

Alma mater studiorum - Università di Bologna

SCUOLA DI INGEGNERIA E ARCHITETTURA

Dipartimento di Informatica - Scienza e Ingegneria

Corso di Laurea Magistrale in Ingegneria Informatica

TESI DI LAUREA

in

Scalable and reliable services - M

Implementation of an ETS on the Solana Blockchain

CANDIDATO

Federico Pomponii

RELATORE

Michele Colajanni

Anno Accademico 2021/2022

1 - Emission trading systems	7
1.1 - Baseline and credit system	8
1.2 - Cap and trade system	8
1.3 - EU ETS	9
1.4 - Benefits and Challenges of using blockchain for ETS	10
2 - Blockchain	12
2.1 - What is a blockchain?	12
2.1.2 - Origin of the Blockchain	13
2.1.3 - The blockchain revolution	14
2.1.4 - Blockchain and Cryptocurrencies	15
2.2 - The Solana Blockchain	15
2.2.1 - Solana network	16
2.2.2 - Proof of History (PoH)	18
2.2.3 - What is Byzantine Fault Tolerance?	19
2.3 - Smart Contracts	21
2.3.1 - Smart Contracts within the Solana Ecosystem	23
2.3.2 - Accounts	24
2.3.3 - Transactions	25
2.3.4 - JSON-RPC	26
2.4 - Solana Tokens	27
3 - Rust Programming	30
3.1 - Working with Cargo	31
3.2 - Key Concepts in Rust	31
3.2.1 - Ownership	31
3.2.2 - Reference and Borrowing	33
3.3 - Solana smart contracts in Rust	34
4 - Project	36
4.1 - Client implementation	36
4.2 - Solana program deploy	56
5 - Conclusion	59
Bibliography	61

Considering the growth of blockchain-based technologies and decentralized applications this thesis aims to explore the potential of Solana, an open-source project implementing a new, high-performance, permissionless blockchain, and taking steps to bring the chain's footprint to zero. The Solana Blockchain, designed by Anatoly Yakovenko, stands out for its innovative approach to scalability. By utilizing a unique combination of breakthrough technologies, such as Proof-of-History (PoH), Tower BFT consensus, and high-performance architecture, Solana has achieved remarkable throughput and low-latency transaction processing. These key features make it an ideal platform for building complex and data-intensive applications, capable of handling thousands of transactions per second.

Specifically, the thesis will examine how Solana's features, such as its high transaction speed and low transaction fees, can facilitate the implementation and operation of a simplified version of an Emission Trading System.

An Emission Trading System, or ETS, is a market-based mechanism used to regulate and reduce greenhouse gas emissions. It involves the issuance, trading, and retirement of emission allowances or permits among participating entities, encouraging them to invest in cleaner technologies and practices. By introducing a Solana-powered ETS, we can explore how this blockchain's features can enhance the efficiency, transparency, and accessibility of the system.

Leveraging Solana's robust infrastructure, we can address scalability and real-time data and we can create an ETS that is both more efficient and user-friendly. The high-speed transaction processing and low fees on the Solana blockchain could significantly reduce the costs associated with trading emissions permits and enable seamless transactions at scale.

Furthermore, Solana's ability to handle large amounts of data in real-time can empower regulators, participants, and market observers to access up-to-date and reliable emission data. This transparency could facilitate accurate monitoring, reporting, and verification of emissions, improving compliance and accountability within the ETS. The decentralized nature of the Solana blockchain also ensures data integrity and immutability, adding an extra layer of trust and security to the system.

Through this thesis, I will explore the technical aspects, challenges, and potential benefits of implementing a simple Solana-based ETS. By conducting an analysis of the existing ETS frameworks, examining the capabilities of the Solana blockchain, and designing and implementing a prototype system, we can gain insights into the practicality and scalability of such an application. The outcome of this project is a simple implementation of a trading system, that could be used as a starting point for more detailed projects.

1 - Emission trading systems

The Emission Trading System (ETS) is a market-based mechanism designed to reduce greenhouse gas emissions. A broader use of an ETS would be one of the most efficient ways of promoting green growth.

Polluters who would find it costly to reduce their emissions are allowed to buy emission allowances from polluters that can abate at lower costs. In a ‘perfectly’ working market, the costs of reducing an additional unit of emissions would be equalized, and the total costs of reaching a given environmental target would be minimized.

There are two different main types of ETS, the “**Cap-and-trade**” and the “**baseline-and-credit systems**”.

The [European Union emissions trading system](#), or EU ETS, is currently the world’s largest system. It operates in all EU countries plus Iceland, Liechtenstein, and Norway, limiting emissions from more than 11,000 heavy users of energy including power stations and industrial plants, and airlines operating between the ETS member countries.

Under the third phase of the ETS, which runs from 2013–20, a single, centralized cap covering the whole EU was set. This cap is reduced year by year during phase 4 (2021-2030). The International Carbon Action Partnership (ICAP) estimates that emissions trading now [covers 15% of global emissions](#) [1]. Two of the main credit system type are the Baseline and credit system and the Cap and Trade system.

1.1 - Baseline and credit system

Baseline and credit schemes identify, measure and provide incentives for activities that reduce emissions. Under a baseline and credit scheme, an emissions intensity is set for emitting activities against a baseline, and credits are created for activities that achieve emissions intensities below the baseline. Activities that have emissions intensities above the baseline have to buy such credits. The ability to generate credits from emissions reductions relative to baseline and the pressure to avoid having to buy permits for emissions above the baseline provide incentives for participants to find lower emission production processes.

1.2 - Cap and trade system

The essence of cap-and-trade is encapsulated in its name: there is a "cap" or maximum limit on the total amount of certain pollutants (such as carbon dioxide) that can be emitted by certain industries or sectors, and the ability to "trade" emission permits. The cap is set by a regulatory authority and is reduced gradually over time with the ultimate goal of reducing total emissions.

Companies are issued to purchase tradable allowances, each equivalent to the right to emit a specific amount. Firms that reduce their emissions below their allowance level may sell or "trade" their excess allowances to firms that find it harder or more costly to make reductions. This market mechanism incentivizes companies to innovate and reduce their emissions cost-effectively.

The cap-and-trade principle first gained prominence in the U.S. during the late 20th century as a policy tool to mitigate acid rain. It achieved significant success by reducing sulfur dioxide emissions from power plants. The approach has since been applied to GHG emissions, with the European Union Emissions Trading Scheme (EU ETS) becoming the largest ETS system globally.

However, cap-and-trade systems are not without their critics. Issues around the allocation of allowances, the potential for market manipulation, and impacts on energy prices have all been raised. Additionally, the success of a cap-and-trade system depends on effective monitoring, reporting, and enforcement mechanisms.

1.3 - EU ETS

The EU ETS, a cap-and-trade system focused on CO₂, is the world's largest and first multicountry emissions trading system.

The EU ETS directive was adopted for the first time in 2003 and then launched in 2005. The cap on allowances was set at the national level through [national allocation plans](#) (NAPs).

The EU ETS is divided into distinct phases or trading periods. The first phase (2005-2007) was a learning period. The second phase (2008-2012) coincided with the first commitment period of the Kyoto Protocol. From the third phase (2013-2020) onward, the system was significantly reformed, including the introduction of auctioning as a default method

for allocating allowances, harmonized rules across the EU, and the expansion of the system to cover more sectors and gases.

The EU ETS provides a valuable case study in operating a large-scale cap-and-trade system. It illustrates both the potential of such systems to drive emission reductions and the challenges associated with their design and implementation.

1.4 - Benefits and Challenges of using blockchain for ETS

Blockchain technology has been proposed as a potential solution to enhance the performance and transparency of emissions trading systems (ETS), such as the EU ETS.

It can help to create a more transparent and auditable system, as all transactions are recorded on an immutable ledger that can be verified by anyone. Secondly, it can enable faster and more secure settlement of trades. Finally, it can help to reduce the administrative load of ETS, as it can automate many of the processes involved in allowance allocation and trading.

However, several challenges need to be addressed when using blockchain for ETS.

Scalability and performance are critical issues, as the system needs to be able to handle a large number of transactions in real-time.

Also, interoperability with existing ETS systems and market participants is important, as it ensures that the blockchain-based system can integrate with the existing market infrastructure. Thirdly, privacy and security are

key concerns, as sensitive information such as emissions data and allowance ownership needs to be protected from unauthorized access.

2 - Blockchain

2.1 - What is a blockchain?

A blockchain is a distributed ledger technology (DLT). It works like a database, with some differences, it is stored in various locations around the network and the world, called nodes. Each node processes transactions submitted by clients, that become committed records, called the ledger, of a replicated database on all nodes.

The difference from traditional DLT is that a Blockchain uses a consensus mechanism when records are committed to immutable ledgers. Each record is ordered in time, and each block of records is cryptographically linked to the prior committed block. The records are typically represented by a hash tree known as *Merkle Tree*.

When a new transaction is initiated, it is broadcast to all the nodes in the network. These nodes verify the transaction and add it to a new block, which is then added to the chain. Once a block is added to the chain, it cannot be altered or deleted, since doing so would require changing the hash of that block and all subsequent blocks in the chain.

Because of this immutability and decentralization, blockchains are often used to power cryptocurrencies such as Bitcoin, Ethereum, and Solana. However, they can also be used for a wide range of other applications,

including supply chain management, healthcare, finance, voting systems, and identity verification.

2.1.2 - Origin of the Blockchain

It's common to date the birth of the blockchain with the publication of the Bitcoin paper authored by Satoshi Nakamoto, in 2008. In the paper "Bitcoin: a peer-to-peer Electronic Cash System" [2], Satoshi Nakamoto, proposed a new payment system that solved the double-spending problem: the risk in digital currency systems where a user can spend the same amount of money more than once, undermining the integrity and trust of the system. It arises due to the inherently digital nature of transactions, which allows for easy replication and simultaneous use of the same fund. The solution, without using a third-party entity, was an ongoing chain of hash-based proof-of-work, forming a record that cannot be changed without redoing the proof-of-work [2].

While blockchain was initially designed to support Bitcoin, the potential of this technology soon became evident. The launch of Ethereum in 2015 introduced the concept of smart contracts, self-executing contracts with the terms of the agreement directly written into lines of code. This evolution extended the applicability of blockchain technology beyond cryptocurrencies to numerous sectors, including finance, supply chain management, healthcare, and energy. Some years later, In November of 2017, Anatoly Yakovenko published a whitepaper describing Proof of

History, a technique for keeping time between computers that do not trust one another. That whitepaper created the basis for the Solana blockchain, often called “The Ethereum Killer”.

2.1.3 - The blockchain revolution

Bitcoin is commonly described as Blockchain 1.0, a blockchain technology without the concept of Smart Contracts. After the release of the Ethereum paper people started to talk about Blockchain 2.0.

With the revolution introduced by Ethereum, interest from enterprise companies grew and IBM, Intel, Oracle, Linux Foundation and others started to invest in and develop tools for Blockchains.

The financial industry has been one of the earliest adopters of blockchain technology. With its ability to securely and transparently track transactions, the blockchain has the potential to revolutionize the way we transfer and manage money. For example, blockchain-based payment systems can eliminate the need for intermediaries like banks, reducing transaction costs and increasing speed and efficiency.

Blockchain can also enable the creation of decentralized finance (DeFi) platforms, which can provide financial services without the need for traditional financial institutions. These platforms can offer a range of services, like trading, in a decentralized and secure manner.

2.1.4 - Blockchain and Cryptocurrencies

The blockchain decentralized ledger can be utilized to store and validate currency transactions. All blockchain protocols have coins that represent value and can be traded (SOL, ETH, BTC ...).

Unlike traditional currencies, cryptocurrencies are not issued by a government or financial institution and are not backed by physical assets or commodities. Instead, their value is based on supply and demand in the market.

One of the defining characteristics of cryptocurrencies is their security and resistance to fraud and counterfeiting. Cryptocurrencies utilize advanced cryptographic techniques to secure transactions and prevent double-spending (the act of spending the same cryptocurrency more than once).

2.2 - The Solana Blockchain

Solana is a high-performance, open-source blockchain platform designed to support decentralized applications (dApps) and cryptocurrencies. It was founded in 2017 by Anatoly Yakovenko, a former Qualcomm engineer, and the mainnet beta was officially launched in 2020.

Anatoly watched as blockchain systems without clocks, such as Bitcoin and Ethereum, struggled to scale beyond 15 transactions per second worldwide when centralized payment systems such as Visa required peaks of 65,000 tps. Without a clock, it was clear they'd never graduate

to being the global payment system or global supercomputer most had dreamed them to be. When Anatoly solved the problem of getting computers that don't trust each other to agree on the time, he knew he had the key to bring 40 years of distributed systems research to the world of blockchain. The resulting cluster wouldn't be just 10 times faster, or 100 times, or 1,000 times, but 10,000 times faster, right out of the gate![7]

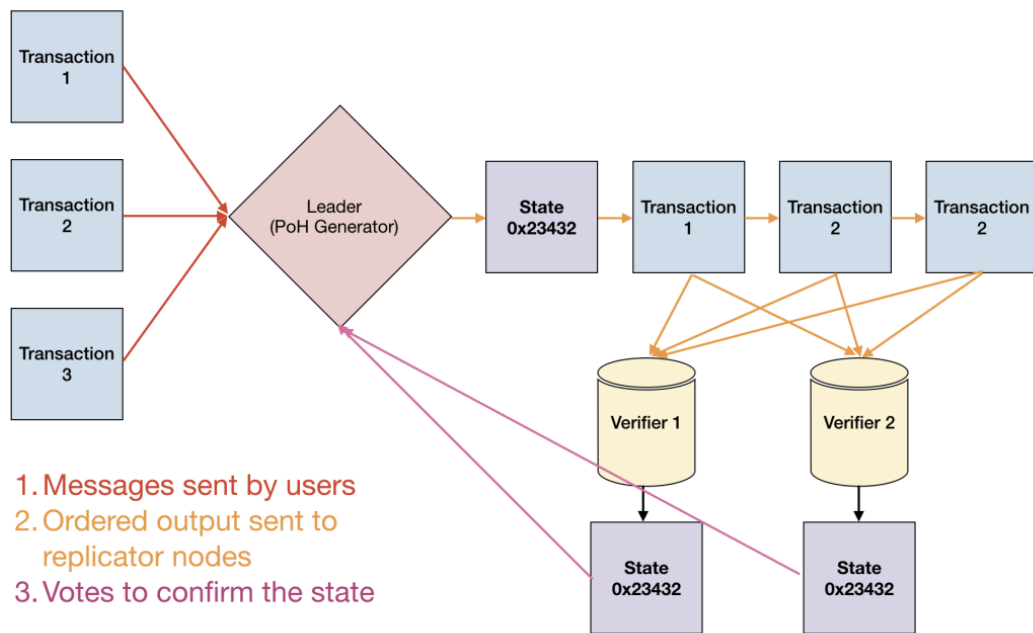
Solana's primary objective is to address the scalability and performance limitations of existing blockchain platforms while maintaining decentralization and security. Solana's unique architecture, which includes innovations such as Proof of History (PoH), Tower Byzantine Fault Tolerance (BFT) consensus mechanism, and a highly optimized network stack, enables the platform to process thousands of transactions per second (tps) with low latency and minimal fees. As a result, Solana has attracted a growing number of developers, projects, and investors seeking to leverage its capabilities for various use cases, including DeFi (decentralized finance), NFTs (non-fungible tokens), and gaming.

2.2.1 - Solana network

The Solana Blockchain proposes a new implementation of a blockchain based on the Proof Of History- PoH. The PoH is a mechanism that provides a proof of time between two events, using a cryptographic algorithm.

The proof of history, used alongside Proof of Stake, enhances the efficiency and resilience and can reduce messaging overhead in a Byzantine Fault Tolerant (class of failures that belong to the Byzantine Generals' Problem) replicated state machine, resulting in sub-second finality times [3].

In the Solana network at each time, a Node is responsible to generate a consistent Proof of History, as described in the following figure. Each node executes the transaction as is stored in the RAM and other nodes, called verifiers, replicate the same transaction with the confirmations as computed signatures, returning the same state.



2.1 Solana network design

2.2.2 - Proof of History (PoH)

Proof of history is a sequence of instructions cryptographically executed. The output of the function can then be re-computed and verified by external computers in parallel by checking each sequence segment on a separate core.

With a cryptographic hash function, whose output cannot be predicted without running the function (e.g. sha256, ripemd, etc.), run the function from some random starting value and take its output and pass it as the input into the same function again [3].

PoH Sequence		
Index	Operation	Output Hash
1	sha256("any random starting value")	hash1
2	sha256(hash1)	hash2
3	sha256(hash2)	hash3

2.2 - Solana PoH Sequence

This sequence of hashes can be used to timestamp functions and record that some data are recorded before some particular hash index is created. If some external events occur a function append is used to compute the new data recorded, the image below describes this workflow

POH Sequence

Index	Operation	Output Hash
1	sha256("any random starting value")	hash1
200	sha256(hash199)	hash200
300	sha256(hash299)	hash300
336	sha256(append(hash335, photograph1_sha256))	hash336
400	sha256(hash399)	hash400
500	sha256(hash499)	hash500
600	sha256(append(hash599, photograph2_sha256))	hash600
700	sha256(hash699)	hash700

2.3 - Solana PoH Sequence 2

We can see that based on the hash we can determine at which moment the photograph1 and photograph2 entered the chain.

2.2.3 - What is Byzantine Fault Tolerance?

Solana uses a variant of the Practical Byzantine Fault Tolerance (PBFT) protocol to achieve consensus [3].

Byzantine Fault Tolerance (BFT) is a critical feature of a blockchain network that allows it to reach consensus and continue to function correctly even if some nodes fail or act maliciously. The name comes from the Byzantine Generals' Problem, a situation where actors must coordinate their actions while dealing with potential traitors.

PBFT is a widely-used BFT protocol that is designed to tolerate Byzantine faults, which occur when nodes in the network behave maliciously or fail. In PBFT, a leader is elected to propose a block of

transactions, which is then validated by a group of nodes called validators. The validators communicate with each other to ensure that they all agree on the proposed block before it is added to the blockchain [4].

In Solana, the validators are selected based on their stake in the network, and they use PoH to establish a time-ordered sequence of events. The validators are responsible for validating transactions and adding them to the blockchain.

To ensure that the validators are behaving honestly, Solana uses a mechanism called gossip to detect Byzantine faults. Gossip is a process by which validators communicate with each other to exchange information about the state of the network. If a validator detects that another validator is behaving maliciously or is failing, it can broadcast this information to the other validators.

The other validators can then use this information to remove the malicious or failing validator from the network. Handling Byzantine Faults In Solana, if a validator behaves maliciously or fails, the other validators can detect this through gossip and remove the faulty validator from the network. When a validator is removed, its stake in the network is slashed, which incentivizes validators to behave honestly. If a validator has a high stake in the network, the penalty for malicious behavior is greater, which further incentivizes validators to act honestly. In addition,

Solana has a process for replacing faulty validators with new validators, which ensures that the network remains secure and operational.

Combining PoH and PoS, Solana can efficiently handle the 'Byzantine Generals' Problem' as it ensures that validators agree on the order of transactions (PoH) and the state of the network (PoS). Even if some nodes act maliciously or go offline, the rest of the network can continue processing transactions, maintaining the integrity and security of the blockchain.

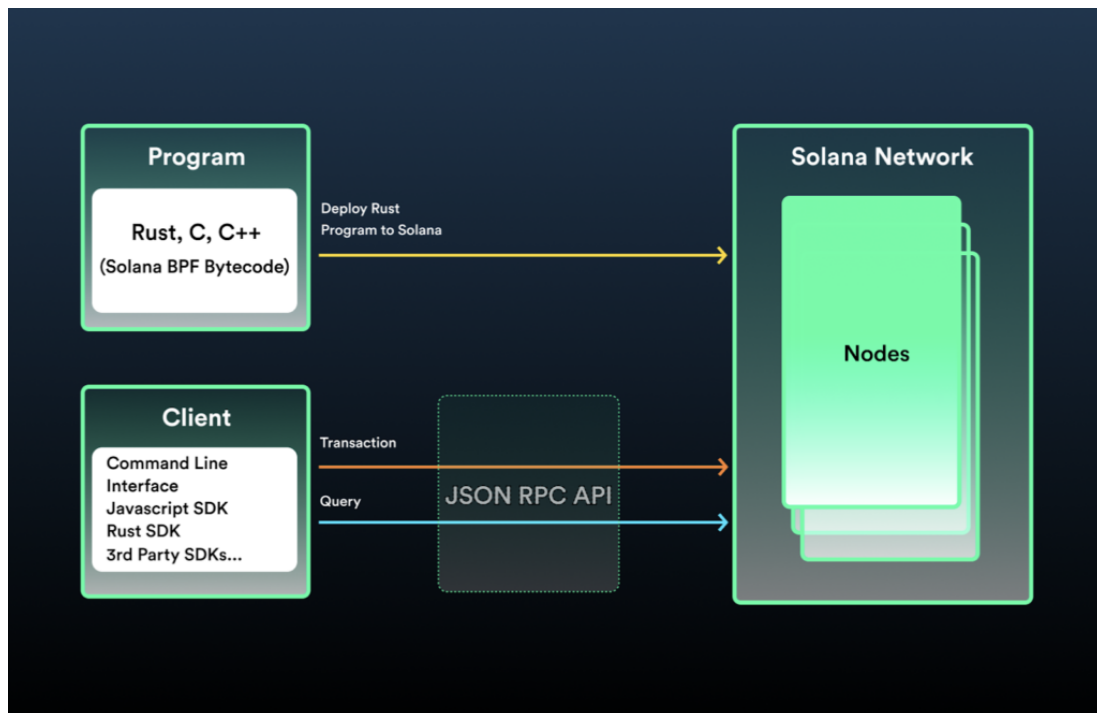
2.3 - Smart Contracts

Looking at the overall history of blockchain, it has been more than a decade since Bitcoin was proposed in the late 2000s. A few years after Bitcoin became the first and the biggest case of blockchain technology, the emergence of Ethereum immediately changed the research direction on blockchain at that time. Digital encryption currency moved into the programmable “era of smart contract”. Ethereum is a collection of protocols that provide an Ethereum Virtual Machine (EVM) that can execute any complex code and allow users to write their applications. Ethereum is a Turing completeness system compared to complex script in Bitcoin [8].

Unlike blockchains like Ethereum, where the heart of the system is the EVM, Solana offers different structural logic. The development of decentralized applications involves two dimensions – Program (creating

and deploying smart contracts called programs) and Client (writing Dapps to communicate with these deployed programs).

This diagram shows this division in the build process. Find out about three key distinctions of Solana below.



2.4 - Solana Smart Contracts overview

We can resume by saying that a smart contract is an executable that runs on the blockchain. It is used to facilitate, verify and enforce the negotiation between parties without the need for an intermediary.

Smart contracts can automate different operations like financial transactions, property transfer, supply chain management, and more.

They are particularly useful in situations where trust is an issue.

2.3.1 - Smart Contracts within the Solana Ecosystem

In Solana, smart contracts are referred to as 'programs', and they are written in either Rust or C. These contracts are open source and reside at an address on the blockchain. The terms 'smart contract' and 'program' are often used interchangeably in the Solana ecosystem.

A key aspect of Solana's architecture is that it treats smart contracts as shared libraries, pieces of code that any program can link to. This design allows Solana to execute transactions involving smart contracts much faster and more efficiently than other blockchains.

Rust is known for its safety and performance, making it well-suited for developing smart contracts that require high-speed execution and low latency [5].

Solana programs can be deployed to the core of the network as native programs or deployed on-chain by users. Programs are the core building of blocks and can be used for sending tokens between wallets, accepting votes of a DAO, or tracking ownership of NFTs.

There are plenty of Solana programs deployed on the net that any users can use and link to their programs, on this GitHub repository we can access the source code of that programs:

<https://github.com/solana-labs/solana-program-library>.

2.3.2 - Accounts

On a high level, memory inside a Solana cluster can be thought of as a monolithic heap of data and each smart contract have access to its part of that heap.

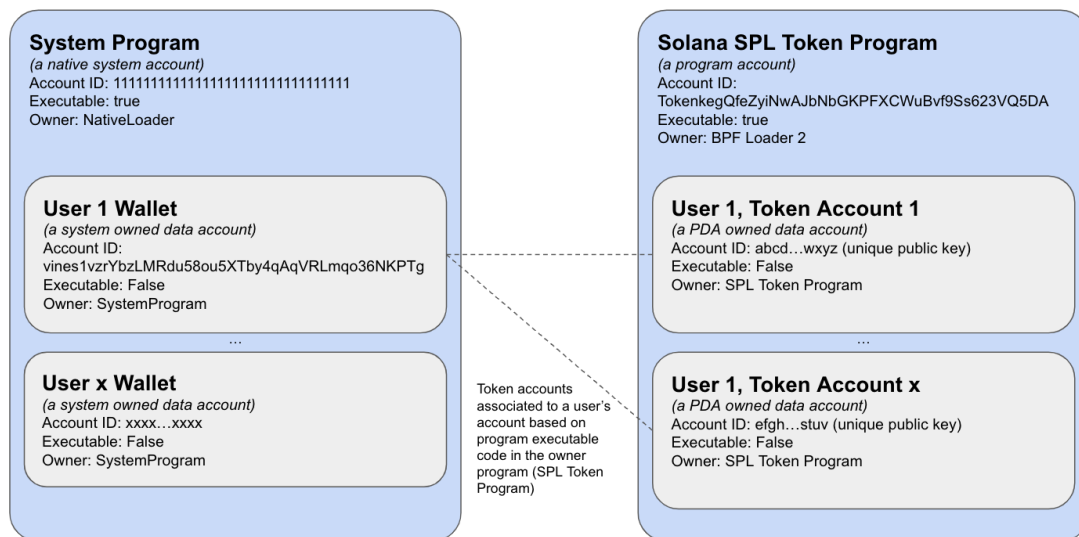
All state lives in this heap. Your SOL accounts, smart contracts, and memory that is used by smart contracts. Each memory region has a program that manages it (sometimes called the “owner”). The Solana term for a memory region is "account".

An Account is like a file in an operating system and it also includes metadata that tells the runtime who is allowed to access the data and how. If an account is marked "executable" in its metadata, then it is considered a program that can be executed by including the account's public key in an instruction's “program_id” [5]. Since all state lives in the heap, even programs themselves live there. Accounts that the BPFLoader owns store programs. This is a program that can be used to deploy and upgrade other programs. The Native Loader owns the BPFLoader and that is where the recursion ends.

The lifetime of each account is defined by the lamports owned by the account itself. A Lamport is a fraction of the SOL that can be used to “rent” a certain amount of time for the account. Every account that drops to zero lamports is purged and the only way to maintain an account active is to “rent-exempt” by charging a sufficient amount of lamports.

Transactions include one or more digital [signatures](#) each corresponding to an account address referenced by the transaction. Each of these

addresses must be the public key of an ed25519 keypair, and the signature signifies that the holder of the matching private key signed, and thus, "authorized" the transaction. In this case, the account is referred to as a signer [5].



2.5 - Solana Accounts overview

2.3.3 - Transactions

In Solana, each program's execution begins with a transaction being submitted to the cluster [5]. A transaction contains a compact array of signatures, followed by a message.

In Solana transactions, there are three essential components. First, signatures are required from all accounts authorizing any state update, but an account can sign the transaction once, even if it's updated multiple times within the transaction. Second, metadata includes information about the number of signing, non-signing, and read-only accounts, as well as a recent block hash. Finally, the instructions form the core of the transaction, specifying the accounts involved, the program ID to be

executed, and the data buffer containing relevant information. In Solana transactions, there are three essential components. First, signatures are required from all accounts authorizing any state update, but an account can sign the transaction once, even if it's updated multiple times within the transaction. Second, metadata includes information about the number of signing, non-signing, and read-only accounts, as well as a recent block hash. Finally, the instructions form the core of the transaction, specifying the accounts involved, the program ID to be executed, and the data buffer containing relevant information.

2.3.4 - JSON-RPC

Solana JSON-RPC is an application programming interface (API) that allows developers to interact with the Solana blockchain network using the JSON-RPC protocol. It provides a standardized way to send requests and receive responses in JSON format, facilitating communication between client applications and Solana nodes.

With Solana JSON-RPC, developers can access a wide range of functionalities and retrieve information from the Solana blockchain. These functionalities include querying blockchain state, submitting transactions, retrieving account balances, fetching transaction history, and more. It serves as a powerful tool for building decentralized applications (dApps), wallets, and other blockchain-related services on the Solana network.

To use Solana JSON-RPC, developers typically construct JSON-RPC requests specifying the method they want to invoke, along with any

required parameters. These requests are sent to a Solana node via HTTP or WebSocket connections. The Solana node processes the request and sends back a JSON response containing the requested data or confirmation of the operation.

Solana JSON-RPC supports various methods, each serving a specific purpose. For instance, the *getAccountInfo* method retrieves information about a specific account, while the *sendTransaction* method submits a new transaction to the network. By leveraging these methods and their associated parameters, developers can interact with the Solana blockchain programmatically and build sophisticated applications that leverage the blockchain's capabilities.

A common widely-used method to access the JSON-RPC API is the Solana Javascript SDK (<https://solana-labs.github.io/solana-web3.js/>). The SDK allows every application in JS to connect to the RPC endpoint and interact with the network.

2.4 - Solana Tokens

Solana tokens refer to digital assets or cryptocurrencies built on the Solana blockchain. Solana tokens can represent various types of assets, including utility tokens, security tokens, and non-fungible tokens (NFTs). They leverage the Solana blockchain's infrastructure and smart contract capabilities to enable secure and efficient token transfers and interactions.

Developers and projects can create their tokens on the Solana blockchain by utilizing Solana's smart contract language, which is based on Rust and

provides flexibility for creating customized token functionalities. These tokens can be used for various purposes, such as accessing dApps, participating in decentralized finance (DeFi) protocols, or representing unique digital assets in the form of NFTs.

Inside the context of smart contracts, the operation of creating new tokens is called mint. In the Solana Blockchain in order to mint or exchange tokens is needed to have an associated token account to the program or to a wallet.

That's because an user may own arbitrarily many token accounts belonging to the same mint which makes it difficult for other users to know which account they should send tokens to and introduces friction into many other aspects of token management. This program introduces a way to deterministically derive a token account key from a user's main System account address and a token mint address, allowing the user to create a main token account for each token they own. We call these accounts Associated Token Accounts.

In order to easily handle this operation Solana defines a common implementation for Fungible and Non-Fungible tokens, called *SPL-Token Program* [9].

The token program CLI can be installed using cargo (*see Chapter 3 for cargo and Rust details*), with the following script:

```
cargo install spl-token-cli
```

2.6 - Cargo install instruction

or is possible to use a Javascript SDK in order to use the Spl-Token program inside dApps (*like described in Chapter 4*).

The Spl-Token program offers different commands for many use cases, some of these are:

- *spl-token create-token*: allows to create a new Fungible Token and returns the signature of the newly created token.
- *spl-token mint [signature] [amount]*: mint the amount provided using the signature of the token.
- *spl-token accounts*: list all the tokens that the user owns
- *spl-token create-account [signature]*: create a new token associated with the users.

For a full reference of the spl-token program take a look at the documentation: <https://spl.solana.com/token>.

3 - Rust Programming

Rust is a high-performance systems programming language that offers a high level of safety against common programming errors, such as null pointer dereferencing and data races. It was first developed by Graydon Hoare at Mozilla Research in 2006 to provide memory safety without sacrificing performance, and it has seen growing adoption in industry and open-source projects since its 1.0 released in 2015.

One of the primary advantages of Rust is its emphasis on zero-cost abstractions, which means that it provides high-level functionalities without imposing a runtime overhead. Rust accomplishes this through a combination of compile-time checks and a unique ownership model.

The Rust compiler is renowned for its helpful error messages. The compiler does more than just check the syntax; it also ensures that memory use is safe across the entire program, tracking who owns what data and preventing many common bugs that would be runtime errors in other languages.

Rust's ownership model is one of its defining features. This model governs how Rust programs manage memory and resources. It consists of three main concepts: ownership, borrowing, and lifetimes. This ownership model eliminates entire classes of bugs associated with null or dangling pointers, double-free, and concurrent data access.

Moreover, Rust supports a hybrid of imperative procedural, concurrent actor, object-oriented, and pure functional styles. It also supports generic programming and metaprogramming, in both static and dynamic styles.

Lastly, the Rust ecosystem is growing rapidly. With its built-in package manager, Cargo, and the online repository Crates.io, it is easier than ever to find libraries (referred to as "crates" in the Rust community) for virtually any task.

3.1 - Working with Cargo

Cargo is Rust's build system and package manager. It is responsible to build the code, download libraries and maintain the dependencies of a project. All the settings and configuration of Cargo are written in a file at the root of every project, the *Cargo.toml*.

Relative to Solana development, Cargo is very useful because it allows developers to import libraries such as the **solana_program**.

3.2 - Key Concepts in Rust

In Rust, as in many other low-level programming languages, there are common concepts such as variables mutability, data types, functions, and structs, but the main three concepts are *Ownership*, *Borrowing*, and *Lifetimes*.

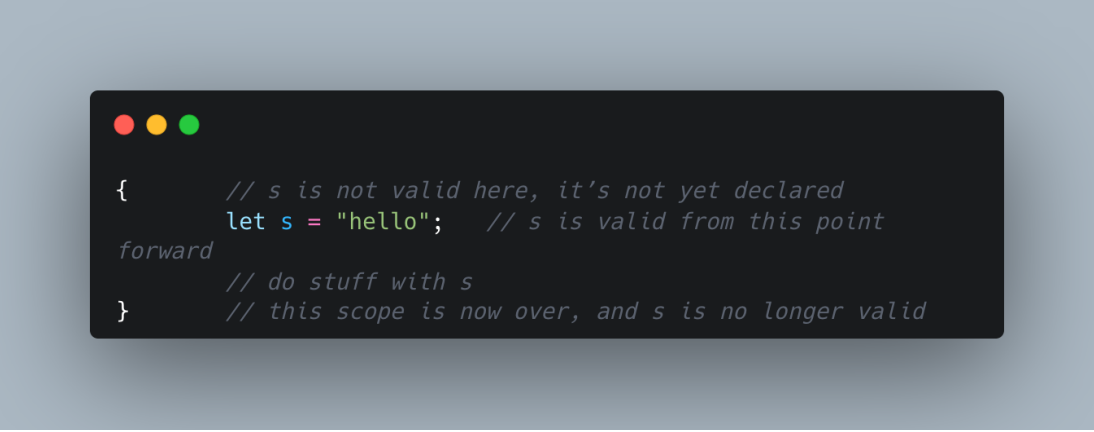
3.2.1 - Ownership

Ownership is Rust's most unique feature and has deep implications for the rest of the language. It enables Rust to make memory safety guarantees without needing a garbage collector.

All programs have to manage the way they use a computer's memory while running. Some languages have garbage collection that regularly looks for no longer-used memory as the program runs; in other languages, the programmer must explicitly allocate and free the memory. Rust uses a third approach: memory is managed through a system of ownership with a set of rules that the compiler checks. If any of the rules are violated, the program won't compile. None of the features of ownership will slow down your program while it's running [6].

Every value in Rust has a single 'owner' at a specific moment. When the owner goes out of scope, the value is automatically deallocated and its memory is reclaimed. This ensures that memory is managed efficiently, and there are no memory leaks.

The following snippet will highlight how the ownership of a variable is related to its scope (closure).



```
{ // s is not valid here, it's not yet declared
  let s = "hello"; // s is valid from this point
  forward
  // do stuff with s
}
```

3.1 - Rust ownership example

3.2.2 - Reference and Borrowing

If a function or another part of the code needs to temporarily access a value without taking ownership, Rust allows us to *reference* that value by using the “&” keyword. A reference is like a pointer in that it’s an address we can follow to access the data stored at that address; that data is owned by some other variable. Unlike a pointer, a reference is guaranteed to point to a valid value of a particular type for the life of that reference. We call the action of creating a reference *borrowing* [6].

There are two types of borrowing: mutable and immutable. Immutable borrowing means that multiple parts of the code can read the value simultaneously, but none can modify it. Mutable borrowing allows only one part of the code to access and modify the value, but no other part can access it at the same time. This ensures that data races and concurrent modifications are prevented.

When ownership of a value is transferred from one variable to another, the original variable can no longer be used to access the value. This is called 'moving' the value. Moving ensures that there is always a single owner for a value, and once the ownership is transferred, the original owner loses access to prevent double-free errors.

A terminal window with a dark background and three colored window control buttons (red, yellow, green) at the top left. The code is written in a light-colored font with syntax highlighting. The code defines a `main` function that creates a mutable string `s` with the value "hello" and then calls a `change` function. The `change` function takes a mutable string reference and appends " world" to it.

```
fn main() {  
    let mut s = String::from("hello");  
  
    change(&mut s);  
}  
  
fn change(some_string: &mut String) {  
    some_string.push_str(", world");  
}
```

3.2 - Rust ownership transfer example

These rules are checked by the Rust compiler at compile time, which helps guarantee memory safety and improve code quality without introducing a runtime performance penalty.

3.3 - Solana smart contracts in Rust

According to the Solana docs, Solana allows developers to write smart contracts in C/C++ or Rust.

Before developing smart contracts we need to pull in the *solana-program* library (<https://crates.io/crates/solana-program>).

TBC

4 - Project

The goal of this thesis is to explore the potentiality of the Solana blockchain applied to a real scenario such as the ETS. The outcome of this work is a project written in React (<https://react.dev/>) and served with Next.JS (<https://nextjs.org/>), a Server-side rendering framework widely used.

4.1 - Client implementation

The client of the application allows users to interact with different programs (smart contracts) deployed on the Solana devnet network.

To allow users to interact with the Smart Contracts the application is secured by an authentication layer with auth0 (<https://auth0.com/>) that allows users to use the Private Keys stored in a third-party tool called web3auth (<https://web3auth.io/>), in this way the application allows users to login with SSO or email and password and use a non-custodial wallet for the key pair.

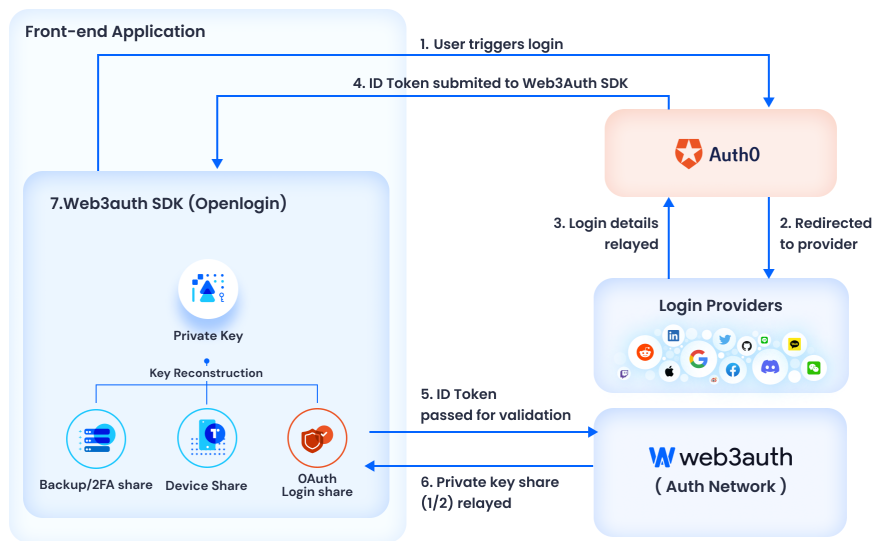


Figure 4.1 - Project authentication overview

The dashboard of the application allows users to see the actual balance and to trade custom tokens that are backed by the carbon price.

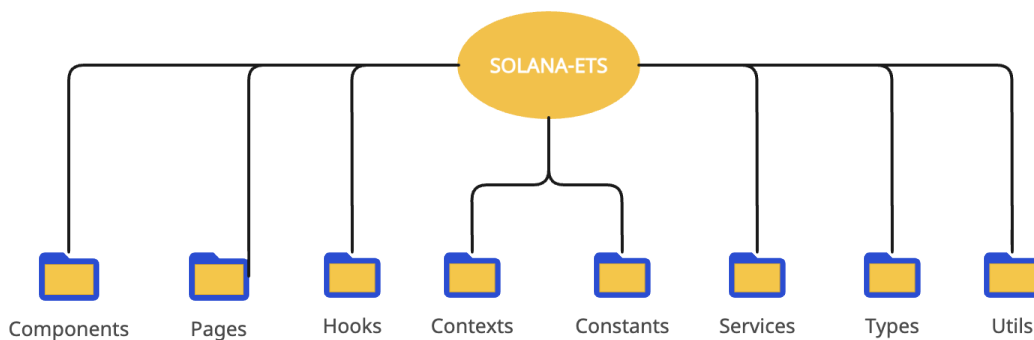


Figure 4.2 - Project structure

The project is splitted into features packages, as described in Figure 4.2. The entry point of the project is the `pages/_app.tsx` file, it instantiates the root of the application and defines layout and styles. All the logic related to the wallet interactions is in the `ApplicationContextProvider`, which

wraps the entire application (Fig. 4.3). Under the *components* folder there are all the modules re-usable inside the application, such as atoms, cards, headers, containers, and so on...

The *utils* folder contains all the code utilities like functions for formatting public keys, formatting class names of the application, and other valuable functions. The *types* folder contains all the Typescript types used in the application. The *hooks* folder has inside all the React hooks used in the project, like the *useProfile.ts* hook.

```
import { ApplicationContextProvider } from '@/contexts/ApplicationContext';

import '@/styles/globals.css';
import { NextPage } from 'next';
import type { AppProps } from 'next/app';
import { Inter } from 'next/font/google';
import { ReactElement } from 'react';

export type NextPageWithLayout = NextPage & {
  // eslint-disable-next-line no-unused-vars
  getLayout?: (page: ReactElement | ReactElement[]) => any;
};

type AppPropsWithLayout = AppProps & {
  Component: NextPageWithLayout;
};

const inter = Inter({ subsets: ['latin'] });

const App = ({ Component, pageProps }: AppPropsWithLayout) => {
  const getLayout = Component.getLayout || ((page) => page);

  return (
    <ApplicationContextProvider>
      <main className={inter.className}>
        {getLayout(<Component {...pageProps} />)}
      </main>
    </ApplicationContextProvider>
  );
};

export default App;
```

Figure 4.3 - *_app.tsx* file

The `ApplicationContextProvider` uses the React `useContext` hook and allows to reuse logic and the wallet state all around the application. The state exported by this context is described in Figure 4.4.

```
type ApplicationState = {
  user?: Partial<OpenLoginUserInfo>;
  tokenAccount?: Account;
  balance?: number;
  walletPublicKey?: PublicKey;
  transactions: ETSTokenTransaction[];
  company?: Company;
  onLogin: () => void;
  onLogout: () => void;
  // eslint-disable-next-line no-unused-vars
  buyTokens: (amount: number) => void;
  // eslint-disable-next-line no-unused-vars
  sellTokens: (amount: number) => void;
  openBuyTokenDialog: () => void;
  openSellTokenDialog: () => void;
  // eslint-disable-next-line no-unused-vars
  updateCompany: (obj: { name: string; address: string }) => void;
  // eslint-disable-next-line no-unused-vars
  createCompany: (obj: { name: string; address: string }) => void;
};
```

Figure 4.4 - Application Context State

As we can see in the image above the `ApplicationContext` exports the login and logout functions, which are implemented using the `web3Auth` + `Auth0` and then standard pieces of information regarding the blockchain interaction, like `tokenAccount`, `balance`, and the `transferTokens` function.

To setup the web3 authentication a `Web3AuthNoModal` object is instantiated, as shown in Figure 4.5.

```
const web3Auth = new Web3AuthNoModal({
  clientId,
  web3AuthNetwork: 'testnet',
  chainConfig: {
    chainNamespace: CHAIN_NAMESPACES.SOLANA,
    chainId: '0x3',
  },
});
```

Figure 4.5 - Web3AuthNoModal implementation

and to interact with the auth0 verifier for the login with a JWT token (Figure 4.6).

```
const openloginAdapter = new OpenloginAdapter({
  adapterSettings: {
    clientId,
    uxMode: 'popup',
    loginConfig: {
      jwt: {
        verifier: 'auth-0-verifier',
        typeOfLogin: 'jwt',
        clientId: AUTH_0_CLIENT_ID,
      },
    },
  },
});
```

Figure 4.6 - OpenLoginAdapter implementation

then is possible to implement the login flow as in the function onLogin (Figure 4.7).

```
const onLogin = useCallback(async () => {
  let isAuthenticated = false;

  try {
    console.debug('[WEB 3] - authenticateUser');
    await web3Auth.authenticateUser();
    isAuthenticated = true;
    return;
  } catch (e) {
    console.error('[WEB 3] - error on authenticateUser', e);
  }

  try {
    await web3Auth.connectTo(WALLET_ADAPTERS.OPENLOGIN, {
      loginProvider: 'jwt',
      extraLoginOptions: {
        domain: AUTH_0_DOMAIN,
        verifierIdField: 'email',
      },
    });
    isAuthenticated = true;
  } catch (e) {
    console.error(e);
  } finally {
    if (isAuthenticated) {
      console.debug('[WEB 3] - connected to wallet, redirect to dashboard');
      // Redirect window to dashboard
      window.location.href = '/dashboard';
    }
  }
}, []);
```

Figure 4.7 - Login flow inside the onLogin callback

At this point, after that the user login using the modal provided by auth0, the application is ready to use the SolanaWallet provider and starts to sign transactions using the key pair of the user. Transactions are used to

interact with the smart contracts of the application. The most important smart contract of this project is the one used for exchange tokens, I'll provide info about its implementation in Rust later in this chapter. There are two functions exported by the `ApplicationContext`: *buyTokens* (Figure 4.8) and *sellTokens* (Figure 4.9).

Breaking down the implementation of these two functions we can understand how the communication with the smart contract works.

```

const buyTokens = useCallback(
  async (amount: number) => {
    if (!connection) {
      console.error('[TX TOKEN] connection not initialized yet');
      return;
    }
    if (!web3Auth.provider) {
      console.error('[TX TOKEN] provider not initialized yet');
      return;
    }
    if (!walletPublicKey) {
      console.error('[TX TOKEN] walletAccount not initialized yet');
      return;
    }

    if (!tokenAccount?.address) {
      console.error('[TX TOKEN] tokenAccount not initialized yet');
      return;
    }

    const provider = new SolanaWallet(web3Auth.provider);
    const blockhash = (await connection.getRecentBlockhash('finalized'))
      .blockhash;

    const tIx = await setupBuyTokensIx({
      walletPK: walletPublicKey,
      userTokenAccount: tokenAccount?.address,
      amount: (LAMPORTS_PER_SOL / 100) * amount,
    });

    const tx = new Transaction();
    tx.add(tIx);
    tx.feePayer = walletPublicKey;
    tx.recentBlockhash = blockhash;
    tx.sign(ADMIN_WALLET);

    await provider.signAndSendTransaction(tx);
  },
  [connection, walletPublicKey, tokenAccount?.address]
);

```

Figure 4.8 - buyTokens callback implementation

```

const sellTokens = useCallback(
  async (amount: number) => {
    if (!connection) {
      console.error('[TX TOKEN] connection not initialized yet');
      return;
    }
    if (!web3Auth.provider) {
      console.error('[TX TOKEN] provider not initialized yet');
      return;
    }
    if (!walletPublicKey) {
      console.error('[TX TOKEN] walletAccount not initialized yet');
      return;
    }

    if (!tokenAccount?.address) {
      console.error('[TX TOKEN] tokenAccount not initialized yet');
      return;
    }

    const provider = new SolanaWallet(web3Auth.provider);
    const blockhash = (await connection.getRecentBlockhash('finalized'))
      .blockhash;

    const tIx = await setupSellTokensIx({
      walletPK: walletPublicKey,
      userTokenAccount: tokenAccount?.address,
      amount: (LAMPORTS_PER_SOL / 100) * amount,
    });

    const tx = new Transaction();
    tx.add(tIx);
    tx.feePayer = ADMIN_WALLET.publicKey;
    tx.recentBlockhash = blockhash;
    tx.sign(ADMIN_WALLET);

    await provider.signAndSendTransaction(tx);
  },
  [connection, walletPublicKey, tokenAccount?.address]
);

```

Figure 4.9 - sellTokens callback implementation

By looking at the implementation of the `buyTokens` and `sellTokens` functions we can see that using the `SolanaWallet` is possible to sign and send a `Transaction` object in which is added a `TransactionInstruction`, as shown in Figure 4.10.

```
const tx = new Transaction();
tx.add(tIx);
tx.feePayer = ADMIN_WALLET.publicKey;
tx.recentBlockhash = blockhash;
tx.sign(ADMIN_WALLET);

await provider.signAndSendTransaction(tx);
```

Figure 4.10 - `SolanaWallet` provider sign and send a transaction

Taking a look at the `buyTokens` function we can see that to build the `TransactionInstruction` the method `setupBuyTokensIx` is called.

Inside the `setupBuyTokensIx` (Figure 4.11), is defined the transactions as requested by the smart contract for the exchange. Keys for the wallet and the token accounts are provided to allow the smart contract to transfer the exact amount provided in the `TransferInstruction`.

The `programId` field inside the `TransferInstruction` tells the network the program that the transaction is interacting with, in this specific scenario, is the public key of the exchange smart contract deployed on the network.

To allow the smart contract to decode the informations provided, Borsh is used to serialize the payload, using the same struct defined in the program (Figure 4.12). Borsh is a serialization hashing format specifically designed for the Rust programming language. It provides a way to convert Rust data structures into a binary format that can be easily stored or transmitted.

The name "Borsh" stands for "Binary Object Representation Serializer for Hashing." It focuses on simplicity, compactness, and speed. Borsh aims to provide an efficient serialization solution for Rust applications, especially for blockchain and decentralized application (dApp) development.

```

export const setupBuyTokensIx = async ({
  walletPK,
  userTokenAccount,
  amount,
}: TransferTx): Promise<TransactionInstruction> => {
  const programId = new PublicKey(PROGRAM_ID);

  const payload = new TransferInstruction({ amount });
  const adminAccount = new PublicKey(ADMIN_ACCOUNT_KEY);
  const data = borsh.serialize(TransferInstructionSchema, payload);

  const ix = new TransactionInstruction({
    keys: [
      {
        pubkey: walletPK,
        isSigner: true,
        isWritable: true,
      },
      {
        pubkey: adminAccount,
        isSigner: true,
        isWritable: true,
      },
      {
        pubkey: new PublicKey(ETS_TOKEN_ACCOUNT),
        isSigner: false,
        isWritable: true,
      },
      {
        pubkey: userTokenAccount,
        isSigner: false,
        isWritable: true,
      },
      {
        pubkey: TOKEN_PROGRAM_ID,
        isSigner: false,
        isWritable: false,
      },
      {
        pubkey: SystemProgram.programId,
        isSigner: false,
        isWritable: false,
      },
    ],
    programId,
    data: Buffer.concat([Buffer.from(new Uint8Array([2])), Buffer.from(data)]),
  });

  return ix;
};

```

Figure 4.11 - setupBuyTokensIx method implementation


```

export class TransferInstruction {
  amount = 0;
  constructor(fields: { amount: number } | undefined = undefined) {
    if (fields) {
      this.amount = fields.amount;
    }
  }
}

export const TransferInstructionSchema = new Map([
  [TransferInstruction, { kind: 'struct', fields: [['amount', 'u64']] }],
]);

```

Figure 4.12 - Borsh serialization on the client

After analyzing the client is important to take a look at the smart contract implementation to understand how it works.

In Figure 4.13 is shown the code of the instructions interface of the program.

```

use borsh::{BorshDeserialize, BorshSerialize};

#[derive(BorshSerialize, BorshDeserialize)]
2 implementations
pub enum CompanyInstruction {
  Create { name: String, address: String, created_at: u64 }, // 0
  Update { name: String, address: String, updated_at: u64 }, // 1
  ExchangeTokens { amount: u64 }, // 2
}

```

Figure 4.13 - Smart contract instruction interface

This interface means that the program is processing three different types of instructions and that the function for exchange tokens is the one with index 2, if we look at the implementation of the *setupBuyTokensIx* (Figure 4.11) we can see that in the *TransactionInstruction* a

Buffer.from(new Uint8Array([2])) is passed: in that way, the contract knows that the transaction is for the ExchangeTokens action.

Then by looking at the implementation of the smart contract processor, we can get information regarding all the accounts and the signers needed for the program to be executed.

As described in the figure 4.14 the program needs 6 Account info to be executed:

- `wallet_ai`: the wallet in which we want to transfer tokens.
- `admin_ai`: the wallet that will receive the SOL and will exchange tokens, in this case, is called `admin_ai` because is the wallet owned by the ETS.
- `src_token_account_ai`: the Account that owns the tokens, associated with the admin wallet
- `dst_token_account_ai`: the Account owned by the wallet which stores the tokens.
- `token_program_ai`: this is the token program account info, needed for calling the tokens transfer instruction.
- `system_program`: the Solana system program account info, needed for the execution of the SOL transfer instruction.

The same order of accounts is provided in the TransactionInstruction in the *setupBuyTokensIx* function (Figure 4.11).

```

CompanyInstruction::ExchangeTokens { amount: u64 } => {
  msg!("Transfer company account V2...");
  let account_info_iter: &mut Iter<'_, AccountInfo<'>> = &mut accounts.iter();

  let wallet_ai: &AccountInfo<'> = next_account_info(account_info_iter)?;
  let admin_ai: &AccountInfo<'> = next_account_info(account_info_iter)?;
  let src_token_account_ai: &AccountInfo<'> = next_account_info(account_info_iter)?;
  let dst_token_account_ai: &AccountInfo<'> = next_account_info(account_info_iter)?;
  let token_program_ai: &AccountInfo<'> = next_account_info(account_info_iter)?;
  let system_program: &AccountInfo<'> = next_account_info(account_info_iter)?;

  if !wallet_ai.is_signer {
    return Err(ProgramError::MissingRequiredSignature);
  }
}

```

Figure 4.14 - ExchangeTokens processor implementation

Going deeper into the implementation in figure 4.15 there is the implementation of the transfer instruction. As we can see we're calling a specific Solana program included in the SPL (Solana Program Library), responsible to transfer tokens from a source to a destination.

Once the token transfer instruction is instantiated it is invoked using the invoke program function provided by Solana (figure 4.16)

```

let transfer_tokens_instruction: Instruction = spl_token::instruction::transfer(
  token_program_id: &token_program_ai.key,
  source_pubkey: src_token_account_ai.key,
  destination_pubkey: dst_token_account_ai.key,
  authority_pubkey: admin_ai.key,
  signer_pubkeys: &[admin_ai.key],
  tk_transfer_amount,
)?;

```

Figure 4.15 - spl_token transfer instruction implementation

```

invoke(
    &transfer_tokens_instruction,
    account_infos: &[
        src_token_account_ai.clone(),
        dst_token_account_ai.clone(),
        admin_ai.clone(),
    ],
)?;

```

Figure 4.16 - spl_token instruction invoke

And at this point, the exact amount of token requested is traded for the corresponding SOL amount needed.

The other due instructions exposed by the smart contract are the Create and Update. These two functions allow to create and/or update a Company profile. The behaviour of the interaction between the client and the program is almost the same as the ExchangeTokens, with the same differences that we can see by looking at the program implementation (Figure 4.17, 4.18) and the Company state (Figure 4.19). To serialize and deserialize data inside the Company object is used Borsh in Rust. The *try_from_slice_unchecked* allows deserializing an object using the Company struct. On the other hand, the serialize function serializes the Company struct.

```

CompanyInstruction::Create {name: String, address: String, created_at: u64}>= {
  msg!("Create company account V2, setting init values...");
  let account_info_iter: &mut Iter<'_, AccountInfo<'_>> = &mut accounts.iter();

  let wallet_account: &AccountInfo<'_> = next_account_info(account_info_iter)?;
  let pda_account: &AccountInfo<'_> = next_account_info(account_info_iter)?;
  let system_program: &AccountInfo<'_> = next_account_info(account_info_iter)?;

  if !wallet_account.is_signer {
    return Err(ProgramError::MissingRequiredSignature);
  }
  // GENERATE ACCOUNT INFO PDA
  let (_pda: Pubkey, bump_seed: u8) =
    Pubkey::find_program_address(seeds: [&wallet_account.key.as_ref()], program_id: _program_id);
  msg!("PDA key: {}", _pda);
  let len: usize = 4 + 100 + 4 + 100 + 8 + 8;

```

Figure 4.17 - Create instruction implementation

```

CompanyInstruction::Update { name: String, address: String, updated_at: u64 } => {
  msg!("Update company account V2...");
  let account_info_iter: &mut Iter<'_, AccountInfo<'_>> = &mut accounts.iter();
  let wallet_account: &AccountInfo<'_> = next_account_info(account_info_iter)?;
  let pda_account: &AccountInfo<'_> = next_account_info(account_info_iter)?;

  if !wallet_account.is_signer {
    return Err(ProgramError::MissingRequiredSignature);
  }

  msg!("Deserializing account");
  let mut company: Company =
    try_from_slice_unchecked::<Company>(data: &pda_account.data.borrow_mut()[..]) Result<Company, Error>
    .unwrap();

```

Figure 4.18 - Update instruction implementation

```

use borsh::{BorshDeserialize, BorshSerialize};

#[derive(BorshSerialize, BorshDeserialize)]
2 implementations
pub struct Company {
    pub name: String,
    pub address: String,
    pub created_at: u64,
    pub updated_at: u64,
}

```

Figure 4.19 - Company State

The most important part of the Company instructions is that for storing company informations the program is using a Program Derived Address. Program Derived Addresses (PDAs) are home to accounts that are designed to be controlled by a specific program. With PDAs, programs can programmatically sign for certain addresses without needing a private key. To understand the concept behind PDAs, it may be helpful to consider that PDAs are not technically created, but rather found. PDAs are generated from a combination of seeds and a program id. This combination of seeds and program id is then run through a sha256 hash function to see whether or not they generate a public key that lies on the ed25519 elliptic curve (Figure 4.20).

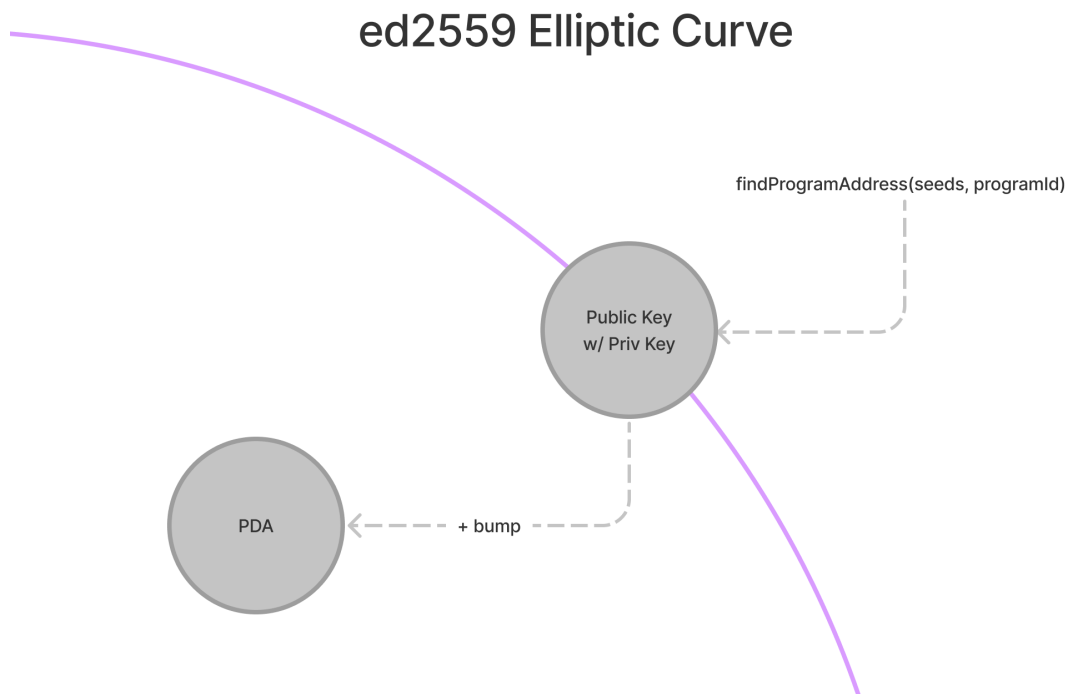


Figure 4.20 - Pda generation from the elliptic curve

Using a PDA allows the program to store information based on the public key of the user, in this way, after the login a user can provide and retrieve information regarding his company.

In the client, the function that allows interaction with the PDA is the *getCompanyPDA* callback described in Figure 4.21.

It uses borsh to perform a `deserializeUnchecked` operation using the `BorshCompanySchema` struct (Figure 4.22)

```

const getCompanyPDA = useCallback(
  async (connection: Connection, walletPublicKey: PublicKey) => {
    const { pda } = await generatePda(
      new PublicKey(PROGRAM_ID),
      walletPublicKey
    );

    const accountInfo = await connection.getAccountInfo(pda);
    if (!accountInfo) throw new Error('Company account not found');

    const company: Company = deserializeUnchecked(
      BorshCompanySchema,
      BorshPayload,
      accountInfo?.data
    ) as Company;

    setCompany(company);
  },
  []
);

```

Figure 4.21 - Retrieve information about the PDA in the client


```

import { BorshPayload } from '@serialization/borsh';

export type Company = {
  name: string;
  created_at: bigint;
  updated_at: bigint;
  address: string;
};

export const BorshCompanySchema = new Map([
  [
    BorshPayload,
    {
      kind: 'struct',
      fields: [
        ['name', 'string'],
        ['address', 'string'],
        ['created_at', 'u64'],
        ['updated_at', 'u64'],
      ],
    },
  ],
]);

```

Figure 4.22 - Retrieve information about the PDA in the client

4.2 - Solana program deploy

This chapter covers the steps needed in order to develop and deploy the Solana program

First of all we need to setup the local environment as described in the Solana quick start (<https://docs.solana.com/getstarted/local>).

Then a file system wallet is needed, in order to create a new one we can run the script:

```
solana-keygen new
```

and then create a new keygen and config as the default wallet in the local file system with the script:

```
solana config set -k ~/.config/solana/id.json
```

Once that the local environment is ready we need to generate the byte-code of the program, in order to do that we can run the following script inside the program folder:

```
cargo build-sbf
```

Once that the BPF is compiled the program can be deployed with the script:

```
solana program deploy ./target/deploy/{PROGRAM_NAME}.so
```

At this point, the public key returned by the CLI is the `program_id` that needs to be called inside the client of the application.

Is possible to see all the transactions and the status of the program on the network using the Solana explorer (<https://explorer.solana.com/?cluster=devnet>), using the address of the deployed program.

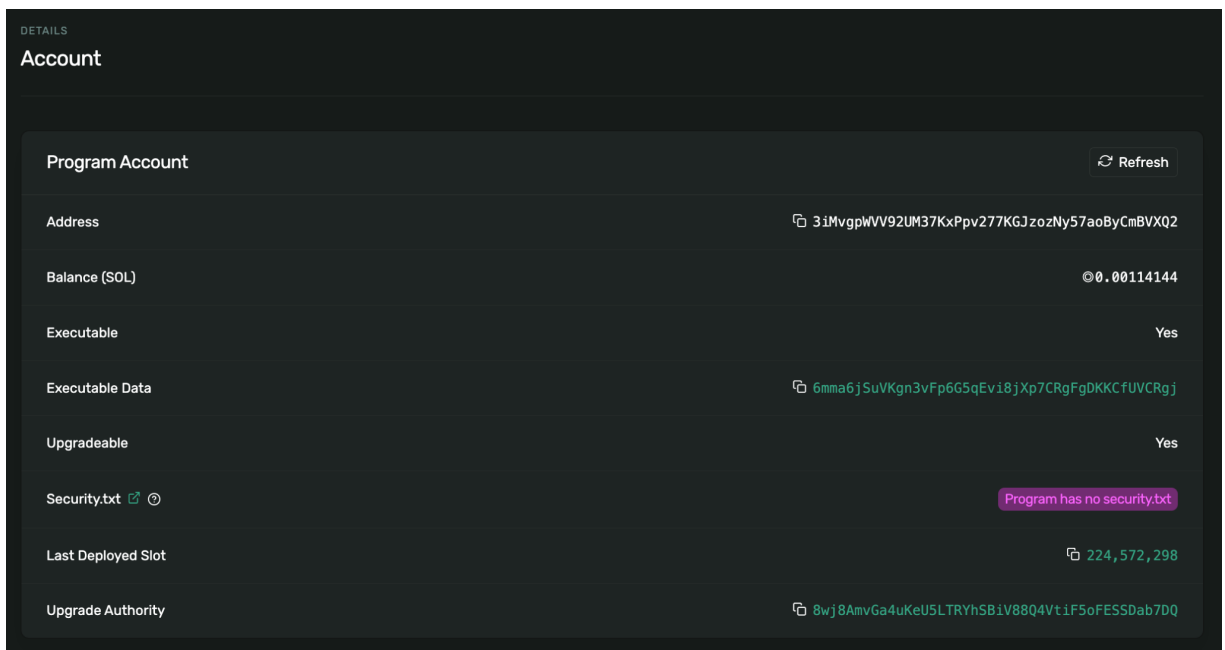


Figure 4.23 - Smart contract details on the Solana explorer

5 - Conclusion

This research aimed to identify the key features of the Solana Blockchain and the benefits to use a blockchain infrastructure in a real scenario such as the Emissions Trading System. The outcome of this thesis is a decentralized application that allows users to interact with the blockchain inside the context of an ETS. The high number of transactions per second, low carbon impact, and average lower fees compared to the main competitor like Ethereum, make Solana the perfect choice for a system like the ETS.

The use of Rust for the Smart Contracts development is a good win for Solana, Rust is a common development language nowadays, so is easy to deep into program development for developers, without learning a new language (like Solidity for Ethereum).

I really liked the Program Derived Addresses, it allows to organize accounts into smart contracts in a different way that makes Solana a very powerful blockchain for dApps.

In the end, I think that this work could be a good starting point for people that want to explore the Solana network and build something on top of it, in this thesis there is a good example of a non-custodial authentication and the integration of the smart contracts inside a Next.js application. Starting from this is possible to adapt these technologies for many different use cases and projects.

The project is uploaded on GitHub at the link:

<https://github.com/pmpwith2i/solana-ETS>

Bibliography

- [1]- [How do emissions trading system work - LSE UK](https://www.lse.ac.uk/granthaminstitute/explainers/how-do-emissions-trading-systems-work/)
(<https://www.lse.ac.uk/granthaminstitute/explainers/how-do-emissions-trading-systems-work/>)
- [2] - [Bitcoin: A Peer-to-Peer Electronic Cash System](https://bitcoin.org/bitcoin.pdf)
(<https://bitcoin.org/bitcoin.pdf>)
- [3] - [Solana: A new architecture for a high performance blockchain](https://solana.com/solana-whitepaper.pdf)
(<https://solana.com/solana-whitepaper.pdf>)
- [4] - [Practical BFT Research \(Microsoft\)](https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2001-12.pdf)
(<https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2001-12.pdf>)
- [5] - [Solana Docs](https://docs.solana.com/) (<https://docs.solana.com/>)
- [6] - [The Rust Programming Language](https://doc.rust-lang.org/book/) (<https://doc.rust-lang.org/book/>)
- [7] - [History | Solana Docs](https://docs.solana.com/history) (<https://docs.solana.com/history>)
- [8] - [Research and Application of Smart Contract Based on Ethereum Blockchain](https://www.researchgate.net/publication/348822434_Research_and_Application_of_Smart_Contract_Based_on_Ethereum_Blockchain)
(https://www.researchgate.net/publication/348822434_Research_and_Application_of_Smart_Contract_Based_on_Ethereum_Blockchain)
- [9] - [Solana - SPL Token Program Docs](https://spl.solana.com/token) (<https://spl.solana.com/token>)

Alla fine di questo elaborato un ringraziamento speciale va al professore Michele Colajanni, grazie per avermi dato la possibilità di approfondire un tema come quello della Blockchain, svolgere questa tesi è stato davvero interessante sia per le conoscenze acquisite che per le sfide affrontate.

Grazie alla mia famiglia, i miei amici, colleghi e tutte le persone che ho avuto la fortuna di incontrare in questi anni.

Grazie a Claudia, vivere questo periodo con te è stata la cosa più bella che mi sia mai capitata.