Tokenizing Transportation: Blockchain Solutions for MaaS Ticketing

Supervisors

Prof. Danilo BAZZANELLA

Dott. Davide GAROFALO

Dott. Marco BAZZANI

Candidate

Federico PASQUALINI

December 2023

Abstract

In a world of fast-growing cities and the need for easier, eco-friendly travel, Mobility as a Service (MaaS) is a promising solution. MaaS combines different types of transportation like buses, taxis, and bikes into one easy platform for planning and payment. At the current state, MaaS implementations face several problems like fragmented territorial systems and privacy concerns between Mobility Service Providers.

In this thesis, I propose the design and implementation of a possible MaaS infrastructure that tries to solve those problems with the use of blockchain and smart contracts. I created a standard for representing tickets and I was able to code smart contracts to handle this standard and to allow buying and validating tickets. On top of that, I also built the back end in NodeJs and the front end of a mobile application in React Native. The app is able to autonomously fetch data from the Mobility Service Providers and to buy tickets through the blockchain by only knowing the address of the main smart contract. All payments are handled through the blockchain as well.

While in this thesis I focused on coding the contracts for the Ethereum blockchain, the work could be extended by building a completely private and personalised blockchain where those contracts can run more efficiently from the point of view of costs and privacy.

Table of Contents

List of Tables

List of Figures

Listings

Glossary

MSP Mobility Service Provider is the company that offers transportation solutions.

MaaS Mobility as a Service integrates various transportation services into a single platform.

ECDSA Elliptic Curve Digital Signature Algorithm is an algorithm for creating digital signatures.

DApp Decentralized Application is a software application that runs on a decentralised network utilising blockchain technology.

Web3 Web3 is the next generation of the internet, emphasizing decentralization, blockchain, and user empowerment.

PoW Proof of Work, a consensus algorithm.

PoS Proof of Stake, a consensus algorithm.

DPoS Delegated Proof of Stake, a consensus algorithm.

EVM Ethereum Virtual Machine is a stack-based Turing-complete virtual machine.

ERC20 Ethereum Request for Comment 20 is a standard for creating fungible tokens on the Ethereum blockchain.

API Application Programming Interface are defined rules for two or more programs to communicate.

Chapter 1

Introduction

Public transport is one of the backbones of mobility in a city. With growing concerns about climate change and pollution, moving to sustainable and eco-friendly transportation seems one of the only solutions to reduce our carbon footprint and make cities more livable. One possible solution to deal with the complex problem of today's transport is Mobility as a Service, known as MaaS for short. Mobility as a Service aims to aggregate all the different kinds of transportation in a unique platform, making it easier for the end user to plan, book, and use public transport. This platform will enable the user to bring together all types of transportation like buses, taxis, electric scooter rentals, and more. When looking deeper into the world of Mobility as a Service, it becomes clear that there are hurdles that need to be overcome for it to work smoothly.

Firstly, there are gaps in using digital methods for buying tickets at the local, regional, and national levels. MaaS envisions a world where you can easily plan, book, and pay for your entire journey in one go. However, the reality is different in various places. In some areas, you can conveniently purchase electronic tickets using your smartphone, but in other places, traditional paper tickets are still the norm. This difference can make MaaS confusing and less useful, especially when your journey involves multiple regions or cities.

Secondly, there is the challenge of sharing information among the companies that provide transportation services. When different companies need to work together, they might be hesitant to share their data. This hesitation can be due to concerns about privacy, competition, or following the rules.

The objective of this thesis is to implement a full-stack Mobility as a Service (MaaS) application that addresses the existing challenges by leveraging blockchain technology. The focus is to create a decentralised platform where Mobility Service Providers (MSPs) can register and seamlessly integrate their services. This will enable users to conveniently book various mobility options through a unified application, thus simplifying the user experience in accessing transportation services.

Chapter 2

Background

In this chapter, there will be an overview of the theoretical knowledge needed to understand the thesis-related work. It will be covered how blockchains work focusing mainly on the cryptography behind them and the consensus that allows them to remain decentralised. Bitcoin and Ethereum cryptocurrencies will be prioritised as they are the most known and innovative projects in the ecosystem. There will also be a more in-depth overview regarding Ethereum as it is fundamental that it is discussed what the Ethereum Virtual Machine and smart contracts are. At the end, there will be a short introduction to Mobility as a Service (MaaS) and what its purpose is.

2.1 Blockchain

Before the introduction of blockchain, using a trusted third party was nearly the only way to send money between two parties or complete any other type of financial transaction. The third party had the job to make the transaction in a reliable and safe way but also keep track of the balances for every account. This system has multiple flaws as it relies on trust. When you make use of a trusted financial institution, you take several risks. For example, the third party can be hacked. This can determine in the best case a delay in the transaction or worse a loss of data or funds. On top of that, the financial service provider can also become rogue and fraud your money altogether. The third party is the single point of failure [1]. On top of that, using a traditional institution increases transaction costs and usually limits the maximum amount that can be transferred on a single transaction [2].

The first proposal to solve those issues was made by Satoshi Nakamoto when he published the white paper for Bitcoin in 2008. Bitcoin is a trustless distributed ledger that allows, with the use of cryptographic functions, to transfer value between peers. Bitcoin removes the need for trust. The cryptography and the consensus system guarantee that the transactions are legit and the latter can be inspected by anyone to verify their trustworthiness.

2.2 How does a blockchain work?

As previously said, a blockchain is a public ledger.

> "In accounting, a ledger is a book or database for recording transactions such as debits and credits in an account." [3]

Every user can generate one (or multiple) accounts that are identified by an alphanumeric string called address (more details on how the address is generated later).

$$bc1qxy2kgdygjrsqtzq2n0yrf2493p83kkfjhx0wlh$$

Example of Bitcoin address

Every user can send funds to another user by adding a record into the ledger specifying the amount and the address of the receiver. The ledger, therefore, records all the payments ever made between addresses and all these payments can be viewed by anyone. For every address, a balance can be associated. The balance can be calculated as the sum of all the payments received minus the sum of the payments made. It can therefore be enforced that a user can make a transfer only if they have enough balance associated with their address. In order to guarantee that only the owner of the address can send payments from their own funds, a mechanism is required to authenticate the transaction. This is accomplished by using digital signatures.

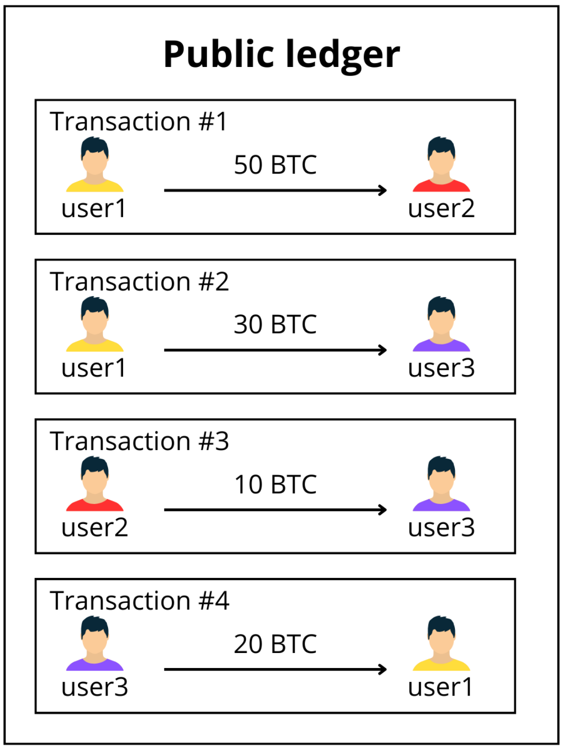


Figure 2.1: Public ledger

2.2.1 Digital signatures

Digital signatures use asymmetric cryptography to validate the authenticity and integrity of a message [4]. To achieve this goal, everyone has to generate a private key/public key pair. Those two keys, in asymmetric cryptography, are mathematically linked and it should be easy to go from the private key to the public key but computationally impossible to go the other way around. Functions with this property are usually called trapdoor functions [5].

Factoring has been used for a long time as trapdoor functions (for example RSA). A more modern approach for creating keypairs is using elliptic curves.

Various types of elliptic curves are used for cryptographic purposes like, for example, secp256k1. In general, they are all defined by an equation that is similar to

$$y^2 = x^3 + ax + b$$

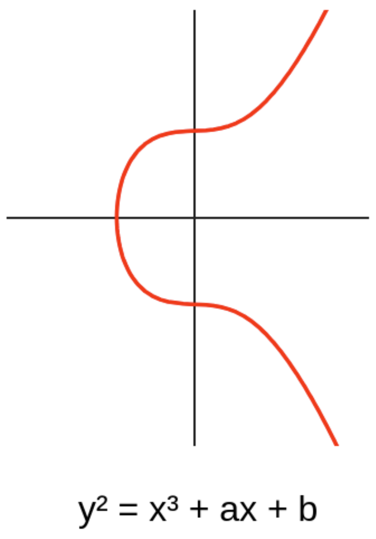


Figure 2.2: Elliptic curve example

In image 2.2, it is shown what a graphed elliptic curve looks like. This curve has 2 important properties. First, the graph is horizontally symmetrical. Any point in the positive y-axis has a corresponding point in the curve on the negative y-axis. Second, any line that is not vertical will intercept the curve at most 3 points [5].

Given a specific curve, the addition operation can be defined. The best way to explain how the addition operation is defined is by showing it graphically.

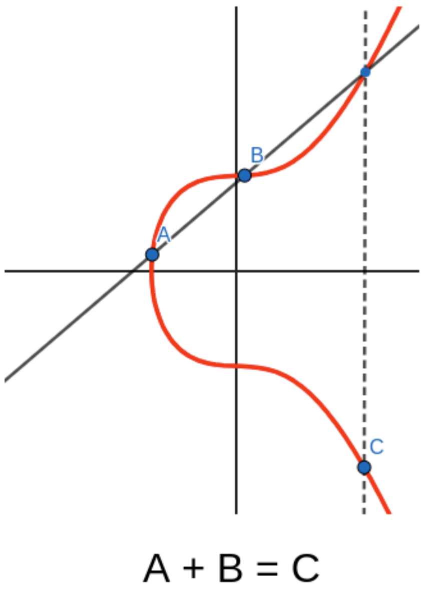


Figure 2.3: Elliptic curve addition example

The addition operation is defined on two points that are part of the curve. Given those two points A and B, a straight line can be drawn passing through them. This line, for the second property of the elliptic curve, will intercept the curve in a third point D. By negating the y of D, the result is the point C. For the first property of the elliptic curves, the point C is still part of the curve. C is the result of the sum of A and B.

From the addition operation, the multiplication with a scalar can be easily defined.

$$n * A = B$$

as it is equivalent to summing the point with himself for n times.

Given this, the private key is defined as an integer k, big enough to not be easily guessed.

For every curve, a generator G is defined such that you can generate any other point of the curve by multiplying it by some integer n [6].

The public key can be calculated by multiplying the generator G with the private key

$$Publickey = privatekey * G$$

From this, the trapdoor property is pretty straightforward. If you know the private key, the public key can be easily computed thanks to the double and add algorithm which has a speed of $\mathcal{O}(\log n)$ [5].

On the other hand, calculating the private key given only the public key and the generator n requires doing

$$Privatekey = publickey/G$$

which is extremely slow. The only best way to compute the private key is to bruteforce which has a speed of $\mathcal{O}(n)$.

Once a key pair is generated, this can be used to sign and validate messages. The private key can be used to generate a signature of the message as follows:

$$Signature = sign(message, privatekey)$$

The signature depends not only on the private key but also on the message. This means that the signature depends entirely on the message and it is valid only for the specific message and private key.

A signature can be validated using the public key associated with the private key as follows:

$$verify(message, signature, publickey) = true/false$$

By using digital signatures, peers can sign transactions with the private key (that is supposed to be kept secret) to prove they are indeed authors of the transactions and that they are authorising spending their cryptocurrencies. To make transactions and signatures unique, and therefore avoid replay attacks where someone rebroadcasts someone else's transaction for a second time, a nonce is added to the transactions. A nonce is just an integer added to the transaction payload that is different for every transaction. Usually, it is enforced for the nonce to be sequential for each transaction sent by an address. This means that the first transaction sent by the address A will be 0 and then it will be incremented by one for any consecutive transaction. This makes it impossible for an attacker to replay transactions twice as the second transaction would require a different nonce and therefore a different signature that only the owner of the account can produce.

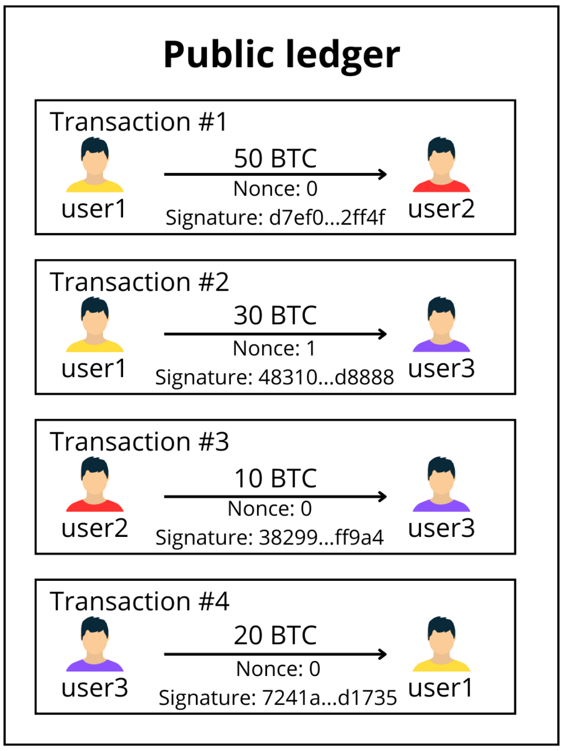


Figure 2.4: Public ledger with signatures and nonce

2.3 Bitcoin

In order to preserve the integrity and availability of the ledger, this has to be distributed on multiple nodes. By making the ledger distributed, the problem of how to make sure that all the ledgers are synced with each other arises. A mechanism is therefore needed to make sure that every ledger has all transactions and those transactions are in the same order. Consensus is defined as the protocol used to achieve distributed agreement about the ledger's state. There are multiple ways to achieve consensus. The most famous algorithms are Proof of Work (PoW), Proof of Stake (PoS), and Delegated Proof of Stake (DPoS).

Bitcoin uses a consensus mechanism known as Proof of Work, PoW for short. All accounts are represented by an address. The address can be considered as the equivalent of the IBAN in the traditional banking system. It is used to identify the account where to send Bitcoin. The address is usually made of alphanumeric characters and it is connected to the public key. The address is derived by hashing the public key.

2.3.1 Hash

A hashing algorithm is a function that takes a message as an input and gives a random fixed-length string of bits as an output. A hashing algorithm has a few important properties. First, it is computationally impossible to reverse. Meaning that given the hash, it is impossible to tell what was the original message. On top of that, a slight change in the input should result in a completely random change in the output.

$$hash(message) = ab530a13e45914982b79f9...1cea1afbf02b460c6d1d$$
$$hash(message1) = 97d035e32036a670058f2b...da6f2af342db4a968e99$$

There are many types of hashing algorithms available. The most used in the blockchain space are sha256, part of the sha-2 family and mostly used by Bitcoin which returns a 256 bits hash, and keccak-256, part of the sha-3 family and used mainly by Ethereum which returns 256 bits hash as well.

2.3.2 Proof of Work

A blockchain, as the name suggests, is divided into blocks. A block is composed of a list of transactions and it has its own ID, which is computed by hashing its content. Every block is linked to the previous one by inserting the id of the previous block in the current block payload. This link makes it impossible to change one single block without having to change all the blocks after the modified one. For example, if a transaction in block number 5 is modified, the id, that is the hash of all the block payload, will change. If the id changes, an edit to the following block (number 6) is needed as it contains the id of the previous block that is now different. But if the previous block ID on block 6 changes, the block 6 ID will also change as the previous block ID is part of the block payload. If block 6 ID has changed, it means that block 7 will need to be edited as well and so on until the last block is reached.

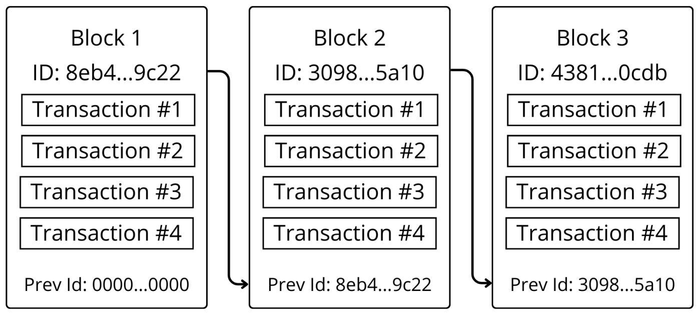


Figure 2.5: Blockchain blocks

With PoW consensus, the nodes will trust the chain of blocks that has the most computational power put into it (that's why it is called proof of work). The way it works is that for one block to be added to the blockchain, a mathematical problem has to be solved. The problem consists of adding a nonce to the block payload such that the id of such a block (that is the hash of the whole block) starts with N number of zeros. The number N is called difficulty and it increases or decreases in such a way as to keep the average block time around 10 minutes. When new computational power enters the network, the mathematical problem will be solved faster and therefore the network automatically adjusts by increasing the difficulty to bring back the average time closer to 10 minutes. All the people involved in trying to solve the mathematical problem are called miners. Once the nonce has been found, the block is submitted to the network and other peers can easily verify the validity of the block by just hashing the payload and checking that the first N bits of the id are zeros. If someone wanted to change even slightly the content of the block, they would have to go through all the work to rebuild a valid one. The same applies if you want to change any previous block as it would invalidate any other consecutive block (as they are linked by the id) and therefore it would require all the work to rebuild the chain from scratch.

So, as it has been said, the network is programmed to follow the longer chain which is the one with more computational power. It can happen that two blocks with the same transactions and different IDs are broadcasted at the same time by two different miners. This creates a fork on the blockchain where there is a branch with two viable options and they are both valid as they are the same length. In this case, the conflict is solved by just waiting. Every miner will focus on creating new blocks on either one of the forks and only one fork will have more than 50% of the computational power and therefore will be eventually longer than the other, solving the fork. This is the reason why a transaction is considered confirmed only after the block in which it is included has been followed by at least 3-4 other blocks. As long as one miner doesn't have more than 50% of the total computational power of the network, it is impossible for a miner to create a fork by sending fraudulent blocks to only a few selected peers. Eventually, it will be impossible for the miner to keep up with the computational power of the rest of the network to keep their fork longer and it will inevitably be invalidated in favour of the network honest chain.

Miners are incentivized to run the network in two different ways. Transactions can include fees to be paid to miners to be included in a block. Since the maximum number of transactions that can be included in a block is limited, the higher the fee, the more likely it is for the transaction to be included in the block. Fees tend to increase when the network becomes congested as there is more competition to be included and decrease when blocks are not completely full. Miners can also include on a block a special transaction, not signed by anyone, that gives them a precise amount of cryptocurrency. This is called block reward. The amount of the block reward is defined by the protocol. With Bitcoin, the network started with 50 BTC block rewards and it is programmed to be reduced in half every 210.000 blocks, which corresponds to around 4 years. The block rewards get halved up to around the year 2140, where the block reward will become 0. This sets the maximum amount of BTC to be ever produced to 21M. The hard cap of Bitcoin supply represents one of its key features. It makes Bitcoin a deflationary and scarce currency by design and it is considered by many as a digital store of value for this reason.

2.4 Ethereum

Ethereum is a decentralised, open-source blockchain platform that has gained significant attention and popularity since its launch in 2015. It was created by a young programmer named Vitalik Buterin and has since become one of the leading platforms for developing and deploying smart contracts and decentralised applications (DApps). It was born as a PoW blockchain, just like Bitcoin, but it recently changed the consensus to PoS.

2.4.1 Proof of Stake

With Proof of Stake, the actors that validate and create new blocks are called validators. In PoS, in order to become a validator, you have to lock (or stake) a specific amount of cryptocurrency. In the Ethereum case, 32 ETH are required to become a validator. Validators are responsible for creating new blocks and verifying that blocks produced by other validators are valid. The process of validation is called attestation. If one validator decides to behave maliciously by sending conflicting attestations or proposing multiple blocks, they can be punished by the rest of the network by destroying a part or all of their staked cryptocurrency. Unlike PoW, where the timing of blocks is determined by the difficulty, the timing in Ethereum is fixed in slots of 12 seconds. The validator that is supposed to create the next block is selected randomly with a deterministic approach so that everyone can agree on which one is selected [7].

So, as long as an attacker doesn't hold more than 50% of the staked ETH, it will be impossible to tamper with the blockchain as the rest of the network would detect it and destroy their stake.

A transaction can be considered final when it is part of a block that can't be changed without burning a huge amount of the staked cryptocurrency. In Ethereum, this huge amount is defined as one-third of the total staked ETH [7].

PoS has a few advantages when compared to PoW. It is for sure more energy efficient. A validator can be run on common hardware like a computer and it isn't as CPU intensive as mining that requires specific computers designed to be efficient at hashing. It is also less prone to centralisation as the barrier to entry is much smaller when it comes to hardware requirements, making it easier to participate. On top of that, the cost of a 51% attack is higher for an attacker when compared to PoW. With PoW, the attacker can gather enough computation power to successfully attack the network without losing the capital needed to reach such a goal. In fact, the hardware won't lose its value after the attack and can be theoretically resold. In PoS, the attacker can gather enough coins to attack the network but that will result in extreme loss of value of the coins they are holding, losing nearly entirely the initial capital [7].

2.4.2 Ethereum Virtual Machine and smart contracts

The PoS consensus is not what makes Ethereum special. The Ethereum blockchain is not only capable of exchanging currencies between accounts but it is also able to create more complicated interactions through smart contracts.

Inside every node of the Ethereum network, there is an EVM, which stands for Ethereum Virtual Machine. The EVM is a stack-based Turing-complete virtual machine that runs its own opcodes. Those opcodes implement the most basic operations like AND, XOR, and ADD but the inputs are taken from the stack instead of registers. The EVM also implements some blockchain-specific opcodes like ADDRESS, BALANCE, and BLOCKHASH [8].

Thanks to the EVM, users can deploy smart contracts on the blockchain. Smart contracts are accounts similar to user accounts. They have their own address and also their own compiled code, written using the EVM opcodes. A contract is basically a list of methods that can be called with parameters and that can perform various actions like sending ETH, doing checks or any other computation. Every EVM execution is done on a transient memory that is deleted at the end. Every contract has its own permanent memory or state that can be written or read by the contract itself.

By generalising, Ethereum can be seen as a complex state machine. The state is composed of all the accounts balances and all the smart contracts balances and their internal state. Every transaction, which can be either a transfer or a contract method call, modifies the current state to a new state.

$$Y(S, T) = S' \qquad [8]$$

The EVM and smart contracts allow developers to create a variety of applications running on the blockchain with complex logic. For example, one of the most famous applications is creating a cryptocurrency, called token, inside the Ethereum ecosystem. In fact, a contract can be created where the internal state records which address owns which quantity of this new token. The token contract can then expose methods to send tokens between addresses. The smart contract will be able to check if the sender address has enough tokens and it will be able to transfer it by changing its internal state variables. To make integration easier for those tokens, some standards have been defined like ERC20 (Ethereum Request for Comment) for fungible tokens. This standard defines an interface a contract should follow to allow easy integrations of fungible tokens inside the ecosystem.

```
function name() public view returns (string)
function symbol() public view returns (string)
function decimals() public view returns (uint8)
function totalSupply() public view returns (uint256)
function balanceOf(address _owner) public view returns (uint256
    balance)
function transfer(address _to, uint256 _value) public returns (bool
    success)
function transferFrom(address _from, address _to, uint256 _value)
    public returns (bool success)
function approve(address _spender, uint256 _value) public returns (
    bool success)
function allowance(address _owner, address _spender) public view
    returns (uint256 remaining)
```

Listing 2.1: ERC20 contract standard

On top of the tokens, many more applications can be built for example decentralised token exchanges running entirely with smart contracts or stablecoins, tokens that are meant to have the same value of some other asset like USD or gold.

2.4.3 Solidity

To write a contract, a programming language that compiles to EVM bytecode is needed. Solidity is, at the moment, the de facto standard for programming when it comes to Ethereum smart contracts.

Solidity is a statically-typed programming language. It supports various data types like signed and unsigned integers up to uint256, strings, arrays, and mappings. Solidity supports inheritance, enabling developers to create modular and reusable smart contracts. You can build on existing contracts by inheriting their functions and variables, and then add your own custom functionality. Another feature of smart contracts is events. They allow you to emit notifications about specific occurrences within the contract that can be later easily fetched. This is really important as the only way to read the state of the contract is to either call a view function within the contract or listen for events. A best practice is to emit an event whenever the internal state of the contract has been changed.

A simple example of a contract can be seen on listing 2.2.

```
pragma solidity >=0.5.0 <0.7.0;

contract Coin {
    // The keyword "public" makes variables
    // accessible from other contracts
    address public minter;
    mapping (address => uint) public balances;

    // Events allow clients to react to specific
    // contract changes you declare
    event Sent(address from, address to, uint amount);

    // Constructor code is only run when the contract
    // is created
    constructor() public {
        minter = msg.sender;
    }

    // Sends an amount of newly created coins to an address
    // Can only be called by the contract creator
    function mint(address receiver, uint amount) public {
        require(msg.sender == minter);
        require(amount < 1e60);
        balances[receiver] += amount;
    }

    // Sends an amount of existing coins
    // from any caller to an address
    function send(address receiver, uint amount) public {
        require(amount <= balances[msg.sender], "Insufficient balance
.");
        balances[msg.sender] -= amount;
        balances[receiver] += amount;
        emit Sent(msg.sender, receiver, amount);
    }
}
```

Listing 2.2: Simple contract example [9]

It is a very simple example of a token implementation. At the top of the code, there are the state variables of the contract.

```
address public minter;
mapping (address => uint) public balances;
```

Listing 2.3: Contract variables

In this case, there is an address variable to record who created the contract and also a hashmap to record the balance of the token for every address.

Under that, there is the constructor that runs when the contract is deployed.

```
constructor() public {
    minter = msg.sender;
}
```

Listing 2.4: Contract constructor

At the end, there are all the methods of the contract that can be called. The mint function, for example, creates new tokens and sends them to the receiver address passed as a parameter. This function can only be called by the owner of the contract as the require statements impose that the sender of the transaction is the same as the owner saved in the state variable ‘minter‘ that was initialised with the constructor.

```
    require(msg.sender == minter);
```

Listing 2.5: Require statement

The send function instead can be used by anyone to send tokens to another address. A check of the sender balance is performed to make sure the sender is not allowed to spend more than they have.

Therefore, the Ethereum blockchain allows to create agreements enforced by code between parties. The agreement, represented by the smart contract, once deployed cannot be modified and it can be easily audited by all parties for transparency.

2.4.4 MetaMask

MetaMask is a popular and widely used cryptocurrency wallet and browser extension that primarily focuses on providing a user-friendly interface for interacting with decentralised applications (DApps) and managing Ethereum-based assets.

Metamask is important because, through their browser extension, it allows websites to connect to it and users can share their address with the website without exposing their private keys. The website can also request, through the extension, to sign messages and transactions. Those requests have to be explicitly approved on Metamask before they are confirmed. Metamask will provide the user with a clear overview of what they are accepting and only then it will use the private key to sign the payload. The website, during the whole process, never gains access to the private keys but only to the approved signatures and related information.

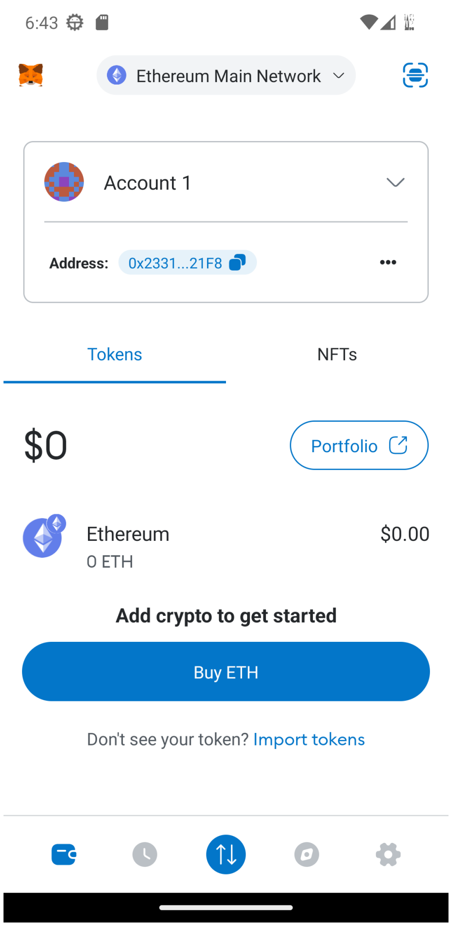


Figure 2.6: Metamask app

2.5 MaaS

The traditional transportation system is characterised by local and fragmented services. Each one of those systems is characterised by its own ticketing scheme and its own payment methods, making it very hard for the final user to navigate between them. This leads to inconvenience, suboptimal travel experiences, and limited options for sustainable mobility. To overcome these difficulties, there is a need for a comprehensive and integrated approach that combines various transportation services into a cohesive system.

This is where MaaS comes into play. MaaS stands for Mobility as a Service. It is a term used to describe the combination of several transportation providers onto a single, easily navigable platform. By fusing several means of transportation, including public transportation, ridesharing, bike-sharing, vehicle rentals, and more, MaaS aims to give passengers convenient and seamless travel experiences.

Users of MaaS platforms can plan, reserve, and pay for their travel using a single interface like mobile apps or internet platforms that provide them with a variety of transportation options.

MaaS seeks to promote overall mobility, lessen congestion, and encourage the use of different modes of transportation by encouraging their use and providing a unified solution.

The main actors of the MaaS ecosystem are:

- Mobility Service Providers (MSP) are all the businesses that provide mass transportation like buses, trams or shared mobility like bikes and electric scooters.
- MaaS Operator is the operator that allows the user to book multiple services offered by multiple MSPs through a single platform [10].

Chapter 3

Rationale for blockchain use in MaaS

In this chapter, there will be an overview on what are the current challenges of MaaS and how the use of blockchain could solve some of those challenges. There will also be an outline of some of the potential drawbacks of the blockchain and how those could be affected by the choice between a private or a public network.

3.1 Current challenges in MaaS

Although MaaS is promising, there are still challenges that need to be fixed in order to be a viable option for the future.

For example, one of the biggest problems is the fragmented nature of the transportation system. Given the existence of various suppliers, technologies, operators, and business models, creating seamless digital infrastructures for MaaS is a challenging undertaking. Every country, every region, and every local city uses its own solution for ticketing, its own app and its own payment method making the integration between them very hard. The difficulty lies in combining these current ticketing options into a single framework that can accommodate the various stakeholders.

One other problem of MaaS is the intrinsic different ticketing disparity between public transport and shared/on-demand mobility. The latter frequently have access fees that are established dynamically, in contrast to fixed-fare systems, which makes it challenging to manage the integration of the two various fare/pricing systems. For the purpose of creating a smooth, adaptable, and dependable mobility chain, these worlds must be bridged.

Last but not least, trust and privacy issues between Mobility Service Providers.

Data exchange plays a vital role in the success of a MaaS application. However, coordination issues have resulted in data privacy concerns, as private mobility operators are reluctant to share their data due to intense competition within the industry. Conversely, public transport authorities and operators hesitate to open their systems without a compelling business case. Moreover, there are apprehensions about losing touch with public transport users and potential revenue loss in the long run. Overcoming these challenges requires addressing regulatory aspects, building trust among stakeholders, ensuring data privacy, and developing collaborative frameworks that strike a balance between competition and cooperation. [10]

3.2 Advantages of blockchain

Up until now, a description of the current problems MaaS is facing has been discussed. In this chapter, there will be an overview of how the use of blockchain could help solve some of those problems.

The first benefit of the use of blockchain is the possibility to define a common standard. The blockchain allows to upload a contract that defines all the standards for ticketing such as the data format for a ticket or how this data has to be exchanged. Once MSPs adhere to the standard and register through the blockchain, MaaS Operators are able to easily discover new services and easily interact with the new MSPs without the need for additional code. This makes it very easy to develop platforms that offer multimodal journeys on different MSPs.

Having the blockchain as the unique ticketing system gives MSPs some advantages too. In the first place, it lowers the costs of maintaining the ticketing infrastructure. Most of the logic will be moved on the blockchain and the blockchain infrastructure is shared into multiple nodes run by multiple entities. The blockchain also offers a unique payment system shared between all MSPs. This eliminates the need for any third party to handle payment like credit card circuits or banks.

3.3 Disadvantages of blockchain

The use of blockchain technology can also bring some disadvantages. Some of them depend on what kind of network is used (private network or public network) and the difference will be discussed in the next section.

One of the main problems is privacy for users. As the transactions can be visible to anyone in a public distributed ledger, privacy concerns arise when you store all movement information of a person on a blockchain. The only defence is the pseudo-anonymity of public addresses, but once the association of the real person with the address is achieved, all movement information can be easily retrieved. This applies only to public networks like Ethereum but it could be mitigated by using private networks.

Public networks also face the problem of transaction fees. As fees are determined by the free market and the current blockchain workload and since a public blockchain is shared with thousands of other applications, fees can have very large fluctuations and it may determine the unusability of the system over certain periods of time. For example, as can be seen on Etherscan [11], the average fees paid to include a transaction are quite volatile depending especially on the amount of transactions being sent. In 2022, the average amount for sending transactions was around 22 USD. This fee cost would make the blockchain unusable for the purpose of the project of this thesis during those times. The problem of transaction fees is less relevant if a public network that is cheaper than Ethereum is chosen. For example, there are plenty of EVM-compatible networks that have lower fees like BNB chain, Avalanche and many more. Another viable possibility is to use a Layer 2 solution (usually referred to as L2). An L2 is a network of their own that has the aim of making Ethereum scalable. They allow you to move some of the transactions off Ethereum (thus reducing their cost) and only periodically batch those transactions together and submit them to Ethereum in a unique transaction. Although all those alternatives seem very promising, the focus of this thesis will mostly remain on Ethereum as it is the most known blockchain and therefore the one with the biggest community and support.

A more general problem is defining a standard for tickets. The final solution should be able to support most (if not all) of the possible business models of every MSP. A generic standard would be able to include most of the MSP but it would mean that fewer checks and logic can be carried out on the blockchain making the latter just a distributed database. A stricter logic, on the other end, would inevitably cut off some of the existing ticketing solutions or make the code on the blockchain side very complicated and hard to handle.

3.4 Comparison between public and private networks

When it comes to the first two disadvantages described in the previous section, the difference can be made by a wise choice of type of network.

	PUBLIC	PRIVATE	CONSORTIUM
Structure	Decentralised	Centralised	Partially decentralised
Access	Open read/write	Permissioned	Permissioned
Speed	Slower (around 10 mins)	Faster (same as a transactional system)	Depends on the number of nodes
Consensus	Proof of work, Proof of Stake	Pre-approved	Pre-approved
dentity	Anonymous	Identity known	Identity known
Use cases	Cryptoeconomy	Reference data management	Secure data sharing
Examples	Bitcoin, Ethereum, Dash	MONAX, Multichain	R3 EWF

Table 3.1: Comparison between public and private blockchains [12]

Public networks, like Ethereum, are run by the community. Everyone can join the network with a node by running an Ethereum client on their own hardware. This means that all the data written in the blockchain is public and can be easily accessed by anyone. On a private network, access can be restricted to a defined number of parties. This reduces transparency from the point of view of the user but allows to keep the transactions, and therefore the data, private.

Private networks can solve the privacy and fee problems. The private network could be run simply by MSPs that will be able to control the network and also share the data with each other. By using a blockchain, the network can't be modified or tampered with by only one or a small group of MSPs but a quorum would be necessary to make any change. This allows them to increase trust in the system and, more importantly, between each other. Another advantage of private networks is that they are more scalable and fast as they will have fewer nodes that will have to coordinate with each other but also less transaction traffic as the blockchain is hosting a single use case. Fees for sending transactions can also be decided by the MPSs or removed altogether.

Although private networks seem a better solution for this problem, the thesis will focus more on the implementation of the smart contract itself rather than the design of the network for time constraints reasons. As mentioned before, Ethereum will be used as it is the most widely known EVM-compatible blockchain with the biggest community and support.

Chapter 4

Design and implementation

In this chapter, the proof of concept that has been developed will be presented. First, a brief introduction of the problem to be addressed will be provided. Then, there will be a presentation of the design and the requirements for the system. At the end, there will be a description of the actual implementation of the blockchain contracts, the mobile application and all the required backend infrastructure.

4.1 Problem description

The aim of the thesis is to create a mobile application that allows users to rent or book tickets for the widest range of public transport in a decentralised way by using blockchain technology.

First of all, there will be a need to define which public transport can be supported. Since the aim is to support as many public transport as possible, the best way to do it is to generalise the concept of ticketing as much as possible while still being able to run some validation on the blockchain side. To reach this goal, all public transport has been split into two main categories:

- **Unlockable free-floating transport** that includes all kinds of transportation modes where the MSP makes available a set of vehicles that can be parked everywhere inside a specific perimeter of a city. To be used, they have to be explicitly unlocked by the user (usually through a mobile application). Once the user has finished their journey, the vehicle is locked again and it will be back available to be used by anyone else. This category includes bike sharing, car sharing, and scooter sharing but they can also be adapted for parking fees or taxis.
- **Prepaid travel-specific tickets** that include all kinds of transportation modes where a single-use ticket is required to get access to it. This category

contains all traditional city public transport with fixed fares like buses, metro, or trams but also public transport with variable fares like trains, planes, or boats.

From this point, those 2 categories will be treated as 2 different problems each of which will require its own solution.

4.2 Design

4.2.1 Actors

Having defined what is the problem to solve, the next step involves defining the entities that will participate and interact with the system.

There are 4 main actors:

- Mobility Service Provider is the entity that runs the transport business. It manages his vehicles by interacting directly with the blockchain and it exposes an API to advertise all the routes it covers
- MaaS Operator is the entity that aggregates the data of multiple MSPs and displays it to the user through the app
- Vehicle is associated to a unique address and can interact with the blockchain to update its status
- User utilises the app to book a trip or rent a vehicle

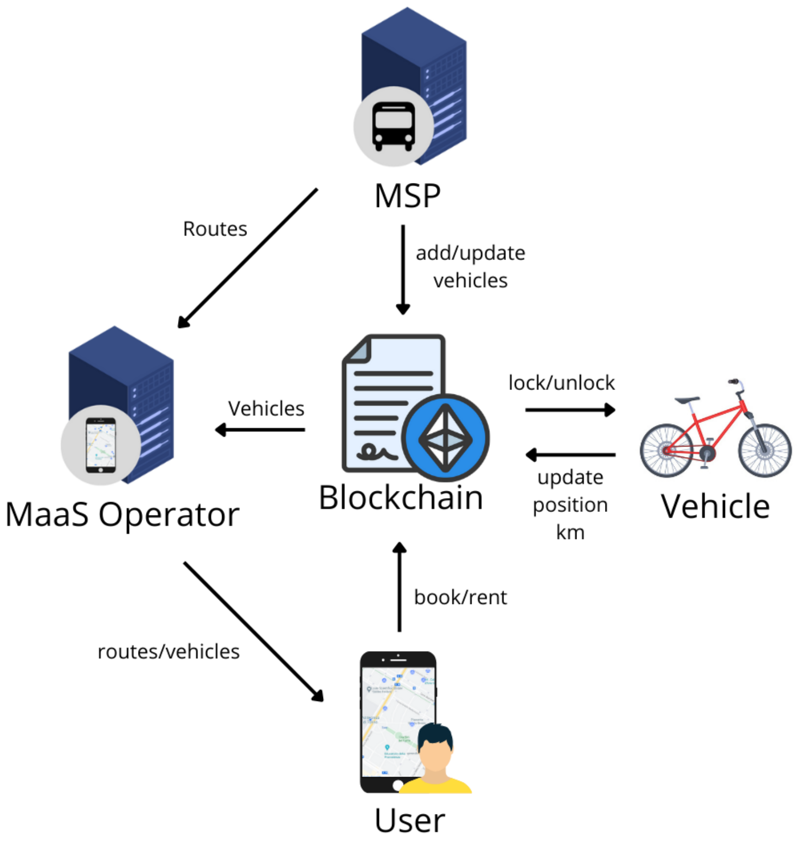


Figure 4.1: Context diagram

4.2.2 Functional Requirements

After discussing the actors and their interconnections with the system, the next step is to proceed with listing the functional requirements.

One important degression before discussing the requirements is deciding what is better to keep on-chain as opposed to off-chain. The term on-chain means that the data is directly stored on the blockchain while off-chain means that the data is stored in traditional databases but enough information is stored on the blockchain to verify that those data didn't change. This can be achieved by either storing hashes of the data or signatures in case some kind of authentication is wanted too.

The main problem of storing data on-chain is that it can be very expensive. In the case of public blockchains, the fee to pay to include a transaction depends on the amount of computing power needed to execute such an operation. In particular, storing data permanently in a blockchain can be one of the most expensive operations. On top of that, even if the system is run on a private network where fees can be secondary, storing a lot of data has another side effect too. In fact, bigger and more frequent transactions lead to bigger blocks. If the blocks are too big, it may become very difficult for all the nodes participating in the network to keep up with each other and remain in sync leading to a degraded network. For this reason, when developing an application that runs on a blockchain, some thought has to be put into what should be kept on-chain or off-chain.

For this application, it has been decided to keep everything about unlockable transports on-chain as it doesn't require storing too much information. For the tickets side, all the routes advertised from the MSPs will be kept off-chain as there can be quite a lot and frequent data to update. What is actually stored on-chain is only the tickets actually purchased by the users.

Having specified that, it is possible to proceed with the functional requirements.

Registration

1. MSP can register themselves on the blockchain to provide services.
2. MSP can register their own vehicles. An address is associated with the vehicle.
3. MSP can advertise the routes that it offers.
4. MSP can define fares for every vehicle/route.
5. MSP can modify fares for any vehicle/route.

Unlockable transport

1. User can unlock a vehicle offered by the MSP not being used by any other user.
   (a) User can fetch the list of vehicles available and their properties and status.
   (b) User can pick the vehicle that he prefers.
   (c) User can pick the best offer proposed by the MSP that owns the chosen vehicle.
   (d) User can unlock the vehicle by transferring currency to the contract. The user will be able to use the vehicle as long as the currency transferred is enough.
2. Vehicle periodically reports its status like battery, km travelled and position.

3. User can lock back the vehicle if they are currently using it. All the unspent currency will be sent back to the User.

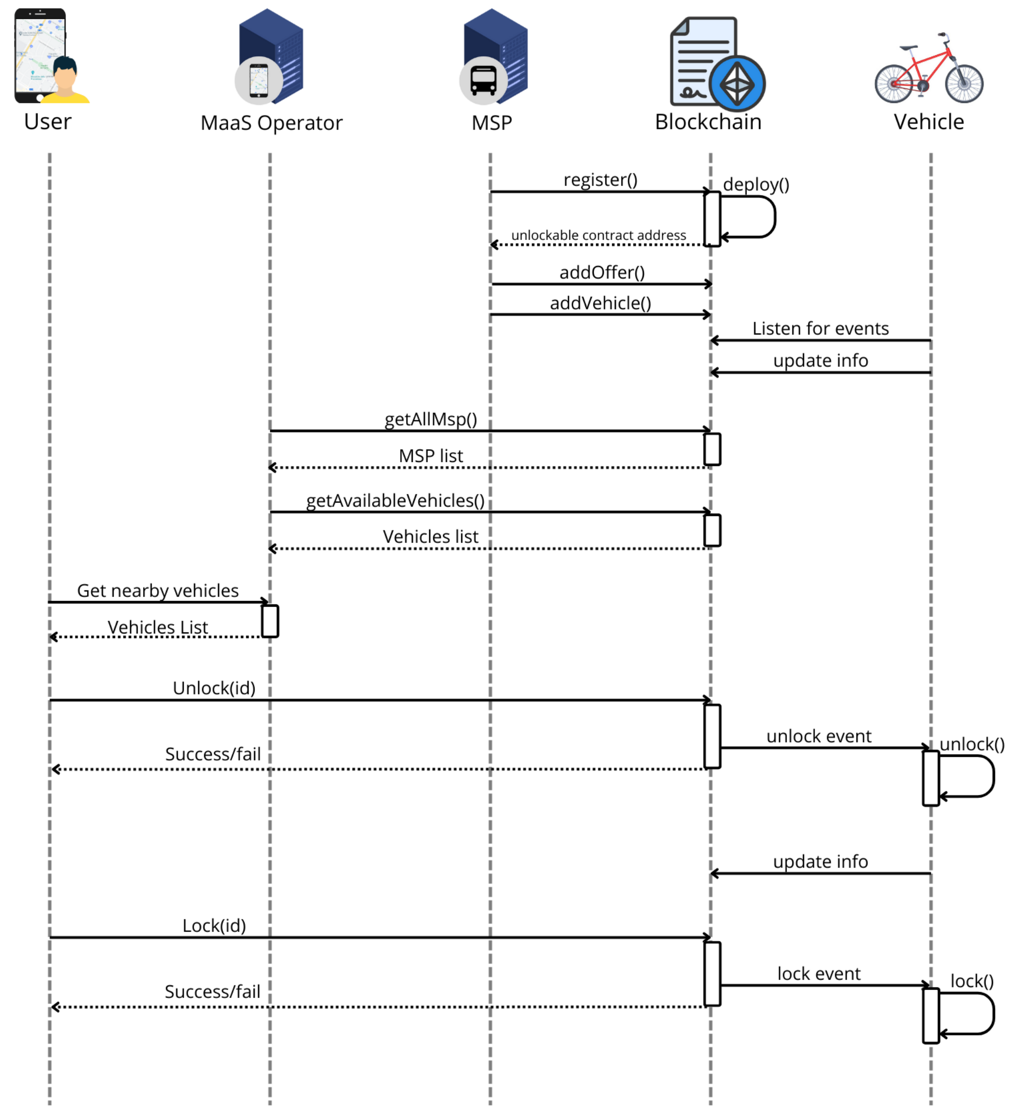


Figure 4.2: Interactions unlockable

Prepaid Travel-Specific Tickets

1. User can buy a ticket from any MSP.
   (a) MSP should offer a standard API where he advertises his routes.
   (b) User can fetch all registered MSPs on the blockchain.
   (c) User can fetch routes offered by every MSP through their standard API.
   (d) MSP can sign route offers.
   (e) User chooses a route offer and pays by sending the signed offer from the MSP with the related currency to pay the ticket. The ticket is created on-chain.
2. MSP can validate tickets.
3. MSP can set the ticket to used to avoid replays.

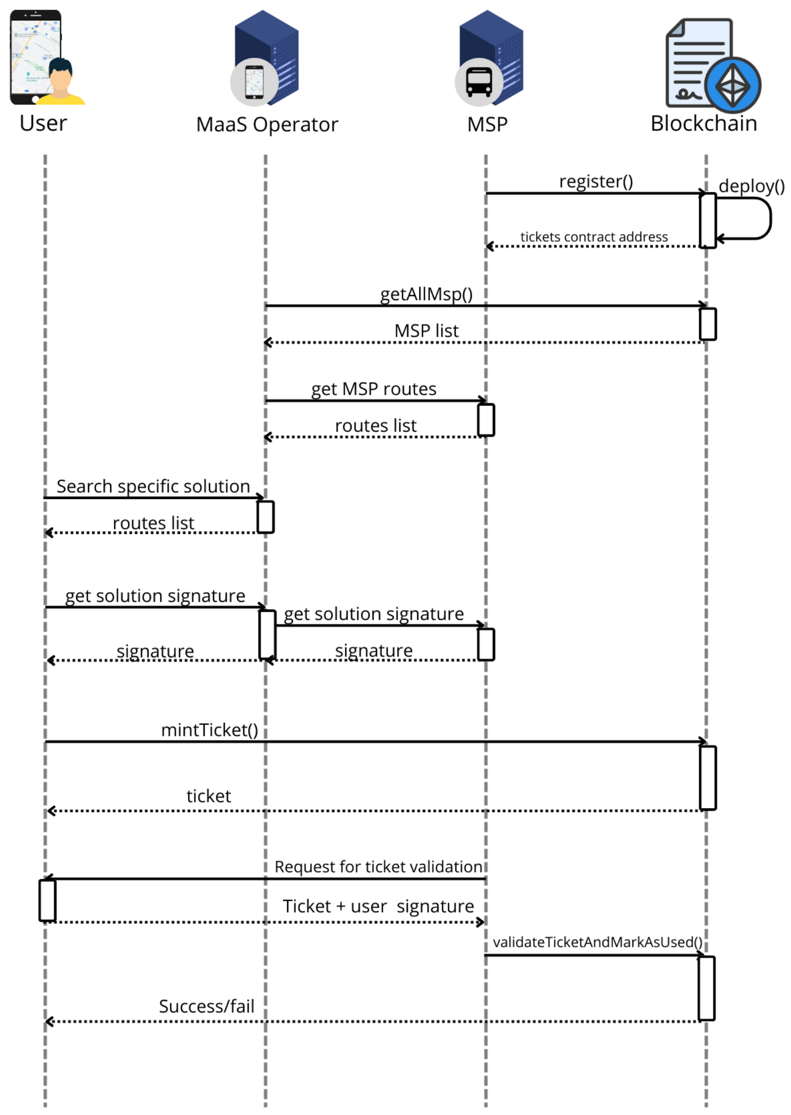


Figure 4.3: Interactions tickets

4.3 Implementation

4.3.1 Blockchain

On the blockchain side, there are a total of 3 contracts:

1. MspRegister.sol
2. PrepaidTickets.sol
3. UnlockableTransport.sol

The contracts have been written using solidity and the foundry framework [13]

MspRegister

`MspRegister` is the main contract. Its job is to be able to register new MSPs on the blockchain and also retrieve them when required.

The structure is very simple. Apart from the getter to retrieve the list of registered MSPs, it is composed of a single function called `register`.

```
    function register(
        string calldata name,
        string calldata endpoint
    ) external returns (UnlockableTransport, PrepaidTickets)
```

Listing 4.1: Register function signature

To register a new MSP, it is required to call this `register` function and pass the name and the route endpoint of the MSP. The function will internally deploy the two other contracts `PrepaidTickets` and `UnlockableTransport` specific for the registering MSP and return their addresses. All this information about the MSP will be saved on the blockchain.

Internally (and when retrieving the MSP from the getters), an `Msp` struct is returned that lists the name and the endpoint of the MSP, the latter used to retrieve all the routes for this specific MSP. On top of that, the struct also contains the addresses of the MSP `PrepaidTickets` and `UnclockableTrasport` contracts deployed during the registration. Those addresses are needed to interact with those contracts and therefore rent vehicles or buy tickets from this specific MSP.

```
// Struct representing a Mobility Service Provider
struct Msp {
    string name; // Name of the MSP
    string endpoint; // Url of the MSP standard API
    UnlockableTransport unlockableContract; // Address of the MSP
   unloackable contract
    PrepaidTickets ticketContract; // Address of the MSP ticket
   contract
}
```

Listing 4.2: Msp struct

PrepaidTickets

The `prepaidTickets` contract has the role to store and validate all the tickets relative to the owner MSP.

Users can mint a new ticket by using the `mintTicket` function.

```
function mintTicket(Types.Ticket calldata ticket) external payable
```

Listing 4.3: mintTicket function signature

It accepts only one parameter which is a struct of `Ticket`.

```
// Struct that describes a point of interest with its name and
    coordinates
struct PointOfInterest {
    string name; // Name of the point of interest
    uint256 x;
    uint256 y;
}


// Struct that describes a ticket
struct Ticket {
    bytes signature; // Signature from the MSP over the id
    bytes32 id; // keccak256 over all fields of the ticket
    TypeOfTransport typeOfTransport; // Transport type
    uint256 price; // Price to pay (in wei) for the ticket
    PointOfInterest from; // Place from where the journey starts
    PointOfInterest to; // Place where the journey ends
    uint256 departureTime; // Time of departure
    uint256 eta; // Estimated time of arrival
    uint8 class; // Class at which the passenger travels (MSP defined
    )
    bytes custom; // Custom data (MSP defined)
    uint256 expiration; // Expiration time (in unix) after which the
    ticket offer is invalid
    address validFor; // Address to which this ticket offer is valid
    TicketState state; // The state of the ticket
}
```

Listing 4.4: Ticket struct

The ticket has a few general fields like from and to that describe the departure and arrival for the journey, the departure time, the estimated arrival time and other general attributes.

It is important to highlight the meaning and purpose of `id` and `signature`. The `id` is a unique hexadecimal string that identifies the ticket. It is generated by hashing with keccak256 the concatenation of all the other properties. The `signature` is used to validate the ticket. When the MSP exposes the routes that it offers, its API also allows users to sign valid offers. The signature is just an ECDSA (Elliptic Curve Digital Signature Algorithm) signature over the id. The user, once he has fetched the routes and picked the one he prefers, needs the MSP to sign the offer. Once he has the signature, he can submit the ticket to the blockchain and the mint function will verify that the signature corresponds to the MSP address and only in that case create the ticket. The offer signed by the MSP is valid only for a specific address indicated by the `validFor` property and it also has an expiration that can be set by the MSP itself with the `expiration` field. This function is a payable function. For the creation of the ticket to be successful, the user has to send exactly the amount of eth to the contract specified in the `price` field.

Once the ticket has been created, the MSP can validate a ticket with the `validateTicketAndMarkAsUsed` function.

```
function validateTicketAndMarkAsUsed(
    bytes32 id,
    bytes calldata userSignature
) external onlyOwner
```

Listing 4.5: validateTicketAndMarkAsUsed function signature

To do so, the MSP has to request the signature of the ticket id from the user (so that the MSP can't validate tickets without the user's approval). Once the user has provided the signature, the MSP can call the validate function with the id and user signature and the ticket will be internally marked as used.

The last function is `withdraw` which allows the MSP to withdraw all the funds made from ticket sales from the contract to their own wallet.

```
function withdraw() external onlyOwner
```

Listing 4.6: withdraw function signature

Unlockable Transport

Unlockable transport is the most complicated contract of the three. Basically, it allows the owner of the contract (the MSP) to manage their fleet of vehicles and it allows users to rent them. It is also possible for the vehicle itself to send updates about its status like position, km and battery levels.

The owner has lots of functions available. He can add a vehicle by calling `addVehicle` and passing a `vehicle` struct as a parameter.

```
function addVehicle(
    Types.UnlockableVehicle calldata vehicle
) external onlyOwner
```

Listing 4.7: addVehicle function signature

```
// Struct that describes a vehicle that can be locked and unlocked
struct UnlockableVehicle {
    uint256 id; // Unique ID that represents this vehicle
    Position position; // Current position of the vehicle
    uint256 totalKm; // Total km for the vehicle (for car rent)
    uint8 battery; // Current battery/fuel percentage
    address vehicleAddress; // Address associated with pkey of
   vehicle
    bool enabled; // Boolean to indicate if the vehicle is currently
   in service
}
```

Listing 4.8: UnlockableVehicle struct

The most important part here is the `vehicleAddress` field. This address is the address of the vehicle and it determines which private key (corresponding to this address) can interact with the contract as the vehicle. The id is defined by the owner but it has to be unique within the contract and it has to be bigger than 0.

The owner can also delete a vehicle by passing its id or change the vehicle availability status. The bool `enabled` defines the availability of the vehicle. If the owner needs the vehicle to be put off renting for a while (for example for maintenance), he can disable the vehicle. A disabled vehicle cannot be rented anymore by the user and, if the vehicle is currently in use, it will allow it to finish the current journey but it can't be used anymore once locked. Only vehicles that are not in use can be deleted from the contract. Disabling it can be also used to make it easier for the owner to delete a vehicle without risking someone else renting it in the meantime.

The owner can also add an offer.

```
function addOffer(
    Types.UnlockablesOffer calldata offer
) external onlyOwner
```

Listing 4.9: addOffer function signature

Every contract can have multiple offers (that can be either enabled or disabled) and the user can pick any of those when they want to start renting a vehicle.

```
// Struct that describes a possible offer at which a vehicle can be
   rented
struct UnlockablesOffer {
    uint256 id; // Unique ID of the offer
    uint256 pricePerKm; // Price (in wei) for every km
    uint256 pricePerSec; // Price (in wei) for every sec
    uint256 UnlockPrice; // Price (in wei) to unlock the vehicle
    uint256 KmAllowance; // Free of charge amount of Km before
   applying the above rate
    uint256 TimeAllowance; // Free of charge amount of seconds before
    applying the above rate
    bool enabled; // Boolean to indicate if the offer is currently
   available
}
```

Listing 4.10: UnlockablesOffer struct

An offer is made of multiple parts. The owner can set a price for each km (mainly used for cars), and a price for each minute (used by both cars and bikes for example). The owner can also define an allowance for those two. When an allowance is defined, the user will not pay any amount until the allowance is consumed. The last parameter they can modify is the price to pay when unlocking.

As for vehicles, the owner can delete and enable/disable offers. As said before, also vehicles can interact with the contract. In particular, they can set their position, their km and their battery.

```
function setPosition(
    Types.Position calldata position
) external onlyVehicle
function setKm(uint256 km) external onlyVehicle
function setBattery(uint8 percentage) external onlyVehicle
```

Listing 4.11: setPosition, setKm and setBattery function signatures

While the position and battery are used as pure information, the km fed through the vehicles are used to calculate the charge for the user currently renting. The vehicle has also the possibility to use the forceLock function that will finish the current renting and charge the user accordingly. This is used, for example, when the user has finished the money they provided to the contract for renting.

```
    function ForceLock() external onlyVehicle
```

Listing 4.12: ForceLock function signature

From the side of the user, excluding the getters, they can interact with only 2 functions: `unlock` and `lock`.

When the user is in front of a vehicle, he can call `unlock` to start renting it. The `unlock` functions accept as parameters the id of the vehicle to rent and the offer id to use for the rent.

```
    function Unlock(uint256 vehicleId, uint256 offerID) external
   payable
```

Listing 4.13: Unlock function signature

The function is a payable function. The user has to send to the contract enough credit to pay for the unlock fees and also allow him to complete the journey. When the user successfully unlock a device, the rent is kept track thanks to a status variable called `s_status` which maps every vehicle to their current rent status.

```
// Struct that represent the status of the ongoing rental
struct RentStatus {
    address currentOwner; // Address of the owner currently renting
    uint256 offerId; // Id of the offer being used by this rent
    uint256 startKm; // Total km of the vehicle at which the rent was started
    uint256 startTime; // Time (in unix) at which the rent was started
    uint256 maxToSpend; // Amount of the ETH sent when the vehicle was unlocked (and therefore the maximum that can be spent on this rent)
}
```

Listing 4.14: RentStatus struct

On the rent status, it is stored which address is currently using the vehicle, which offer they are using, how much credit they sent to the contract for the rent and it keeps track at which km and time the rent started. Those 2 parameters are used to calculate the final fare.

When the user finishes his journey, they can use the `lock` function to lock the vehicle back and make it available for the next customer.

```
    function Lock(uint256 vehicleId) external
```

Listing 4.15: Lock function signature

When the `lock` function is called, the total fare for the rent is calculated and the user is refunded the difference between the initial credit and the fare cost.

As for tickets, the owner can use the `withdraw` function to move all the income from the contract to his own wallet.

```
    function withdraw() external onlyOwner
```

Listing 4.16: withdraw function signature

Testing

The contracts have been tested using mainly unit tests. All the 3 main contracts have a coverage higher than 90%. Fuzzing was also considered for testing the contracts but it was deemed hard to apply to this specific case. The difficulty was to find an invariant that could be considered useful. The way fuzzing works on the foundry framework requires providing an invariant, that is one or more statements that always hold true regardless of the inputs and the functions called. Foundry will then take care of fuzzing combinations of function calls with random parameters to try to break any of the invariants. On the specific contracts of this thesis, it was difficult to find any meaningful invariant that would have made fuzzing testing relevant and therefore it has not been used.

4.3.2 Mps Backend

The Msp backend provides all the data that were not stored in the blockchain for economical and speed reasons.

This server has to be run on the endpoint that the MSP specified when registering on the blockchain and it has to follow a standard structure to allow users to easily gather its content.

Through this server, the MSP can advertise all the routes they service. The server needs to implement at least 2 routes:

GET /api/routes/

This endpoint will return all the currently available routes for the MSP. The routes have the schema shown on listing 4.17.

```
class PointOfInterest {
    name: string;
    x: number;
    y: number;
}


class Routes {
    id: string;
    typeOfTransport: TypeOfTransport;
    price: number;
    from: PointOfInterest;
    to: PointOfInterest;
    departureTime: number;
    eta: number;
    classNumber: number;
    custom: string;
}
```

Listing 4.17: ES6 (ECMAScript-2015) Listing

This endpoint will not be used directly by the user as it would be too expensive from the time perspective to fetch all routes from all MSPs just to find a travel solution. For this reason, this endpoint will be fetched by the MaaS Operator that runs the app backend to get all the routes at fixed intervals. The MaaS Operator will handle the task of aggregating all the data and returning it in a more user-friendly way to the user.

GET /api/sign/:address/:id

This endpoint is used to sign a specific valid travel solution for a specific address so that the user can submit a ticket to the blockchain. This endpoint accepts two URL parameters. The address of the user that will be inserted in the field `validFor` of the ticket struct that will be returned and the internal MSP id of the route. The route will be signed only if the id refers to a valid route offered by the MSP.

For this thesis, the web server has been developed on nodejs with typescript. Expressjs has been used as a framework to handle the requests and SQLite as a database.

4.3.3 App Backend

The app backend fulfils the purpose of being the middle between the data on the blockchain and the user. As it has been said, it fetches the data at regular intervals from both the blockchain and the MSP backend and stores them on its own database. From this data, they provide a set of APIs used by the app to display the data in an easy-to-read way for the user.

The API for the App backend doesn't have to be standard therefore any MaaS Operator can make their own. On the application developed for this thesis, there are three main groups of routes:

- Google APIs: the backend provides some Google Maps API for things like walking distance for the nearest vehicle or searching a place from text
- Routes APIs: provides a way for the user to search the most convenient route given their destination, arrival and time of departure. It also provides a way to relay the signature request of a route to the appropriate MSP
- Vehicles APIs: similarly to the routes APIs, it provides a way for the user to get the vehicles in a defined area of a map, usually the vehicle closer to them. It also provides a way to check if they are currently renting any vehicle from any MSP or not.

Like the MSP backend, this web server has been developed on nodejs with typescript. Expressjs has been used as a framework to handle requests and SQLite as a database.

4.3.4 App

The application is the only point of contact with the user. It fetches the information needed from the backend and the blockchain and it manages the transaction with the blockchain through an external wallet. The application has been developed with React Native.

In order to be able to make any action in the app, the user has to log in through a third-party wallet. The wallet will handle the private key management and also signing and sending transactions.

To handle connections with wallets, the WalletConnect library has been used. The library provides an easy way to connect to multiple web3 wallets with a few lines of code. When the user clicks on the login button, in the user tab, a modal is opened at the bottom with a list of wallets to connect to. In Figure 4.5, the wallet being used is Metamask. When clicking on the wallet of choice, the respective wallet app is opened and a request to connect is shown to the user. When accepted, the user is redirected back to the app where he will be shown his address and all the tickets he has bought/used. Those tickets are fetched directly from the blockchain. The app also fetches the ENS (Ethereum Name System) for any username or avatar the address may have set. If there is any, it will be shown on this screen.

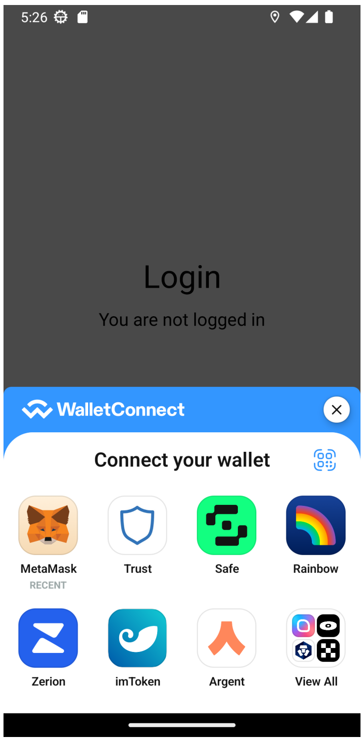
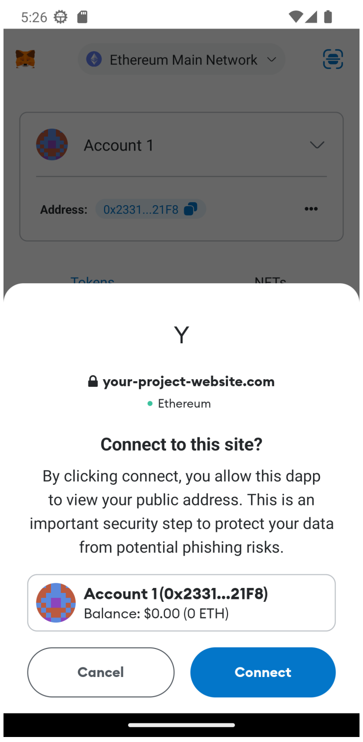


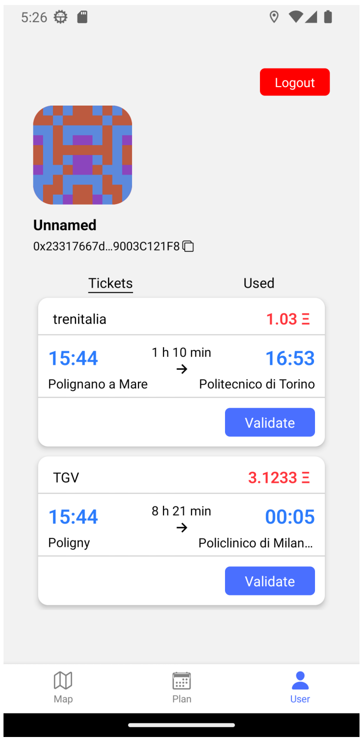
Figure 4.4: Login modal



By clicking connect, you allow this dapp to view your public address. This is an important security step to protect your data from potential phishing risks.

Account 1(0x2331...21F8)
Balance: $0.00 (0 ETH)

Figure 4.5: Metamask confirm connection

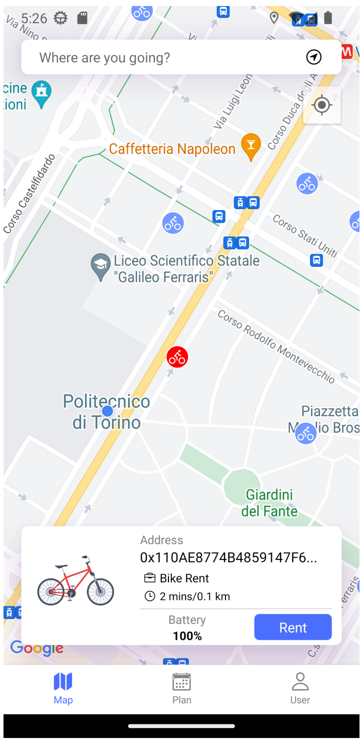


trenitalia		1.03 Ξ
15:44	1 h 10 min →	16:53
Polignano a Mare		Politecnico di Torino
TGV		3.1233 Ξ
15:44	8 h 21 min →	00:05
Poligny		Policlinico di Milan...

Figure 4.6: User tab

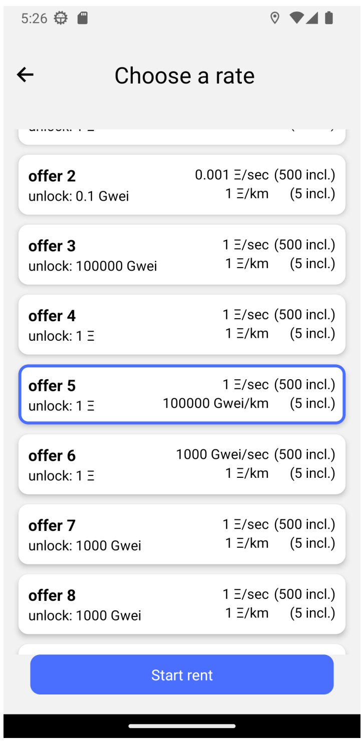
Once logged in, the user can fully use the app. On the map tab, the user can see his location and all the vehicles nearby. The vehicles are fetched directly from the app backend. When he clicks on any of the vehicles, the app shows the user the vehicle address, how distant it is and what battery level is at this time. Once clicking on the rent button, the user can choose the offer he prefers. The offers are directly fetched from the blockchain. Once the user has chosen both the offer and vehicle, he can click on start rent and he will be redirected to the wallet the app is connected to where he can review the transaction, sign it and send it to the network. When he goes back to the app, he will be able to see the vehicle he is currently renting together with all the stats of time and km passed and the expected fare to pay at the current time.

When the user has finished with the journey, he can click on the stop rent button to be redirected to the wallet to send the lock transaction.



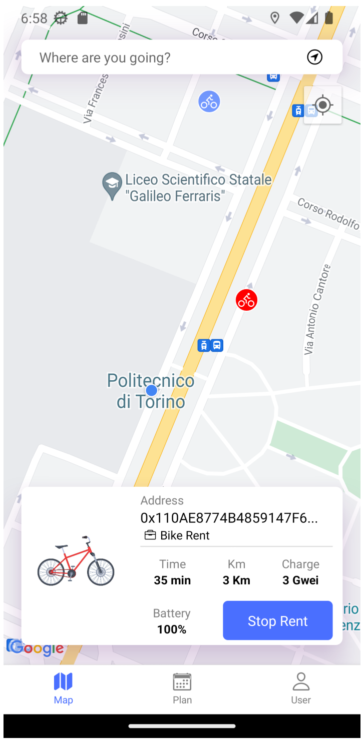
Address
0x110AE8774B4859147F6...
Bike Rent
2 mins/0.1 km
Battery
100%
Rent

Figure 4.7: Map tab



offer 2 unlock: 0.1 Gwei	0.001 Ξ/sec (500 incl.) 1 Ξ/km (5 incl.)
offer 3 unlock: 100000 Gwei	1 Ξ/sec (500 incl.) 1 Ξ/km (5 incl.)
offer 4 unlock: 1 Ξ	1 Ξ/sec (500 incl.) 1 Ξ/km (5 incl.)
offer 5 unlock: 1 Ξ	1 Ξ/sec (500 incl.) 100000 Gwei/km (5 incl.)
offer 6 unlock: 1 Ξ	1000 Gwei/sec (500 incl.) 1 Ξ/km (5 incl.)
offer 7 unlock: 1000 Gwei	1 Ξ/sec (500 incl.) 1 Ξ/km (5 incl.)
offer 8 unlock: 1000 Gwei	1 Ξ/sec (500 incl.) 1 Ξ/km (5 incl.)

Figure 4.8: Offer selection



Address
0x110AE8774B4859147F6...
Bike Rent

Time	Km	Charge
35 min	3 Km	3 Gwei

Battery
100%
Stop Rent

Figure 4.9: Map tab while renting

When it comes to booking a ticket, the user can go to the plan tab where he can input their starting point, destination and estimated time of departure. The destination and departure places are fetched thanks to the Google APIs of the app backend. Once all the fields have been completed, the app shows the best options that match the user's request (fetched always from the app's backend). Once the user chooses the best option, he is again redirected to the wallet where he can review the transaction and send it. Once sent, the ticket will be available on the user screen to be validated.

The validation can be done by clicking the validate button which will once again redirect the user to the wallet where he will sign the ticket. The signature is returned back to the app and it will be shown as a QR code that can be easily scanned by the MSP to submit the validation transaction.

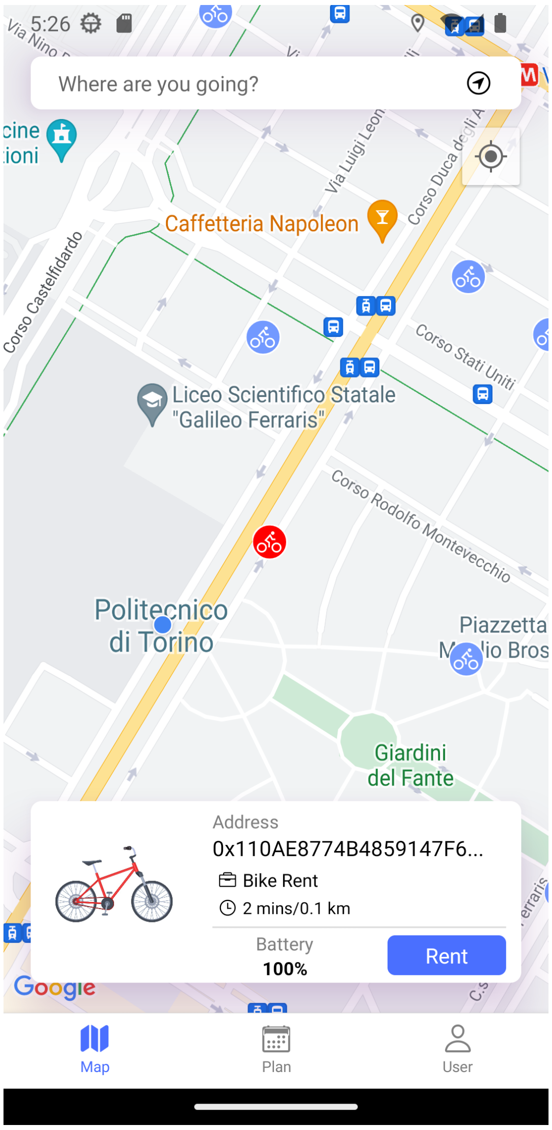
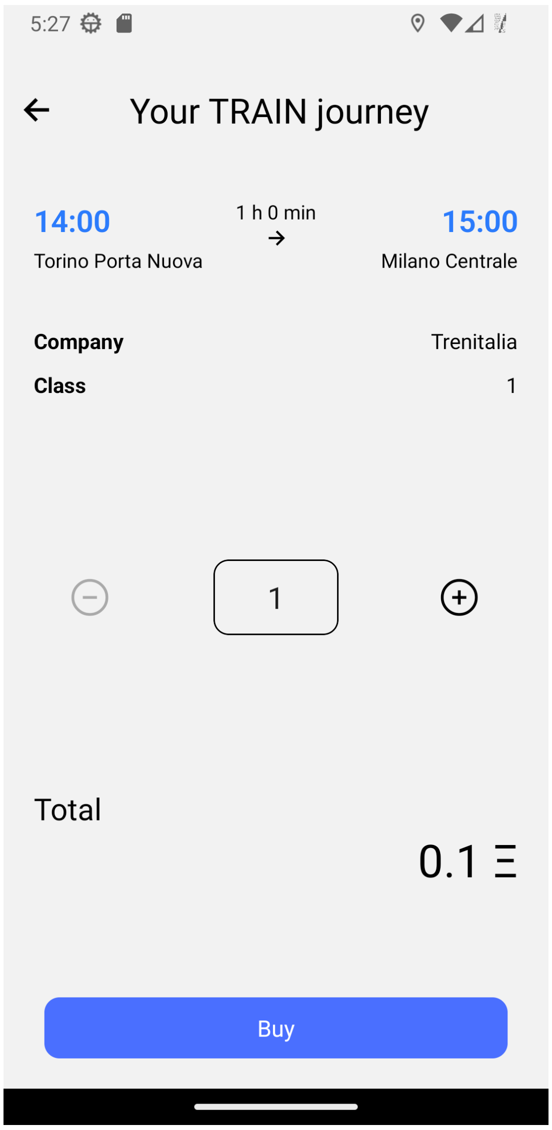


Figure 4.10: Plan tab


5:27

Your TRAIN journey

14:00	1 h 0 min →	15:00
Torino Porta Nuova		Milano Centrale

Company Trenitalia

Class 1

1

Total

0.1 Ξ

Buy

Figure 4.11: Ticket checkout

Chapter 5

Conclusion and future improvements

The thesis aimed to build a MaaS application that would run on the blockchain thanks to smart contracts. The objective was to check if a MaaS application on blockchain was possible and to see how it could fix the current problems MaaS is facing. The result shows that building a decentralised MaaS application is possible and it brings a few advantages when it comes to sharing information and trust between MSPs but also making it easy for an application to integrate several MSPs together.

The app developed is a proof of concept. A lot of work can still be done to improve the ticket search, and the interaction with the blockchain and make the blockchain contracts more efficient. The work can be expanded by deploying the smart contracts on a private custom-made blockchain which is made just for this specific use case.

Bibliography

[1] Deepak Puthal, Nisha Malik, Saraju Mohanty, Elias Kougianos, and Gautam Das. «Everything You Wanted to Know About the Blockchain: Its Promise, Components, Processes, and Problems». In: *IEEE Consumer Electronics Magazine* 7 (July 2018), pp. 6–14. DOI: `10.1109/MCE.2018.2816299` (cit. on p. 3).

[2] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System.* 2008. URL: `https://bitcoin.org/bitcoin.pdf` (visited on 11/04/2023) (cit. on p. 3).

[3] lisk.com. *What is a Blockchain?* URL: `https://lisk.com/learn/about-web3/what-is-a-blockchain` (visited on 11/04/2023) (cit. on p. 4).

[4] cisa.gov. *Understanding Digital Signatures.* Feb. 2021. URL: `https://www.cisa.gov/news-events/news/understanding-digital-signatures` (visited on 11/04/2023) (cit. on p. 5).

[5] Cloudflare. *A (Relatively Easy To Understand) Primer on Elliptic Curve Cryptography.* Oct. 2013. URL: `https://blog.cloudflare.com/a-relatively-easy-to-understand-primer-on-elliptic-curve-cryptography/` (visited on 11/04/2023) (cit. on pp. 5, 6, 8).

[6] *Elliptic Curve Cryptography (ECC).* URL: `https://cryptobook.nakov.com/asymmetric-key-ciphers/elliptic-curve-cryptography-ecc` (visited on 11/04/2023) (cit. on p. 7).

[7] Ethreum community. *PROOF-OF-STAKE (POS).* URL: `https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/` (visited on 11/04/2023) (cit. on p. 13).

[8] Ethreum community. *ETHEREUM VIRTUAL MACHINE (EVM).* URL: `https://ethereum.org/en/developers/docs/evm/` (visited on 11/04/2023) (cit. on p. 14).

[9] Ethreum community. *Solidity.* URL: `https://docs.soliditylang.org/en/v0.6.4/` (visited on 11/04/2023) (cit. on p. 16).

[10] Guido Di Pasquale, Jaap de Bie, and Jaspal Singh. *TICKETING IN MOBILITY AS A SERVICE*. July 2022. URL: `https://cms.uitp.org/wp/wp-content/uploads/2022/07/Report-Ticketing-MaaS-JULY2022-web.pdf` (visited on 11/04/2023) (cit. on pp. 19, 22).

[11] *Average Transaction Fee Chart*. URL: `https://etherscan.io/chart/avg-txfee-usd` (visited on 11/04/2023) (cit. on p. 23).

[12] Ralph Gambetta (Calypso Networks Association), Alok Jain (Trans Consult Ltd), Mark Godin (Trapeze Group), Jaroslaw Kowalski (Huawei), Joao Almeida (Card4B), Jaspal Singh (UITP), and Yossi Treistman (Jerusalem Light Railway). *DISTRIBUTED LEDGER TECHNOLOGY IN PUBLIC TRANSPORT: USE CASES FOR BLOCKCHAIN*. Mar. 2022. URL: `https://cms.uitp.org/wp/wp-content/uploads/2022/04/Report-Blockchain-in-PT-March2022-web.pdf` (visited on 11/04/2023) (cit. on p. 24).

[13] *Foundry*. URL: `https://github.com/foundry-rs/foundry` (visited on 11/04/2023) (cit. on p. 34).