performed by an user.

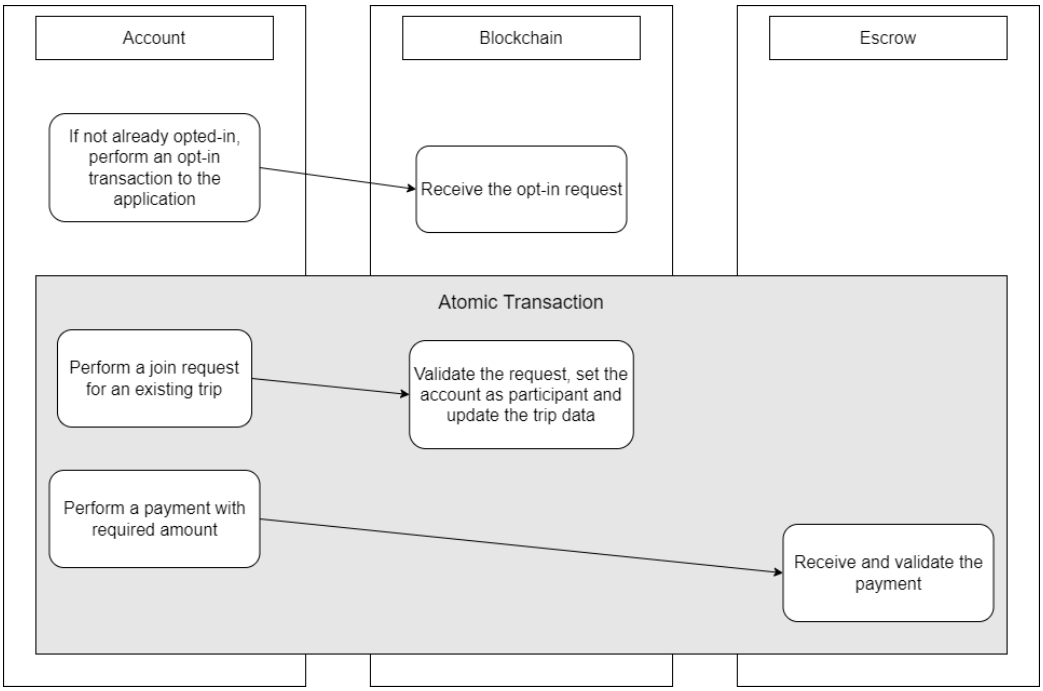


Figure 14: Trip join workflow

This workflow is performed when a user wants to join an existing trip that is valid and not yet started.

The actions are executed sequentially: the first step is to opt-in the user into the dApp in order to access the local-state of the stateful contract. The second step is to perform the join request with an atomic transaction.

In this atomic transfer, two different transactions are performed: the first transaction is needed to perform a payment from the account to the escrow with the trip cost amount. The second transaction will perform a request to let the user join the trip, that is validated, and if accepted will proceed to the change of status inside the user local-state into the contract.

4.3.8 Trip Leave

This is the general workflow for a leave request performed by an user.

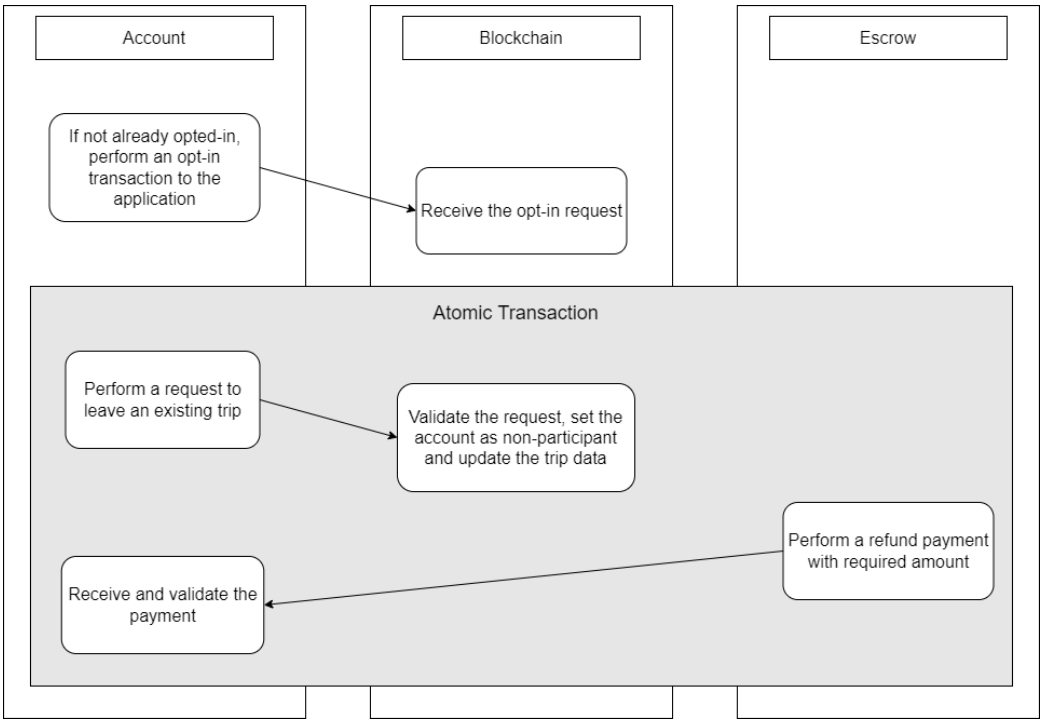


Figure 15: Trip leave workflow

This workflow is performed when a user wants to leave an existing trip that is valid and not yet started.

The actions are executed sequentially: the first step is to opt-in the user into the dApp in order to access the local-state of the stateful contract. The second step is to perform the leave request with an atomic transaction.

In this atomic transfer, two different transactions are performed: the first transaction is needed to perform a refund payment from the escrow to the account with the trip cost amount. The second transaction will perform a request to let the user leave the trip, that is validated, and if accepted will proceed to the change of status inside the user local-state into the contract.

Conclusions

In this chapter we have defined our application model with its workflow and the expected behavior. This section has taken care of the design part of the final application.

5 Development Environment

Introduction

Now we will see an overview of the environment and the tools that are being used in order to develop our application. We will see mostly the Smart Contract development, in which we are more interested in.

5.1 Application Version

The Application is developed in two different versions, both working with the Algorand SDK. These two versions are developed into two different repositories and are open source. We will use the Java version as reference, comparing it to the Python version.

Java The Java version will make use of the Java Algorand SDK to perform the blockchain interaction. This version is developed entirely with Android Studio and is available with an Android mobile application.

Python The Python version will make use of the Python Algorand SDK to perform the blockchain interaction. This version is developed with Python 3.9 and is available with a CLI.

Version Comparison The Java version of the application is available as an Android client application, with an interactive and easy-to-use user interface. Instead, the Python version is available only through CLI. The advantage on the usage of this last version is primarily for debugging and gaining a faster management of the Algorand Accounts and dApps, but it requires a basic knowledge of Python.
On a production mode only the Java version will be deployed, since it is more suitable for a real customer that should have all the features transparently and without any IT knowledge.
Either the 2 versions makes use of the same Blockchain environment, so an app deployed from a version is still available and usable from both the versions.

5.2 Contract Development: PyTeal

Smart Contracts in Algorand are developed in Transaction Execution Approval Language (TEAL), which is an assembly-like language processed by the Algorand Virtual Machine [8]. TEAL programs can be written using a Python library or Reach.

We are interested in the usage of Pyteal, and we will use the Python interpreter to generate the compiled programs that we need. For this project two different TEAl versions are used:

- TEAL v5 for the stateful contract
- TEAL v4 for the stateless contract

There are a lot of rules and constraints while developing with PyTeal, we will see only the most important, in order to get an overview to understand the later code implementation of the contracts.

Data types Pyteal support only two data types:

- **uint64**: 64 bit unsigned integer
- **bytes**: a binary string that can be encoded in different ways

Approve and Reject First of all, a stateful contract must have both the approval program and the clear state program, and they are required to complete their executions with an Approve or a Rejection expression:

- An approve expression is `Return(Int(1))`
- A rejection expression is `Return(Int(0))`

Seq expression One of the most important expressions that we will find out is the `Seq` expression, needed to create a chaining of multiple expressions. This expression will take the value of the last expression as return value, and all the other expressions must not return nothing.
An example of a valid sequence is:

```
Seq(
    Pop(Txn.sender()),
    Return(Int(1))
)
```

Figure 16: Seq expression example

Where the `Pop()` expression is needed to discard the value given by `Txn.sender()`.

Assert expression The `Assert` expression is used to check the validity of a condition before continuing the program, if the condition are not met, the program will stop with a Reject expression.
An example of assertion is:

```
Assert(Txn.type_enum() == TxnType.Payment)
```

Figure 17: Assert expression example

In this example, the program will stop if the transaction is not a payment

Cond expression A `Cond` expression is used in order to chain a sequence of tests and select a result expression, like a switch-statement in a programming language. The tests are checked ordinately, if it results in a 0 value, it will skip through the next test expression, otherwise the expression is evaluated. An example of condition is:

```
Cond([Global.group_size() == Int(5), bid],
     [Global.group_size() == Int(4), redeem],
     [Global.group_size() == Int(1), wrapup])
```

Figure 18: Cond expression example

This condition will execute different behaviors depending on the size of the transaction group.

State Access Stateful contracts can access and modify the state on the Algorand blockchain. A state is a key-value list of pairs, where keys are always byte slices and values can be the same type stated before: **uint64** or **bytes**.
The **state operation table** [11] with all the types of state that a stateful contract can use is:

Context	Write	Read	Delete	Check If Exists
Current App Global	`App.globalPut`	`App.globalGet`	`App.globalDel`	`App.globalGetEx`
Current App Local	`App.localPut`	`App.localGet`	`App.localDel`	`App.localGetEx`
Other App Global		`App.globalGetEx`		`App.globalGetEx`
Other App Local		`App.localGetEx`		`App.localGetEx`

Figure 19: State operation table

On these operations, we are interested in reading / writing the global state and in reading / writing the local state.

The global state is stored on the global context of the application, and is alway accessible for the application. The local state, instead, is stored in a separate context for each account that has opted into the contract and it can only be accessed if an address is passed on the transaction arguments, and that address is associated with an opted-in account.

5.3 Sandbox Environment

In order to deploy any transaction on Algorand Blockchain is needed first of all to set up an Algorand node. This can be made easily with a sandbox node [2], provided by Algorand, with docker and docker-compose. With this sandox node we are able to deploy our dApps as if we are into the real blockchain, and we can perform all our tests.

The sandbox by default will start with the `release` configuration, that is a private network. Is also possible to use a public network configuration, such as the `testnet` configuration. In the `testnet` configuration the sandbox will try to connect to one of the long running networks and connect to the last round. Also to test our contract and perform transactions we need some funded account, we can achieve this generating a new wallet and funding the account on *testnet dispenser* [3]. For our demonstrative purposes we will use the `release` private network, which is created with some already funded accounts and also with a built-in Algorand indexer.

The sandbox node can be setted up with a Docker [13] and a Linux based system, such us a WSL [24].
There is an official guide provided by Algorand to set up a sandbox node, this step is required to perform any kind of request to the blockchain.

5.4 Smart Contract Lifecycle

First of all, let's define the **smart contract lifecycle** [12]. On Algorand, smart contracts are implemented using two programs:

- `ApprovalProgram`: is responsible for processing all application calls to the contract and providing the majority of an application's logic.
- `ClearStateProgram`: is used to deal with accounts that utilize the clear call to delete the smart contract from their balance record.

Both of these programs will succeed either if the stack, upon program termination, will leave one nonzero value or if a positive value is sent to the return opcode.
Also the state updates will not be committed if a global or local state variable is changed and the program fails.

In order to perform calls to the smart contracts `ApplicationCall` transactions are used, and could be:

- **NoOp** - some general application calls to execute the `ApprovalProgram`, this will be the main used type of transactions for our application.
- **DeleteApplication**: delete the application contract.
- **UpdateApplication**: update the application contract.
- **OptIn** - used by accounts to begin participating in a smart contract and access the local state.
- **CloseOut** - used by accounts to close their participation in a smart contract. In case of failure, the account's balance record will not be removed from the contract.
- **ClearState** - The same as **CloseOut**, but the account's balance record is removed from the contract, whether the program succeeds or fails.

The `ClearStateProgram` will manage the `ClearState` type of transactions and all the other types of transactions will be handled by the ApprovalProgram.

Conclusions

In this chapter we have acquired all the basic knowledge needed to write a contract with PyTEAL and also how to set up a testing environment. Developing a Smart Contract requires also to outstand some limits imposed by PyTEAL such as data storage and data management limitations. We will use this information in the next chapter to write the contract.

6 Smart Contracts

Introduction

In this section, we will see the implementation of the stateful / stateless contracts and their interactions with the client application through the SDK. The contracts, as said before, are written with the PyTeal library and they will be compiled into TEAL in order to be deployed into the blockchain.

All the code that will be showed is open source and available between two different repositories on github: Python application with Smart Contracts:

- Python application with Smart Contracts:
  `https://github.com/bara96/algo-carsharing-python`
- Android application:
  `https://github.com/bara96/algo-carsharing-android`

6.1 Stateful Contract

The stateful contract is the most important contract, in fact, once deployed, will be the instance of our dApp implementation. The idea is that **each trip instance will have its own contract** with its information stored into the global state. The client application will retrieve all the valid dApps representing a trip and will manage the previous defined operations on the use cases.

The PyTeal code is taken from the Python version of the implementation, on the `contract_carsharing.py` file. The application schema is similarly defined also in the Java version on the `ApplicationTripSchema.java` interface.

6.1.1 Application Schema

The application schema includes the definition of the **global state schema** and the **local state schema**. For our application logic, we need to store the information into the global / local state.

For the **Global State Schema**:

- creator address: the algorand account address of the creator;
- creator name: a simple username that the creator will use on trip definition;
- departure address: the starting address of the trip;
- arrival address: the ending address of the trip;
- departure date: the starting date and time of the trip;
- departure date rounds: the departure date expressed in algorand block rounds;
- arrival date: the ending date and time of the trip;
- arrival date rounds: the arrival date expressed in algorand block rounds;
- max participants: the max number of participants of the trip;
- trip cost: the unitary cost of the trip, for each participant;
- app state: the internal state of the trip;
- available seats: the n° of seats available;
- escrow address: the linked escrow address;

For the **Local State Schema**:

- is participating: a flag that indicates if an account is participating or not;

This is the definition of all the application variables for the global and the local state:

```
class Variables:
    # Global State Keys
    creator_address = Bytes("creator")                      # Bytes
    creator_name = Bytes("creator_name")                    # Bytes
    departure_address = Bytes("departure_address")          # Bytes
    arrival_address = Bytes("arrival_address")              # Bytes
    departure_date = Bytes("departure_date")                # Bytes
    departure_date_round = Bytes("departure_date_round")    # Int
    arrival_date = Bytes("arrival_date")                    # Bytes
    arrival_date_round = Bytes("arrival_date_round")        # Int
    max_participants = Bytes("max_participants")            # Int
    trip_cost = Bytes("trip_cost")                          # Int
    app_state = Bytes("trip_state")                         # Int
    available_seats = Bytes("available_seats")              # Int
    escrow_address = Bytes("escrow_address")                # Bytes
    # Local State Keys
    is_participating = Bytes("is_participating")            # Int
```

Figure 20: Application variables

In order to correctly store these variables, we need to define the correct state schema on the dApps.

The definition of the Global State Schema is:

```
@property
def global_schema(self):
    """
    global_schema of the contract
    :return:
    """
    return algosdk.future.transaction.StateSchema(num_uints=6,
                                                  num_byte_slices=7)
```

Figure 21: Global state schema

The definition of the Local State Schema is:

```
@property
def local_schema(self):
    """
    local_schema of the contract
    :return:
    """
    return algosdk.future.transaction.StateSchema(num_uints=1,
                                                  num_byte_slices=0)
```

Figure 22: Local state schema

The global state schema, as defined, includes 6 Integers variables and 7 String variables. The local state schema, instead, contains only one Integer variable. These schemas will be provided to the blockchain on the creation transaction. They should coincide with the expected variables, and we should make sure that we do not define more Integers / Strings than required for security reasons.

For clearness and correctness, we also define the application methods that will be callable with a NoOp transaction on the contract definition:

```
class AppMethods:
    initialize_escrow = "initializeEscrow"
    fund_escrow = "fundEscrow"
    update_trip = "updateTrip"
    participate_trip = "participateTrip"
    start_trip = "startTrip"
    cancel_trip_participation = "cancelParticipation"
```

Figure 23: Application methods

6.1.2 Application States

The application logic will include 3 different internal states:

- NOT INITIALIZED
- INITIALIZED
- READY
- FINISHED

```
class AppState:
    not_initialized = Int(0)
    initialized = Int(1)
    ready = Int(2)
    finished = Int(3)
```

Figure 24: Application states

Internal application states are stored inside the global state schema, into the field `trip_state`.

Not initialized In this state, the application is created, this state is set on the create transaction and states that the dApp is created but the escrow is not yet initialized. All the requests will be refused until an escrow is provided.

Initialized In this state, the application is already created and the escrow is initialized, but not funded. This state is set on a `NoOp` transaction that calls the method `initialize_escrow`. All the requests will be refused until the escrow is funded.

Ready In this state, the application is ready and the escrow is funded correctly. This state is set on a `NoOp` transaction that calls the method `fund_escrow`. The application will work as desired until it is terminated or deleted.

Finished In this state, the departure time is passed and the creator has already closed the escrow, taking the earnings. This state is set on a `NoOp` transaction that calls the method `start_trip`. All the requests will be refused except for the deletion.

6.1.3 User States

The application logic will include 2 different user states:

- PARTICIPATING
- NOT PARTICIPATING

```
class UserState:
    participating = Int(1)
    not_participating = Int(0)
```

Figure 25: User states

User states are stored inside the local state schema, into the field `is_participating`.

Participating In this state, the user is flagged as a participant, it cannot join the trip again but he can leave it. This state is set on a `NoOp` transaction that calls the method `participate`.

Not Participating In this state, the user is flagged as not participating, it cannot leave the trip but he can join it. By default, if the user state is not yet defined, this state is considered. This state is set on a `NoOp` transaction that calls the method `cancel_participation`.

6.1.4 Application Logic

In the next sections, we will see the stateful contract methods and the logic behind them. First of all, the contract requires the `approval program` and the `clear state program`.

```
def approval_program(self):
    """
    approval_program of the contract
    :return:
    """
    return self.application_start()
```

Figure 26: Approval program

```
def clear_program(self):
    """
    clear_state_program of the contract
    :return:
    """
    trip_finished = App.globalGet(self.Variables.app_state) == self.AppState.finished

    return Seq(
        Assert(trip_finished),
        Return(Int(1))
    )
```

Figure 27: Clear state program

The approval program will return the expression contained into the `application_start()` function. The `clear state program` will approve the transaction if the trip is finished.
The `application_start` is the main method of our contract, and will be responsible to manage all the transactions performed to the dApp.

We can focus our attention on the actions variable, that is a Condition expression that will manage the right app behavior based on the transaction type. For the deletion and updation methods also some other checks are made:

- `can_update`: the transaction can take place if there are no participants, the trip is not started and the sender is the creator.
- `can_delete`: the transaction can take place if the sender is the creator and there are no participants, or the trip is finished.

Note: the update refers to the transaction that updates the contract programs but not the trip information. This method exists but is not called by the client.

In order to handle the `NoOp` transactions a sequence expression made of Conditions tests is created. Depending on the arguments of the `NoOp` transaction, the right method is chosen.
The available methods are the one defined previously.

```
def application_start(self):
    """
    Start the application, check with transaction to execute
    :return:
    """
    is_creator = Txn.sender() == App.globalGet(self.Variables.creator_address)

    handle_noop = Seq(
        Cond(
            [Txn.application_args[0] == Bytes(self.AppMethods.initialize_escrow),
             self.initialize_escrow(escrow_address=Txn.application_args[1])],

            [Txn.application_args[0] == Bytes(self.AppMethods.update_trip),
             self.update_trip()],

            [Txn.application_args[0] == Bytes(self.AppMethods.participate_trip),
             self.participate_trip()],

            [Txn.application_args[0] == Bytes(self.AppMethods.cancel_trip_participation),
             self.cancel_participation()],

            [Txn.application_args[0] == Bytes(self.AppMethods.start_trip),
             self.start_trip()]
        )
    )

    no_participants = App.globalGet(self.Variables.available_seats) == App.globalGet(
        self.Variables.max_participants)
    trip_started = App.globalGet(self.Variables.app_state) == self.AppState.started

    can_update = And(
        is_creator,
        no_participants,
        Not(trip_started)
    )

    can_delete = And(
        is_creator,
        Or(no_participants, trip_started)
    )

    actions = Cond(
        [Txn.application_id() == Int(0), self.app_create()],
        [Txn.on_completion() == OnComplete.OptIn, self.opt_in()],
        [Txn.on_completion() == OnComplete.NoOp, handle_noop],
        [Txn.on_completion() == OnComplete.UpdateApplication, Return(can_update)],
        [Txn.on_completion() == OnComplete.DeleteApplication, Return(can_delete)],

    )

    return actions
```

Figure 28: Application start method

Application Creation This method will be triggered on the `application_start()` when the test condition that states that the application id is 0 is met.
In this case the `app_create()` method is called.

```
def app_create(self):
    """
    CreateAppTxn
    Set the global_state of the app with given params
    Perform some checks for params validity
    :return:
    """
    valid_number_of_args = Txn.application_args.length() == Int(9)

    return Seq([
        Assert(valid_number_of_args),
        App.globalPut(self.Variables.creator_address, Txn.sender()),
        App.globalPut(self.Variables.creator_name, Txn.application_args[0]),
        App.globalPut(self.Variables.departure_address, Txn.application_args[1]),
        App.globalPut(self.Variables.arrival_address, Txn.application_args[2]),
        App.globalPut(self.Variables.departure_date, Txn.application_args[3]),
        App.globalPut(self.Variables.departure_date_round, Btoi(Txn.application_args[4])),
        App.globalPut(self.Variables.arrival_date, Txn.application_args[5]),
        App.globalPut(self.Variables.arrival_date_round, Btoi(Txn.application_args[6])),
        App.globalPut(self.Variables.trip_cost, Btoi(Txn.application_args[7])),
        App.globalPut(self.Variables.max_participants, Btoi(Txn.application_args[8])),
        App.globalPut(self.Variables.available_seats, Btoi(Txn.application_args[8])),
        App.globalPut(self.Variables.app_state, self.AppState.not_initialized),
        Assert(Global.round() <= App.globalGet(self.Variables.departure_date_round)),  # check dates are valid
        Assert(App.globalGet(self.Variables.departure_date_round) < App.globalGet(self.Variables.arrival_date_round)),
        Assert(App.globalGet(self.Variables.max_participants) > Int(0)),  # at least a seat
        Return(Int(1))
    ])
```

Figure 29: application create method

This method will perform some validation, insert the trip information into the global schema and create the new dApp.

The first assertion states that the transactions arguments must be 9 in the following order:

- creator name
- departure address
- arrival address

- departure date
- departure date round
- arrival date
- arrival date round
- trip cost
- max participants / available seats

These arguments will be saved in the global state.

Since inside the blockchain there is no such concept of date, we have to compare the time status using **rounds**. A date can be converted in rounds with some calculations [10].

The creator address is saved into the global state from this transaction.
The state is set to NOT INITIALIZED.
A check that the departure date is greater than the current date and the arrival date is performed. Also a check that at least a seat is available is also made, otherwise the trip would be meaningless.
If all these validations are `ok`, the new application is created and the `app-id` is given back to the client.

Escrow Initialization This method will be triggered on the `application_start()` when the test condition states that the transaction type is a `NoOp` transaction and the requested method into the arguments is `initializeEscrow`.
In this case the `initialize_escrow()` method is called.

```
def initialize_escrow(self, escrow_address):
    """
    NoOpTxn
    Initialize an escrow for this application
    :return:
    """
    curr_escrow_address = App.globalGetEx(Int(0), self.Variables.escrow_address)
    valid_number_of_transactions = Global.group_size() == Int(1)
    is_creator = Txn.sender() == App.globalGet(self.Variables.creator_address)

    update_state = Seq([
        App.globalPut(self.Variables.escrow_address, escrow_address),
        App.globalPut(self.Variables.app_state, self.AppState.initialized),
    ])
    return Seq([
        curr_escrow_address,
        Assert(curr_escrow_address.hasValue() == Int(0)),
        Assert(valid_number_of_transactions),
        Assert(is_creator),
        update_state,
        Return(Int(1))
    ])
```

Figure 30: initialize escrow method

This method will initialize the escrow, any other request will not be satisfied until the internal state of the app is NOT INITIALIZED. Before changing the state to INITIALIZED and assign the escrow, some validations are made: this method can be only called by the creator and the escrow should be already unset.

Escrow Funding This method will be triggered on the `application_start()` when the test condition states that the transaction type is a `NoOp` transaction and the requested method into the arguments is `fundEscrow`.
In this case the `fund_escrow()` method is called.

```
def fund_escrow(self):
    """
    NoOpTxn
    Fund an escrow for this application
    :return:
    """
    valid_number_of_transactions = Global.group_size() == Int(2)
    is_creator = Txn.sender() == App.globalGet(self.Variables.creator_address)
    trip_init = App.globalGet(self.Variables.app_state) == self.AppState.initialized

    # check if the payment is valid
    valid_payment = And(
        Gtxn[1].type_enum() == TxnType.Payment,
        Gtxn[1].receiver() == App.globalGet(self.Variables.escrow_address),
        Gtxn[1].amount() == self.Constants.escrow_min_balance,
        Gtxn[1].sender() == Gtxn[0].sender(),
    )

    update_state = Seq([
        App.globalPut(self.Variables.app_state, self.AppState.ready),
    ])

    return Seq([
        Assert(trip_init),
        Assert(is_creator),
        Assert(valid_number_of_transactions),
        Assert(valid_payment),
        update_state,
        Return(Int(1))
    ])
```

Figure 31: fund escrow method

This method will fund the escrow, any other request will not be satisfied until the internal state of the app is INITIALIZED.
Before changing the state to READY, some validations are made: this method can be only called by the creator, the escrow has already been initialized and

the current transaction is an atomic transaction with a valid payment. The payment transaction should satisfy some requirements: the receiver is the escrow, the amount is the minimum balance required and the sender is the same that has made the request. If the assertions are `ok`, the application will be on READY state and will work as expected.

Application Updation This method will be triggered on the `application_start()` when the test condition states that the transaction type is a `NoOp` transaction and the requested method into the arguments is `updateTrip`.
In this case the `update_trip()` method is called.

```
def update_trip(self):
    """
    UpdateAppTxn
    Update the global_state of the app with given params
    Perform some checks for params validity
    :return:
    """
    valid_number_of_args = Txn.application_args.length() == Int(10)
    no_participants = App.globalGet(self.Variables.available_seats) == App.globalGet(
        self.Variables.max_participants)
    trip_ready = App.globalGet(self.Variables.app_state) == self.AppState.ready
    is_creator = Txn.sender() == App.globalGet(self.Variables.creator_address)

    can_update = And(
        no_participants,
        trip_ready,
    )

    return Seq([
        Assert(valid_number_of_args),
        Assert(is_creator),
        Assert(can_update),
        App.globalPut(self.Variables.creator_name, Txn.application_args[1]),
        App.globalPut(self.Variables.departure_address, Txn.application_args[2]),
        App.globalPut(self.Variables.arrival_address, Txn.application_args[3]),
        App.globalPut(self.Variables.departure_date, Txn.application_args[4]),
        App.globalPut(self.Variables.departure_date_round, Btoi(Txn.application_args[5])),
        App.globalPut(self.Variables.arrival_date, Txn.application_args[6]),
        App.globalPut(self.Variables.arrival_date_round, Btoi(Txn.application_args[7])),
        App.globalPut(self.Variables.trip_cost, Btoi(Txn.application_args[8])),
        App.globalPut(self.Variables.max_participants, Btoi(Txn.application_args[9])),
        App.globalPut(self.Variables.available_seats, Btoi(Txn.application_args[9])),
        Assert(Global.round() <= App.globalGet(self.Variables.departure_date_round)),  # check dates are valid
        Assert(
            App.globalGet(self.Variables.departure_date_round) < App.globalGet(self.Variables.arrival_date_round)),
        Assert(App.globalGet(self.Variables.max_participants) > Int(0)),  # at least a seat
        Return(Int(1))
    ])
```

Figure 32: update trip method

This method is very similar to the creation one, it updates the global state with the given information as before, but in this case the creator address will not change, also some other validations are performed.

The accepted number of arguments now is 10, because the first argument is the name of the requested method. Additionally to the application cre-

ation checks, another assertion is made to ensure that: the sender is the creator, there are no participants to the trip and the internal state is READY.

Application Deletion This action will be performed on the `application_start()` when the test condition states that the transaction type is a `Delete` transaction. In this case, if the sender is the creator and there are no participants, or the application is FINISHED, the dApp is deleted from the blockchain.

Application Termination This method will be triggered on the `application_start()` when the test condition states that the transaction type is a `NoOp` transaction and the requested method into the arguments is `startTrip`.
In this case the `start_trip()` method is called.

```
def start_trip(self):
    """
    NoOpTxn
    The creator start the trip
    Perform validity checks and payment checks
    :return:
    """
    is_creator = Txn.sender() == App.globalGet(self.Variables.creator_address)
    valid_number_of_transactions = Global.group_size() == Int(2)

    can_start = And(
        App.globalGet(self.Variables.app_state) == self.AppState.ready,
        is_creator,  # creator only can perform this action
        Global.round() >= App.globalGet(self.Variables.departure_date_round),  # check if trip is started
        valid_number_of_transactions
    )

    valid_payment = And(
        Gtxn[1].type_enum() == TxnType.Payment,
        Gtxn[1].receiver() == App.globalGet(self.Variables.creator_address),
        Gtxn[1].sender() == App.globalGet(self.Variables.escrow_address),
    )

    update_state = Seq([
        App.globalPut(self.Variables.app_state, self.AppState.finished),
        Return(Int(1))
    ])

    return Seq([
        Assert(can_start),
        Assert(valid_payment),
        update_state,
        Return(Int(1))
    ])
```

Figure 33: start trip method

This method will transfer the remaining fund from the escrow, closing it, to the creator, and will set the trip state to STARTED. The validations performed are on the `can_start` Assertion, where is checked if the trip is in ready state, if the sender is the creator and if the departure date is passed. Another validation is made on the payment, this transaction is made with an atomic transaction with 2 transactions.
If the assertions pass through, the trip state is set to FINISHED and will reject

any other request apart from the deletion.

Application Opt-in This method will be triggered on the `application_start()` when the test condition states that the transaction type is an `OptIn` transaction.
In this case the `opt_in()` method is called.

```
def opt_in(self):
    """
    OptInTxn
    Opt In a user to allow the usage of local_state
    :return:
    """
    is_creator = Txn.sender() == App.globalGet(self.Variables.creator_address)
    trip_ready = App.globalGet(self.Variables.app_state) == self.AppState.ready

    return Seq([
        Assert(trip_ready),
        Assert(Not(is_creator)),
        Assert(App.globalGet(self.Variables.app_state) == self.AppState.ready),
        Assert(Global.round() <= App.globalGet(self.Variables.departure_date_round)),
        Assert(App.globalGet(self.Variables.available_seats) > Int(0)),
        Return(Int(1))
    ])
```

Figure 34: start trip method

This method `optin` the account into the application, enabling the access to the local state, needed for the join / leave operations. The validations performed are: the account can not be the creator, the application is in READY state, the departure time has not come yet and there is at least an available seat.

Application Join This method will be triggered on the `application_start()` when the test condition states that the transaction type is a `NoOp` transaction and the requested method into the arguments is `participateTrip`.
In this case the `participate_trip()` method is called.

This method will join an user to an existing trip, performing also the payment to the escrow. It is required that the user has opted-in into the application to call this method. This method must be called with an atomic transaction with a payment.

In this context, basically there are 2 different validations: if the sender can participate and if the payment is valid.

The sender can participate if:

- is not already PARTICIPATING;
- the application is on READY status;
- it is not the creator;
- there is at least a seat available;
- the departure time has not already passed.

The payment is valid if:

- the sender is the same that has made the request;
- the amount is correct;
- the receiver is the escrow.

If all the assertions are `ok`, the user's local state will be changed, setting the `is_participating` field to the PARTICIPATING state, and the available seats are reduced by one.

```python
def participate_trip(self):
    """
    NoOpTxn
    A user want to participate the trip
    Perform validity checks and payment checks
    :return:
    """
    get_participant_state = App.localGetEx(Int(0), App.id(), self.Variables.is_participating)
    available_seats = App.globalGet(self.Variables.available_seats)
    valid_number_of_transactions = Global.group_size() == Int(2)
    is_creator = Txn.sender() == App.globalGet(self.Variables.creator_address)

    is_not_participating = Or(
        Not(get_participant_state.hasValue()),
        get_participant_state.value() == Int(0),
    )

    # check if user can participate
    can_participate = And(
        App.globalGet(self.Variables.app_state) == self.AppState.ready,
        Not(is_creator),
        App.globalGet(self.Variables.available_seats) > Int(0),  # check if there is an available seat
        Global.round() <= App.globalGet(self.Variables.departure_date_round),  # check if trip is started
        valid_number_of_transactions,
    )

    # check if the payment is valid
    valid_payment = And(
        Gtxn[1].type_enum() == TxnType.Payment,
        Gtxn[1].receiver() == App.globalGet(self.Variables.escrow_address),
        Gtxn[1].amount() == App.globalGet(self.Variables.trip_cost),
        Gtxn[1].sender() == Gtxn[0].sender(),
    )

    update_state = Seq([
        # check if user is not already participating
        get_participant_state,
        Assert(is_not_participating),
        # update state
        App.globalPut(self.Variables.available_seats, available_seats - Int(1)),  # decrease seats
        App.localPut(Int(0), self.Variables.is_participating, Int(1)),  # set user as participating
    ])

    return Seq([
        Assert(can_participate),
        Assert(valid_payment),
        update_state,
        Return(Int(1))
    ])
```

Figure 35: participate trip method

Application Leave This method will be triggered on the `application_start()` when the test condition states that the transaction type is a `NoOp` transaction and the requested method into the arguments is `cancelParticipation`.
In this case the `cancel_participation()` method is called.

This method will let an already joined user to leave an existing trip, performing also the refund from the escrow. It is required that the user has opted-in into the application to call this method. This method must be called with an atomic transaction with a payment.

In this context, basically there are 2 different validations: if the sender can cancel the participation and if the payment is valid.

The sender can leave if:

- is already PARTICIPATING;
- the application is on READY status;
- it is not the creator;
- the departure time has not already passed.

The payment is valid if:

- the sender is the escrow;
- the amount is correct;
- the receiver is the same that has made the request.

If all the assertions are `ok`, the user's local state will be changed, setting the `is_participating` field to the NOT PARTICIPATING state, and the available seats are increased by one.

```python
def cancel_participation(self):
    """
    NoOpTxn
    A user want to cancel trip participation
    Perform validity checks and payment-refund checks
    :return:
    """
    get_participant_state = App.localGetEx(Int(0), App.id(), self.Variables.is_participating)
    available_seats = App.globalGet(self.Variables.available_seats)
    valid_number_of_transactions = Global.group_size() == Int(2)
    is_creator = Txn.sender() == App.globalGet(self.Variables.creator_address)
    is_participating = And(
        get_participant_state.hasValue(),
        get_participant_state.value() == Int(1),
    )

    # check if user can cancel participation
    can_cancel = And(
        App.globalGet(self.Variables.app_state) == self.AppState.ready,
        Not(is_creator),
        Global.round() <= App.globalGet(self.Variables.departure_date_round),  # check if trip is started
        valid_number_of_transactions,
    )

    valid_refund = And(
        Gtxn[1].type_enum() == TxnType.Payment,
        Gtxn[1].receiver() == Gtxn[0].sender(),
        Gtxn[1].amount() == App.globalGet(self.Variables.trip_cost),
        Gtxn[1].sender() == App.globalGet(self.Variables.escrow_address),
    )

    update_state = Seq([
        # check if user is already participating
        get_participant_state,
        Assert(is_participating),
        # update state
        App.globalPut(self.Variables.available_seats, available_seats + Int(1)),  # increase seats
        App.localPut(Int(0), self.Variables.is_participating, Int(0)),  # set user as not participating
        Return(Int(1))
    ])

    return Seq([
        Assert(can_cancel),
        Assert(valid_refund),
        update_state,
        Return(Int(1))
    ])
```

Figure 36: cancel participation method

6.2 Stateless Contract

The stateless contract will work as **escrow** for the stateful contract, in order to manage the payments for the trip safely. The stateless contracts will be used as *contract account* mode [7], so each escrow will have his own address to perform and receive payments. Each trip will be assigned with an escrow that will validate each payment transaction confirming or rejecting it.

The PyTeal code is taken from the Python version of the implementation, on the `contract_escrow.py` file.

```
def contract_escrow(app_id: int):
    """
    Stateless contract to perform payments for a contract
    :param app_id:
    :return:
    """
    return Seq([
        Assert(Global.group_size() == Int(2)),  # atomic transfer with two transactions
        Assert(Gtxn[0].application_id() == Int(app_id)),    # check the application

        Assert(Gtxn[1].type_enum() == TxnType.Payment),

        Return(Int(1))
    ])
```

Figure 37: escrow contract

The contract is composed by a sequence expression that will make some assertion, if all pass, the atomic transaction is accepted. The first assertion states that the escrow will accept only atomic transactions with 2 transactions. The second assertion checks that the first transaction in the group is being made to the right dApp associated with this escrow. The last assertion checks that the second transaction in the group is a payment transaction.

Conclusions

In this chapter we have managed all the previous designed use cases into our Smart Contract.
The provided design is been studied in order to solve and mitigate these development limitations in order to make our application work proficiently. A big limitation is for instance the global state, where you can store up to 64 key / value pairs and the pair can be no more than 128 bytes. This limitation has carried out the design choice to create a distinct contract for each trip instance instead of making a single big contract for all the trips. Also another big issue is the impossibility to retrieve directly from a contract the accounts that are opted-in, this led us to the usage of the Algorand indexer for this kind of purpose.
At this point the contract is ready to be used by a client application.

7 Algorand Indexer

A very important Algorand tools that this application will make use of is the Algorand indexer [6]. The Indexer is primarily used to search transactions into the blockchain with some REST Api disposal by Algorand.
The indexer query data from the blockchain and store them into a PostgreSQL [26] database for faster retrieval. It can be used to search for transactions, accounts, assets and blocks, each one supported with filter parameters.

For our application we are interested in the search of Application Calls transactions with an encoded note that will be inserted on the create transaction of each dApp. Also we will use the indexer in order to get and verify the account information inserted into the application, like the balance and the local state.

The indexer client, on the Android application, is instantiated and used in two different services:

- `AccountService.java`: provide only methods to get account-related information from the indexer REST Api.
- `IndexerService.java`: provide methods to interact with the indexer REST Api.

`AccountService` is used to get the account related information filtering by address.

```
public Supplier<Account> getAccountInfo(String address) {
    return () -> {
        try {
            Address pk = new Address(address);

            Response<Account> response = client.AccountInformation(pk).execute();
            ServicesHelper.checkResponse(response);

            return response.body();
        }
        catch (Exception e) {
            throw new CompletionException(e);
        }
    };
}
```

Figure 38: getAccountInfo method

`IndexerService` is used to get transactions and application information from the blockchain.

```
public Supplier<TransactionsResponse> getTransactions() {
    return () -> {
        try {
            Response<TransactionsResponse> response = client.searchForTransactions()
                    .notePrefix(transactionNote.getBytes())
                    .txType(Enums.TxType.APPL)
                    .limit(1000L)
                    .execute();
            ServicesHelper.checkResponse(response);

            return response.body();
        }
        catch (Exception e) {
            throw new CompletionException(e);
        }
    };
}
```

Figure 39: getTransactions method

The transactions are retrieved and displayed on a list, filtering only the transactions with type application and with an **encoded note**, that will be setted on the create transaction of each dApp, in order to retrieve all the `app-ids` of the trips.

```
public Supplier<ApplicationResponse> getApplication(Long appid) {
    return () -> {
        try {
            Response<ApplicationResponse> response = client.lookupApplicationByID(appid).execute();
            ServicesHelper.checkResponse(response);

            return response.body();
        }
        catch (Exception e) {
            throw new CompletionException(e);
        }
    };
}
```

Figure 40: getApplication method

When an `app-id` is successfully recovered from the transactions, the application data is retrieved from the blockchain with this method.

8 Client Application

Introduction

In this section will be illustrated the mechanics behind the Android application that makes use of the Algorand SDK to interact with the dApp. All the next described logic is implemented on the `ApplicationService.java` file.

8.1 Trip Creation

When a user makes a request to create a trip, a series of transactions will sequentially start.
First of all, a `Create` transaction is performed from the Account to the Contract, that will be managed by the `app_create` method of the contract:

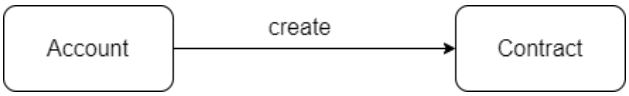


Figure 41: app_create

If the dApp creation succeed, a `NoOp` transaction to initialize the escrow is performed, that will be managed by the `initialize_escrow` method of the contract:

Figure 42: initialize_escrow

If the escrow is correctly initialized, an atomic transaction with a `payment` transaction and a `NoOp` transaction is performed and will be managed by the `fund_escrow` method of the contract:

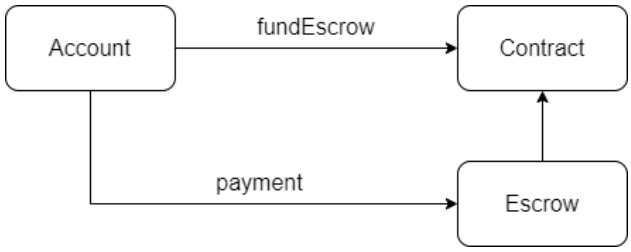


Figure 43: fund_escrow

The first use case is implemented in the following functions:
`createApplication(Context context, Account creator, InsertTripModel tripArgs)`

```
TEALProgram approvalProgram = getCompiledProgram(context, ProgramType.ApprovalState).get();
TEALProgram clearStateProgram = getCompiledProgram(context, ProgramType.ClearState).get();

StateSchema globalState = ApplicationTripSchema.getGlobalStateSchema();
StateSchema localState = ApplicationTripSchema.getLocalStateSchema();

List<byte[]> args = tripArgs.getArgs(client);

// create application
Transaction txn = TransactionsHelper.create_txn(client, creator.getAddress(),
        approvalProgram, clearStateProgram, globalState, localState, args);
SignedTransaction signedTxn = creator.signTransaction(txn);

String txId = TransactionsHelper.sendTransaction(client, signedTxn);
PendingTransactionResponse response = TransactionsHelper.waitForConfirmation(client, txId);

LogHelper.log(this.getClass().getName(), String.format("Created new app-id: %s", response.applicationIndex));
return response.applicationIndex;
```

Figure 44: createApplication method

In this code snippet, the compiled programs and the contract schema are sended into a `Create` transaction with the trip information. Each transaction must be signed with the private key of the sender. After sending the transaction we will wait for the confirmation, usually it comes in a few rounds (a few seconds). After this transaction the dApp internal state will be NOT INITIALIZED.

The last two use cases are implemented in the following function:
`initializeEscrow(Long appId, Account creator)`

We will split this method in two parts:

```
// get escrow address
LogicsigSignature escrowSignature = getEscrowSignature(appId).get();
Address escrowAddress = escrowSignature.toAddress();

List<byte[]> args  = new ArrayList<>();
args.add(ApplicationTripSchema.AppMethod.InitializeEscrow.getValue().getBytes());
args.add(escrowAddress.getBytes());

// link the escrow to the application
Transaction txn = TransactionsHelper.noop_txn(client, appId, creator.getAddress(), args);
SignedTransaction signedTxn = creator.signTransaction(txn);

String txId = TransactionsHelper.sendTransaction(client, signedTxn);
PendingTransactionResponse response = TransactionsHelper.waitForConfirmation(client, txId);
```

Figure 45: initializeEscrow method part 1

This first snippet of code is responsible to create the escrow from the compiled stateless contract and send it to the dApp in order to link them, a `NoOp` transaction is performed to do that.
Since we are calling the `initialize_escrow` method of the stateful contract, we pass its name as argument in the `NoOp` transaction.
After this transaction the dApp internal state will be INITIALIZED.

```
// fund escrow
List<byte[]> args_fund  = new ArrayList<>();
args_fund.add(ApplicationTripSchema.AppMethod.FundEscrow.getValue().getBytes());

Transaction call_txn = TransactionsHelper.noop_txn(client, appId, creator.getAddress(), args_fund);
Transaction payment_txn = TransactionsHelper.payment_txn(client, creator.getAddress(),
        escrowAddress, TransactionsHelper.escrowMinBalance,  closeReminderTo: null);

// group transactions an assign ids
Digest gid = TxGroup.computeGroupID(call_txn, payment_txn);
call_txn.assignGroupID(gid);
payment_txn.assignGroupID(gid);

// sign individual transactions
SignedTransaction call_signedTxn = creator.signTransaction(call_txn);
SignedTransaction payment_signedTxn = creator.signTransaction(payment_txn);

// send transactions
String txId_fund = TransactionsHelper.sendTransaction(client, Arrays.asList(call_signedTxn, payment_signedTxn));
PendingTransactionResponse response_fund = TransactionsHelper.waitForConfirmation(client, txId_fund);

LogHelper.log(this.getClass().getName(),
        String.format("Escrow initialized for app-id %s with address: %s", appId, escrowAddress));
return appId;
```

Figure 46: initializeEscrow method part 2

In this second code snippet, an atomic transaction with a `payment` transaction and a `NoOp` transaction is made. The `payment` transaction is the initial funding from the creator to the escrow and the `NoOp` transaction will be managed by the stateful contract validating the request.
Since we are calling the `fund_escrow` method of the stateful contract, we pass its name as argument in the `NoOp` transaction.
After this transaction the dApp internal state will be READY.

8.2 Trip Updation

When a user makes a request to update a trip, a `NoOp` transaction is performed from the Account to the Contract, that will be managed by the `update_trip` method of the contract:

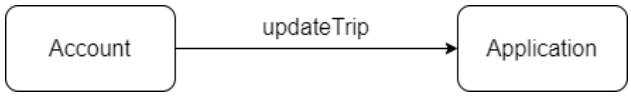


Figure 47: update_trip

This use case is implemented in the following function:
`updateApplication(Long appId, Account creator, InsertTripModel tripArgs)`

```
List<byte[]> args = new ArrayList<>();
args.add(ApplicationTripSchema.AppMethod.UpdateTrip.getValue().getBytes(StandardCharsets.UTF_8));
args.addAll(tripArgs.getArgs(client));

// update application
Transaction txn = TransactionsHelper.noop_txn(client, appId, creator.getAddress(), args);
SignedTransaction signedTxn = creator.signTransaction(txn);

String txId = TransactionsHelper.sendTransaction(client, signedTxn);
PendingTransactionResponse response = TransactionsHelper.waitForConfirmation(client, txId);

LogHelper.log(this.getClass().getName(), String.format("Updated app-id: %s", response.applicationIndex));
return response.applicationIndex;
```

Figure 48: updateApplication method

In this snippet, the new information data of the trip are passed as arguments of the transaction.
Since we are calling the `update_trip` method of the stateful contract, we pass its name as argument in the `NoOp` transaction.
If no problem occurs, the trip is updated regularly.

8.3 Trip Deletion

When a user makes a request to delete a trip, a `Delete` transaction is performed from the Account to the Contract, that will be managed in the `application_start` method of the contract:

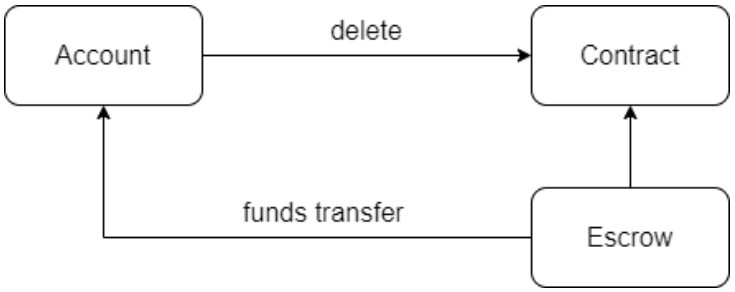


Figure 49: delete

The use case is implemented in the following function:
`deleteApplication(TripModel trip, Account creator)`

```
Long appId = trip.id();
Long amount = Long.valueOf(trip.getGlobalStateKey(ApplicationTripSchema.GlobalState.TripCost));

// delete the trip and perform a payment from the escrow
Transaction delete_txn = TransactionsHelper.delete_txn(client, creator.getAddress(), appId,  args: null);
Transaction payment_txn = TransactionsHelper.payment_txn(client, trip.escrowAddress(), creator.getAddress(),
        amount, creator.getAddress());

if(!trip.isEnded()) {
    //if trip is not already ended, close the escrow with the deletion
    // group transactions an assign ids
    Digest gid = TxGroup.computeGroupID(delete_txn, payment_txn);
    delete_txn.assignGroupID(gid);
    payment_txn.assignGroupID(gid);

    // sign individual transactions
    LogicsigSignature escrowSignature = getEscrowSignature(appId).get();
    SignedTransaction delete_signedTxn = creator.signTransaction(delete_txn);
    SignedTransaction payment_signedTxn = Account.signLogicsigTransaction(escrowSignature, payment_txn);

    // send transactions
    String txId = TransactionsHelper.sendTransaction(client, Arrays.asList(delete_signedTxn, payment_signedTxn));
    PendingTransactionResponse response = TransactionsHelper.waitForConfirmation(client, txId);
}
else {
    SignedTransaction delete_signedTxn = creator.signTransaction(delete_txn);
    String txId = TransactionsHelper.sendTransaction(client, delete_signedTxn);
    PendingTransactionResponse response = TransactionsHelper.waitForConfirmation(client, txId);
}

LogHelper.log(this.getClass().getName(), String.format("Deleted Trip for app-id: %s", appId));
return appId;
```

Figure 50: deleteApplication method

In this code snippet, there are two scenario:

1. the trip has already terminated and the escrow is already being closed.
2. the trip has not yet started.

In the first case, an atomic transaction will manage the escrow closure and the transfer of the remaining fundings to the creator with a `payment` transaction, all together with the delete transaction. The closing of the escrow is also made by passing the creator address as `closeReminderTo` parameter of the `payment` transaction.
In the second case only the delete transaction is done.
The completion of the delete operation will make the trip unavailable to the client application.

8.4 Trip Termination

When a user makes a request to terminate a trip, a `NoOp` transaction is performed from the Account to the Contract, that will be managed by the `start_trip` method of the contract:

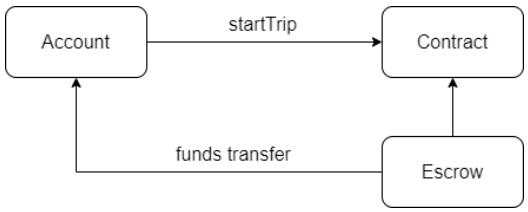


Figure 51: start_trip

This use case is implemented in the following function:
`startTrip(TripModel trip, Account creator)`

```
Long appId = trip.id();
Long amount = Long.valueOf(trip.getGlobalStateKey(ApplicationTripSchema.GlobalState.TripCost));

// participate to the trip and perform a payment from the escrow
List<byte[]> args  = new ArrayList<>();
args.add(ApplicationTripSchema.AppMethod.StartTrip.getValue().getBytes());

Transaction call_txn = TransactionsHelper.noop_txn(client, appId, creator.getAddress(), args);
Transaction payment_txn = TransactionsHelper.payment_txn(client, trip.escrowAddress(),
        creator.getAddress(), amount, creator.getAddress());

// group transactions an assign ids
Digest gid = TxGroup.computeGroupID(call_txn, payment_txn);
call_txn.assignGroupID(gid);
payment_txn.assignGroupID(gid);

// sign individual transactions
LogicsigSignature escrowSignature = getEscrowSignature(appId).get();
SignedTransaction call_signedTxn = creator.signTransaction(call_txn);
SignedTransaction payment_signedTxn = Account.signLogicsigTransaction(escrowSignature, payment_txn);

// send transactions
String txId = TransactionsHelper.sendTransaction(client, Arrays.asList(call_signedTxn, payment_signedTxn));
PendingTransactionResponse response = TransactionsHelper.waitForConfirmation(client, txId);

LogHelper.log(this.getClass().getName(), String.format("Started Trip for app-id: %s", appId));
return appId;
```

Figure 52: startTrip method

In this snippet of code, an atomic transaction with a `payment` transaction and a `NoOp` transaction is made. The `payment` transaction performs the fundings transfer from the escrow to the escrow creator. The closing of the escrow is also made by passing the creator address as `closeReminderTo` parameter of the `payment` transaction. The `NoOp` transaction will be managed by the stateful contract validating the request.
Since we are calling the `start_trip` method of the stateful contract, we pass its name as argument in the `NoOp` transaction.
After this transaction the dApp internal state will be FINISHED.

8.5 Trip Join

When a user makes a request to join a trip, an `OptIn` transaction will be performed, if the user has not yet opted-in.

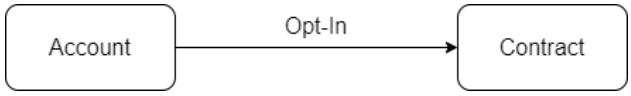


Figure 53: opt_in

After that, a `NoOp` transaction is performed from the Account to the Contract, that will be managed by the `participate_trip` method of the contract:

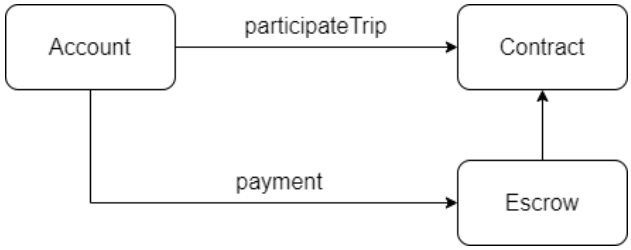


Figure 54: participate_trip

This use case is implemented in the following function:
`participate(TripModel trip, AccountModel account)`
We will split this method in two parts:

```
if(!trip.isValid()) {
    throw new Exception("The application program is not valid");
}
Long appId = trip.id();
Long amount = Long.valueOf(trip.getGlobalStateKey(ApplicationTripSchema.GlobalState.TripCost));

Account sender = account.getAccount();
ApplicationLocalState localState = account.getAppLocalState(appId);
if(localState == null) {
    // optin to trip if needed
    Transaction optin_txn = TransactionsHelper.optin_txn(client, sender.getAddress(), appId,  args: null);
    SignedTransaction optin_signedTxn = sender.signTransaction(optin_txn);

    String txId = TransactionsHelper.sendTransaction(client, optin_signedTxn);
    PendingTransactionResponse response = TransactionsHelper.waitForConfirmation(client, txId);
    LogHelper.log(this.getClass().getName(), String.format("Opted-in to app-id: %s", appId));
}
```

Figure 55: participate method part 1

This first snippet of code is responsible to check the local state of the current sender user, if the account has not opted into the contract yet, then a `OptIn` transaction is performed to access the local state of the contract.

```java
// participate to the trip and perform payment to escrow
List<byte[]> args  = new ArrayList<>();
args.add(ApplicationTripSchema.AppMethod.Participate.getValue().getBytes());

Transaction call_txn = TransactionsHelper.noop_txn(client, appId, sender.getAddress(), args);
Transaction payment_txn = TransactionsHelper.payment_txn(client, sender.getAddress(), trip.escrowAddress(),
        amount,  closeReminderTo: null);

// group transactions an assign ids
Digest gid = TxGroup.computeGroupID(call_txn, payment_txn);
call_txn.assignGroupID(gid);
payment_txn.assignGroupID(gid);

// sign individual transactions
SignedTransaction call_signedTxn = sender.signTransaction(call_txn);
SignedTransaction payment_signedTxn = sender.signTransaction(payment_txn);

// send transactions
String txId = TransactionsHelper.sendTransaction(client, Arrays.asList(call_signedTxn, payment_signedTxn));
PendingTransactionResponse response = TransactionsHelper.waitForConfirmation(client, txId);

LogHelper.log(this.getClass().getName(), String.format("Participated to app-id: %s", appId));
return appId;
```

Figure 56: participate method part 2

In this second code snippet, an atomic transaction with a `payment` transaction and a `NoOp` transaction is made. The `payment` transaction performs the payment for the trip cost from the sender to the escrow. The `NoOp` transaction will be managed by the stateful contract validating the request.
Since we are calling the `participate_trip` method of the stateful contract, we pass its name as argument in the `NoOp` transaction.
After this transaction the local state of the user will be set to PARTICIPATING.

8.6 Trip Leave

When a user makes a request to leave a trip, an `OptIn` transaction will be performed, if the user has not yet opted-in.

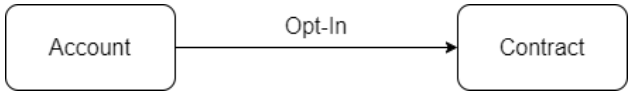


Figure 57: opt_in

After that, a `NoOp` transaction is performed from the Account to the Contract, that will be managed by the `cancel_participation` method of the contract:

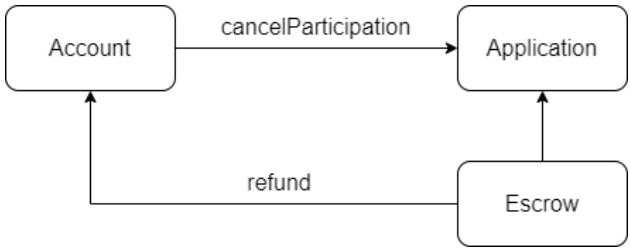


Figure 58: cancel_participation

This use case is implemented in the following function:
`cancelParticipation(TripModel trip, AccountModel account)`

```
// cancel participation to the trip and perform refund
List<byte[]> args  = new ArrayList<>();
args.add(ApplicationTripSchema.AppMethod.CancelParticipation.getValue().getBytes());

Transaction call_txn = TransactionsHelper.noop_txn(client, appId, sender.getAddress(), args);
Transaction payment_txn = TransactionsHelper.payment_txn(client, trip.escrowAddress(), sender.getAddress(),
        amount,  closeReminderTo: null);

// group transactions an assign ids
Digest gid = TxGroup.computeGroupID(call_txn, payment_txn);
call_txn.assignGroupID(gid);
payment_txn.assignGroupID(gid);

// sign individual transactions
LogicsigSignature escrowSignature = getEscrowSignature(appId).get();
SignedTransaction call_signedTxn = sender.signTransaction(call_txn);
SignedTransaction payment_signedTxn = Account.signLogicsigTransaction(escrowSignature, payment_txn);

// send transactions
String txId = TransactionsHelper.sendTransaction(client, Arrays.asList(call_signedTxn, payment_signedTxn));
PendingTransactionResponse response = TransactionsHelper.waitForConfirmation(client, txId);

LogHelper.log(this.getClass().getName(), String.format("Cancelled participation to app-id: %s", appId));
return appId;
```

Figure 59: cancelParticipation method

The optin behavior is the same as described on the join section, and comes before this snippet of code.

In this code snippet, an atomic transaction with a `payment` transaction and a `NoOp` transaction is made. The `payment` transaction performs the refund for the trip cost from the escrow to the sender. The `NoOp` transaction will be managed by the stateful contract validating the request. Since we are calling the `cancel_participation` method of the stateful contract, we pass its name as argument in the `NoOp` transaction. After this transaction the local state of the user will be set to NOT PARTICIPATING.

Conclusions

In this chapter we have seen an implementation overview of the client application. We have described the most important logic behind the client application, that is the section that manages the interaction with the dApp.

9 Security of the System

Introduction

In this section will be performed a small revision of the overall security of the system.

9.1 Algorand Blockchain

Since our application is a Smart Contract it will gain all the security benefits of a Blockchain. Cryptographic primitives and protocols, such as digital signatures and hash functions, support and safeguard the Blockchain system. These primitives ensure that transactions stored in the ledger are secured against tampering, as well as authenticated and non-repudiated. Furthermore, as a distributed system, in order for all users to agree on a single record, Blockchain technology requires a consensus process to be satisfied.

Moreover Algorand protects against threats on either the consensus protocol and the network level, all while ensuring the security of individual user accounts. A user's account must be online to participate in the consensus protocol.

In order for an online user to participate in the consensus protocol on Algorand, it has to generate and register a new secret **participation key** before going online. This key is required in order to reduce the exposure that would be created by using their **spending keys** directly. In this way, the user's stake is kept safe even if participation keys are tempered.

Algorand is also protected against partition attacks on the network. Ideally, an attacker could split the network into separate parts, allowing members of one area to communicate only with members of another area. In this kind of situation the network is completely asynchronous and the attacker has total control on the network traffic.

By this, on Algorand, the attacker is never able to persuade two honest users to accept two distinct blocks for the same round. This is verified even if the partition would last indefinitely and no one knows when it will be resolved. Moreover Algorand's chain never forks, ensuring that users' balances

are safe.

9.2 Payments

As we have already said, an Escrow is a legal arrangement in which a third party holds money until a certain criteria is satisfied. The escrow is managed by a stateless contract, and its usage will provide a layer of trustiness, so an external user will know that the trip creator will not directly receive the money until the trip is started, and he can claim for a refund whenever he wants.

Also the stateless contract will perform a validation on each atomic transaction containing a payment, that will be also validated by the stateful contract. This double check will guarantee the correctness of the payments, since the payments transactions are always made into an atomic transaction.

Payments transactions and the described validations can take place on the following actions:

- An escrow is funded.
- A trip is being terminated.
- A trip is being deleted before its termination.
- A user asks to join a trip.
- A user asks to leave a trip.

9.3 Contracts Validation

Since our application is made by the deployment of in instance of the dApp foreach trip, we need also to perform a validation to the stateful contract programs, in order to guarantee that the approval program and the clear state program are as expected.

This check is made by the client retrieving the hash of the programs from the blockchain and comparing it to the expected one.
The trip validation is performed by the following method, that can be found into the file `TripModel.java`:

```
public boolean isValid() {
    String approvalProgram = this.application.params.approvalProgram();
    String clearStateProgram = this.application.params.clearStateProgram();

    if(!approvalProgram.equals(ApplicationConstants.approvalProgramHash)) {
        return false;
    }
    if(!clearStateProgram.equals(ApplicationConstants.clearStateProgramHash)) {
        return false;
    }
    return true;
}
```

Figure 60: isValid method

The validation is performed for each trip retrieved from the blockchain, discarding the invalid ones, and also is performed before the trip join and the trip leave actions, to prevent unwanted payments to fraudulent contracts.

Conclusions

In this chapter we have demonstrated some security issues related to our application and the blockchain and how to correctly manage them.

10 Results

Introduction

In this section we will see the final result of the client application.
Some example of the working Android client application can be seen on the official github repository of the project:
`https://github.com/bara96/algo-carsharing-android#android-application`

10.1 Home

This is the home section that contains a list with all the valid retrieved trips, displaying some information of each trip. In this list the user's created trips are excluded.
By clicking on a row a new activity will start displaying the trip information.

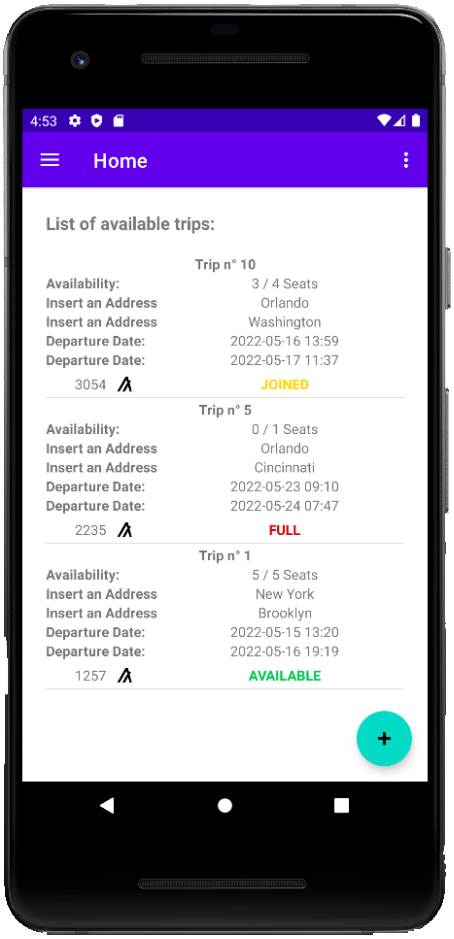


Figure 61: Home view

10.2 Created Trips List

This section contains a list with all the valid trips created by the user, displaying some information of each trip.
By clicking on a row a new activity will start displaying the trip information.

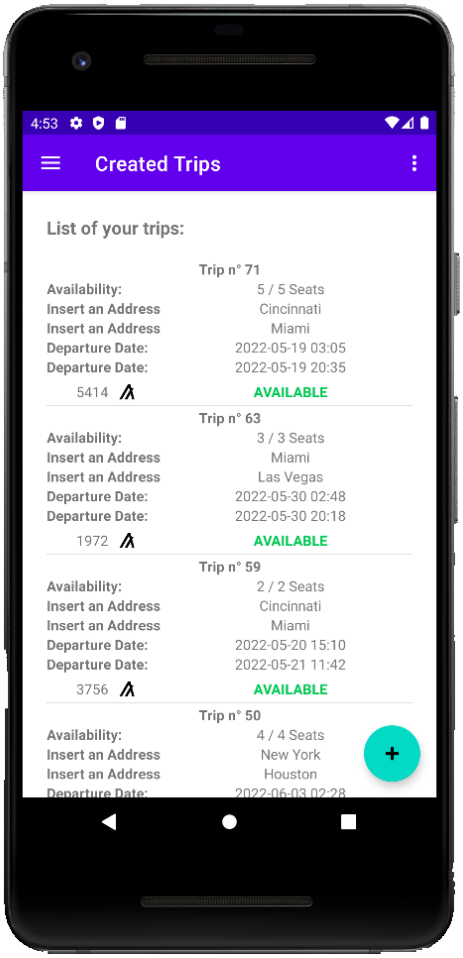


Figure 62: Created trips list view

10.3 Joined Trips List

This section contains a list with all the valid trips the user has joined, displaying some information of each trip.
By clicking on a row a new activity will start displaying the trip information.

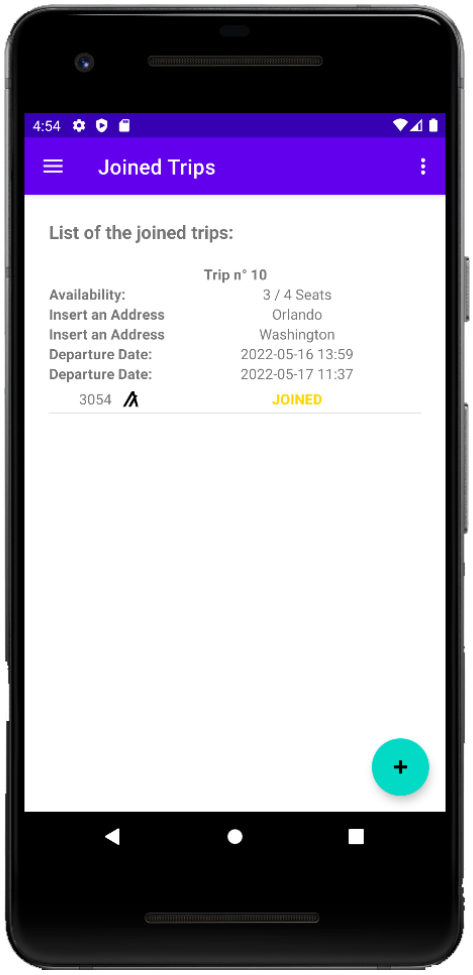


Figure 63: Joined trips list view

10.4 Create Trip

This section is reachable by pressing the floating action button with the "*plus*" symbol. It contains a form to allow the insertion of data for a new trip. Also a "*test*" button is present to easily create a trip with some random factory information.

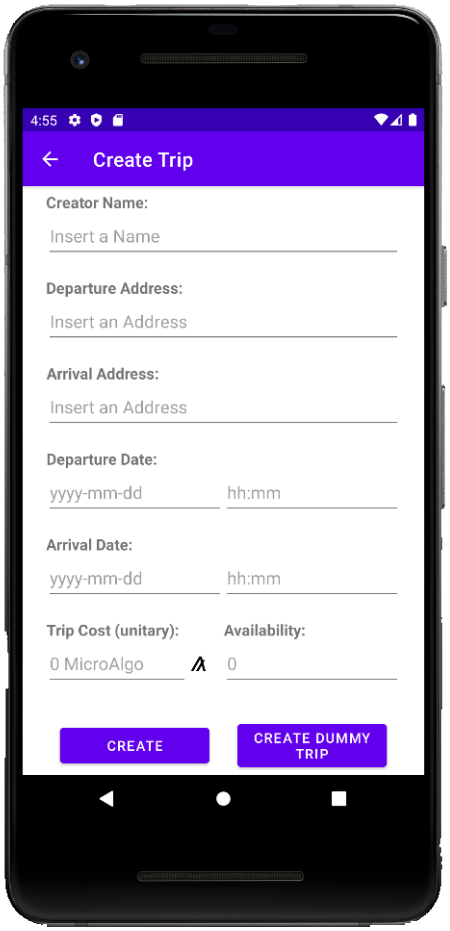


Figure 64: Create trip view

10.5 Update Trip

This section is reachable by pressing a valid trip created by the user from a list. It contains a form that can be enabled or disabled depending on the situation. The form is enabled if the user can update the trip, otherwise it will be disabled. Also the action button may differ depending on the situation. In this case, the screen refers to a valid "*trip update*" situation. From this section is also possible the deletion of the trip.

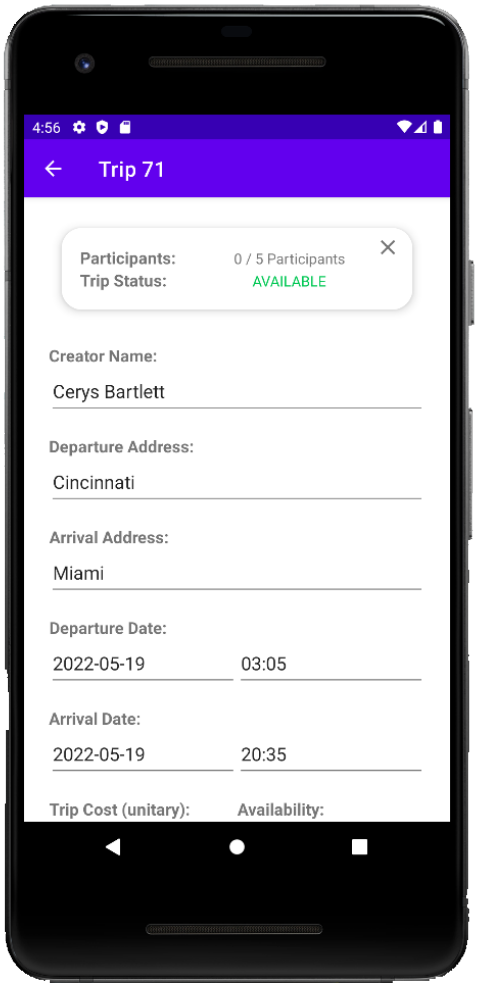
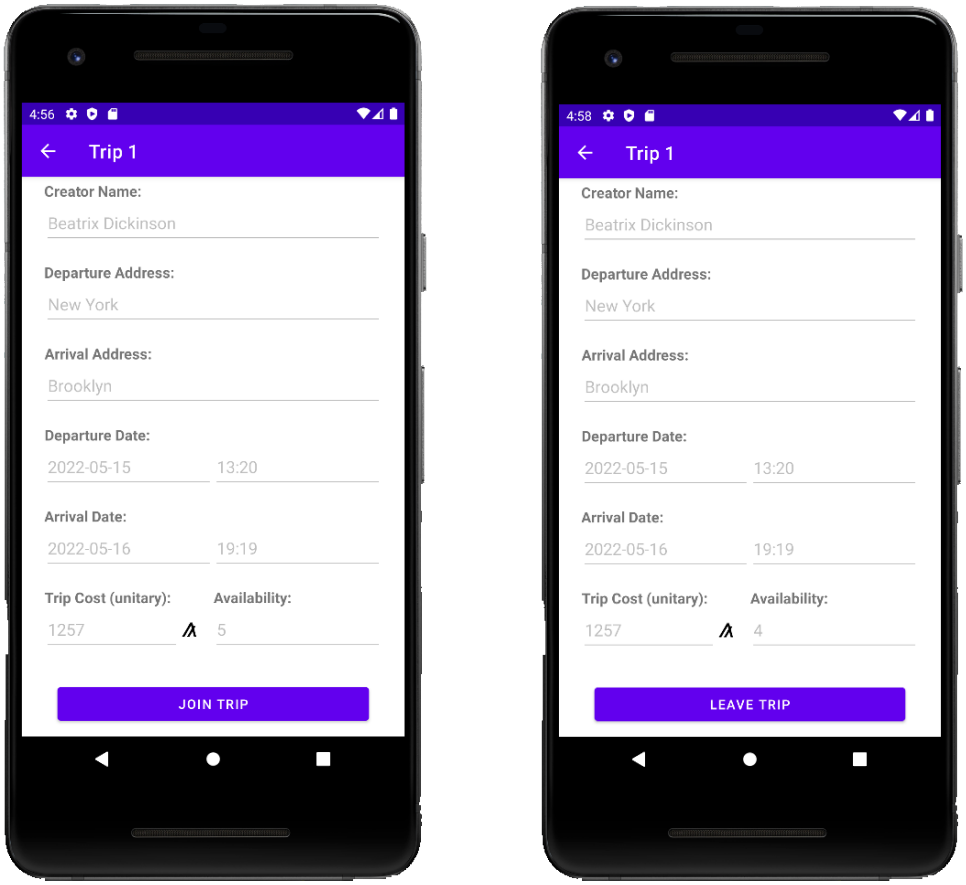


Figure 65: Update trip view

10.6 Join / Leave Trip

This section is reachable by pressing a trip, that is not created by the user, from a list. It contains a form that is disabled and the action button can be the "*join trip*" action or the "*leave trip*" action, depending on the user state.



(a) Join trip view (b) Leave trip view

Figure 66: Join / Leave trip view

10.7 Terminate Trip

This section is reachable by pressing a trip created by the user from a list. Also the departure time must have passed to reach this section.
It contains a form that is disabled and the action button "*end trip*".

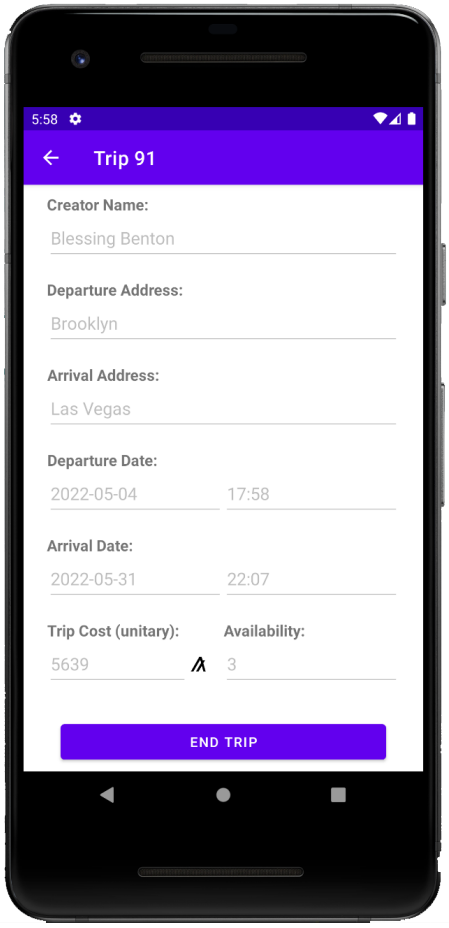


Figure 67: Terminate trip view

10.8 Account

This section contains a form to insert an Account mnemonic, required to perform any action inside the app. Also some user information is retrieved and displayed to the user.

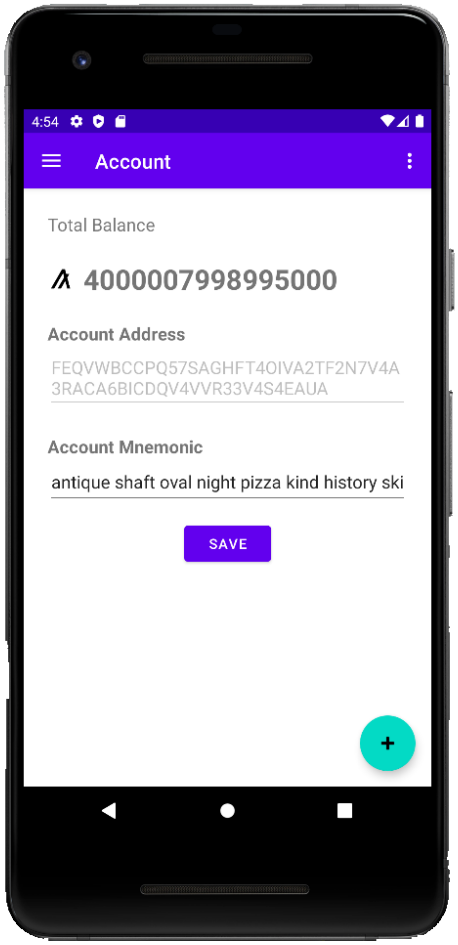


Figure 68: Account view

Conclusions

In this chapter we have seen the final result of the Android application. The application is simple and user friendly, and contains all the previously described use cases.

11 Future Work

Introduction

In this section we will see some other essential features that could be implemented or taken into consideration on a future work.

11.1 Account Management

As stated before on the implementation section, the account management is extremely simplified, and it is not ideal for a real environment usage.
The best practice should be to integrate the account management of a trusted hardware wallet, such as perawallet, this would increase the security of the local user accounts as well as delegating the responsibility to the wallet.

11.2 Contract Verification

With the current logic, each trip instance will be deployed as a different new dApp, and each one will be verified by the client. A different idea could be to create a single big smart contract in which each trip is represented by a new account that is rekeyed [9] to the application account and records trip information.
In this context we have two types of accounts with local storage:

1. the trips that are rekeyed to the application account that would be used instead of global storage.

2. the current local storage account we are using.

Having only one big dApp will remove the necessity of verifying the contract programs, since this application will be deployed only by a trusted user.

11.3 Feedback

The general workflow of the most used carpooling services usually provides a way to give feedback to the trip experience. This feature is essential in order to state a rating-based ranking and let the other users know the affidability of a content creator.
Also if we focus our attention into more famous and used services, also unrelated to carpooling, like booking.com, we find out that the feedback is essential for this kind of applications where you haven not the complete assurance on the provided data and you have to trust the other party.
Giving the possibility to the users to provide feedback should be also the next step of our application, this would make the users more comfortable in the way they choose their trips and could discourage untrusted users.

Conclusions

In this chapter we have seen some other interesting features that should be taken into consideration in the case that the application will be deployed for public access.

12 Conclusions

In this study, we have seen the importance that a decentralized approach on a carpooling service can provide.
Using a blockchain will solve some issues like trust, fault tolerance and other security problems that a centralized authority cannot totally grant. In particular, Algorand blockchain with its Smart Contracts allows the development of a decentralized application that will take advantage of the strengths of its consensus protocol, that aims to solve the blockchain trilemma.
Also, we have to consider the cheap fees that Algorand requires, that compared to the Ethereum gas fees are very small. The development of a Smart Contract is done with TEAL language, that can be compiled with an SDK and next deployed to the blockchain in order to be used by a client application.
We have seen how to correctly manage the transactions, with the usage of an escrow for the payments to guarantee more security to the system. The final result is a fully working application that is composed of an Android client application that interacts with the Algorand blockchain using the Smart Contracts as primary storage to correctly manage carpooling trips.
Furthermore, I would like to emphasize that despite the final obtained result, a lot of problems have been solved during the development period. Some of the hardest issues to solve were caused by the limits imposed by Algorand Smart Contracts, either in the context of data storage and also for the topic regarding the contract interaction with a client. Both these problems had been hardly studied and finally solved or mitigated by strategies deployed also thanks to the Algorand community suggestions.
I want to underline that developing on a Blockchain is way more different that building a client-server application and requires a way more distinct approach.
With this thesis we have demonstrated for the first time that creating a decentralized Carpooling application can be done and will work efficiently and securely. Although the developed project is not yet ready for a wide adoption by the public, it represents a good starting point on a design of a real decentralized application that will aim to solve trust, costs and other issues related to the centralized authority.

13 Acknowledgements

I wish to show my appreciation to my supervisor, prof. Marin Andrea, who guided me throughout this study, for his help and the availability provided to me throughout the project development.

Further acknowledge for the help given by the Algorand community staff, who provided a number of helpful comments and suggestions.

I would also like to thank my parents for their wise advice and their ability to listen to me.

Finally, I thank my friends for the support they have given me in recent months.

A heartfelt thanks to everyone.

References

[1] Algorand. "Algorand consensus - Algorand Developer Portal". In: (2022). URL: https://developer.algorand.org/docs/get-details/algorand_consensus/ (visited on 06/04/2022).

[2] Algorand. "Algorand Sandbox". In: (2022). URL: https://github.com/algorand/sandbox (visited on 06/02/2022).

[3] Algorand. "Algorand Testnet Dispenser". In: (2022). URL: https://dispenser.testnet.aws.algodev.network/ (visited on 06/02/2022).

[4] Algorand. "Atomic transfers - Algorand Developer Portal". In: (2022). URL: https://developer.algorand.org/docs/get-details/atomic_transfers/ (visited on 06/04/2022).

[5] Algorand. "goal - Algorand Developer Portal". In: (2022). URL: https://developer.algorand.org/docs/clis/goal/goal/ (visited on 06/04/2022).

[6] Algorand. "Indexer - Algorand Developer Portal". In: (2022). URL: https://developer.algorand.org/docs/get-details/indexer/ (visited on 06/04/2022).

[7] Algorand. "Modes of use - Algorand Developer Portal". In: (2022). URL: https://developer.algorand.org/docs/get-details/dapps/smart-contracts/smartsigs/modes/ (visited on 06/04/2022).

[8] Algorand. "PyTeal: Algorand Smart Contracts in Python". In: (2022). URL: https://pyteal.readthedocs.io/en/stable/ (visited on 06/03/2022).

[9] Algorand. "Rekeying - Algorand Developer Portal". In: (2022). URL: https://developer.algorand.org/docs/get-details/accounts/rekey/ (visited on 06/04/2022).

[10] Algorand. "Sending a Transaction in the Future - Algorand Developer Portal". In: (2022). URL: https://developer.algorand.org/docs/get-details/transactions/#sending-a-transaction-in-the-future (visited on 06/04/2022).

[11] Algorand. "State Operation Table". In: (2022). URL: https://pyteal.readthedocs.io/en/stable/state.html#state-operation-table (visited on 06/03/2022).

[12] Algorand. "The lifecycle of a smart contract - Algorand Developer Portal". In: (2022). URL: https://developer.algorand.org/docs/get-details/dapps/smart-contracts/apps/#the-lifecycle-of-a-smart-contract (visited on 06/04/2022).

[13] Docker. "Docker". In: (2022). URL: https://www.docker.com/ (visited on 06/01/2022).

[14] Ethereum. "Etherscan". In: (2022). URL: https://etherscan.io/ (visited on 06/02/2022).

[15] Google. "Documentation for app developers". In: (2022). URL: https://developer.android.com/docs (visited on 06/01/2022).

[16] Huaqun Guo and Xingjie Yu. "A survey on blockchain technology and its security". In: *Blockchain: Research and Applications* 3.2 (2022), p. 100067. DOI: 10.1016/j.bcra.2022.100067. URL: https://www.sciencedirect.com/science/article/pii/S2096720922000070.

[17] Viktor Valaštín; Kristian Kostal; Rastislav Bencel; Ivan Kotuliak. "Blockchain Based Car-Sharing Platform". In: *2019 International Symposium ELMAR* (2019), pp. 5–8. DOI: 10.1109/ELMAR.2019.8918650. URL: https://ieeexplore.ieee.org/document/8918650.

[18] Min Xu; Xingtong Chen; Gang Kou. "A systematic review of blockchain". In: *Financial Innovation* 5.1 (2019). DOI: 10.1186/s40854-019-0147-z. URL: https://jfin-swufe.springeropen.com/articles/10.1186/s40854-019-0147-z.

[19] Mayank Raikwar; Danilo Gligoroski; Katina Kralevska. "SoK of Used Cryptography in Blockchain". In: *IEEE Access* 7 (2019), pp. 148550–148575. DOI: 10.1109/ACCESS.2019.2946983. URL: https://ieeexplore.ieee.org/document/8865045.

[20] Pera Wallet LDA. "perawallet". In: (2022). URL: https://perawallet.app/ (visited on 06/02/2022).

[21] William Metcalfe. "Ethereum, Smart Contracts, DApps". In: *Blockchain and Crypto Currency: Building a High Quality Marketplace for Crypto Data* (2020), pp. 77–93. DOI: 10.1007/978-981-15-3376-1_5. URL: https://link.springer.com/chapter/10.1007/978-981-15-3376-1_5.

[22] Jing Chen; Silvio Micali. "Algorand Theoretical Paper". In: *CoRR* abs/1607.01341 (2016). DOI: 10.48550/arXiv.1607.01341. URL: https://arxiv.org/abs/1607.01341.

[23] Silvio Micali. "Algorand 2021 Performance". In: (2020). URL: https://www.algorand.com/resources/algorand-announcements/algorand-2021-performance (visited on 06/01/2022).

[24] Microsoft. "WSL". In: (2022). URL: https://docs.microsoft.com/en-us/windows/wsl/install (visited on 06/01/2022).

[25] Satoshi Nakamoto. "Bitcoin: A Peer-to-Peer Electronic Cash System". In: (2009). URL: https://bitcoin.org/bitcoin.pdf.

[26] PostgreSQL. "PostgreSQL". In: (2022). URL: https://www.postgresql.org/ (visited on 06/01/2022).

[27] B. Sriman; S. Ganesh Kumar; Shamili Prabakaran. "Blockchain Technology: Consensus Protocol Proof of Work and Proof of Stake". In: *Intelligent Computing and Applications* (2021), pp. 395–406. DOI: 10.1007/978-981-15-5566-4_34. URL: https://link.springer.com/chapter/10.1007/978-981-15-5566-4_34.

[28] Simonetta Balsamo; Andrea Marin; Isi Mitrani; Nicola Rebagliati. "Prediction of the Consolidation Delay in Blockchain-based Applications". In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering* (2021), pp. 81–92. DOI: 10.1145/3427921.3450249. URL: https://dl.acm.org/doi/10.1145/3427921.3450249.

[29] Aqeel Khalique; Singh Kuldip; Sandeep Sood. "Implementation of Elliptic Curve Digital Signature Algorithm". In: *International Journal of Computer Applications* 2.2 (2010), pp. 21–27. DOI: 10.5120/631-876. URL: https://www.ijcaonline.org/volume2/number2/pxc387876.pdf.

[30] Jing Chen; Sergey Gorbunov; Silvio Micali; Georgios Vlachos. "ALGORAND AGREEMENT: Super Fast and Partition Resilient Byzantine Agreement". In: *Cryptology ePrint Archive, Report 2018/377* (2018). URL: https://ia.cr/2018/377.

[31] Weizhi Meng; Jianfeng Wang; Xianmin Wang. "Position Paper on Blockchain Technology: Smart Contract and Applications". In: *Network and System Security* (2018), pp. 474–483. DOI: 10.1007/978-3-030-02744-5_35. URL: https://link.springer.com/chapter/10.1007/978-3-030-02744-5_35.

[32] Xinyu Wang. "Research on ECDSA-Based Signature Algorithm in Blockchain". In: *Finance and Market* 4.2 (2019), pp. 55–58. DOI: 10.18686/fm.v4i2.1600. URL: https://ojs.usp-pl.com/index.php/fm/article/view/1600.

[33] Manu Drijvers; Sergey Gorbunov; Gregory Neven; Hoeteck Wee. "Pixel: Multi-signatures for Consensus". In: *Cryptology ePrint Archive, Report 2019/514* (2019). URL: https://ia.cr/2019/514.

[34] Bitcoin Wiki. "Secp256k1". In: (2019). URL: https://en.bitcoin.it/w/index.php?title=Secp256k1&oldid=66342 (visited on 06/01/2022).

[35] Derek Leung; Adam Suhl; Yossi Gilad; Nickolai Zeldovich. "Vault: Fast Bootstrapping for the Algorand Cryptocurrency". In: *Cryptology ePrint Archive, Report 2018/269* (2018). URL: https://ia.cr/2018/269.

[36] Yossi Gilad; Rotem Hemo; Silvio Micali; Georgios Vlachos; Nickolai Zeldovich. "Algorand: Scaling Byzantine Agreements for Cryptocurrencies". In: *Proceedings of the 26th Symposium on Operating Systems Principles* (2018), pp. 51–68. DOI: 10.1145/3132747.3132757. URL: https://dl.acm.org/doi/10.1145/3132747.3132757.