LISBON SCHOOL OF ENGINEERING

Department of Electronical Engineering, Telecommunications and Computers

Ticket management system using Blockchain technology

Rodrigo Filipe Leitão Dias

Bachelor's

Project Work to obtain the masters degree
in Informatics and Multimedia Engineering

Adviser : PhD Carlos Gonçalves

Jury:

President: [Grau e Nome do presidente do juri]

Vogal: [Grau e Nome do primeiro vogal]

September, 2024

LISBON SCHOOL OF ENGINEERING

Department of Electronical Engineering, Telecommunications and Computers

Ticket management system using Blockchain technology

Rodrigo Filipe Leitão Dias

Bachelor's

Project Work to obtain the masters degree
in Informatics and Multimedia Engineering

Adviser : PhD Carlos Gonçalves

Jury:

President: [Grau e Nome do presidente do juri]

Vogal: [Grau e Nome do primeiro vogal]

September, 2024

Abstract

The traditional ticketing industry faces challenges such as ticket scalping, fraud, and limited transparency in secondary markets. Blockchain technology has the potential to revolutionize ticketing by offering a secure, transparent, and efficient solution. This thesis proposes a blockchain-based ticketing system that leverages the core strengths of blockchain technology to address these shortcomings.

The system utilizes smart contracts to manage the ticket lifecycle securely, from creation by event organizers to purchase and transfer by users. This ensures authenticity and eliminates the risk of counterfeiting. Additionally, the system facilitates a secure and transparent secondary market for ticket resale, with fair pricing mechanisms and clear ownership tracking.

Resumo

Contents

List of Figures

List of Tables

1

Introduction

Concerts and festivals play a big role in peoples life's, allowing them to create memorable experiences watching live performances from their favorite artists. Those are the kinds of memories that last for life, so every event organizer wants to ensure that the whole process works seamlessly, from the end user to the entire background planning of the event.

The process of organizing an event starts with the event organizer, who is responsible for the whole planning of the event, from the venue to the artists, to the marketing and ticketing. The event organizer is the one who takes the risk of organizing the event, and the one who will profit from it, and can be a company, a group of people, or even a single person, whom will hire the artists, the venue, the security, the marketing, and the ticketing.

However, there's an issue with the ticketing process, which is the scalping. Scalping is the process of buying tickets in bulk, exploiting high demand, and reselling them at significantly inflated prices. This not only disadvantages people that genuinely want to attend but also undermines the integrity of the ticketing system. Traditional ticketing platforms often rely on centralized databases and intermediaries, providing opportunities for scalpers to manipulate the system and engage in fraudulent activities. Moreover, existing ticketing systems frequently encounter issues related to security, trust, and reliability. Centralized databases are vulnerable to cyber attacks, leading to unauthorized access, data breaches, and the manipulation of ticketing information. Trust in the authenticity of tickets and the reliability of transactions is compromised, creating a

pressing need for innovative solutions that can address these inherent challenges.

That's where blockchain comes in. Blockchain technology, renowned for its decentralized and transparent nature, presents a compelling solution to revolutionize the ticketing industry. By leveraging blockchain, it becomes possible to create a secure and tamper-proof ledger of transactions, mitigating the risk of scalping and ensuring the integrity of the ticketing process. The use of smart contracts further automates transactions, reducing the reliance on intermediaries, therefore extra costs, and enhancing operational efficiency. This allows the event organizer to have a more secure and reliable ticketing process, and the end user to have a more transparent and fair ticketing process.

1.1 Motivation

The motivation for this work is primarily to avoid ticket scalping. By using blockchain, it's possible to create a system that prevents any kind of unwanted operations because of the properties of smart contracts that enforce a certain behavior, allowing for users to resell a ticket, but not for a price higher than the original price. This is a way to guarantee that the end user will have a fair and transparent ticketing process. For this to happen, the idea is to have a marketplace where users are able to resell their tickets, enforcing a price cap on them.

Another important feature is the possibility of having partial refunds. This is something that is not always available in traditional ticketing platforms, and when it is, it's not easy to do. With blockchain, it's possible to have a system that allows for easy and instant refunds, without the need for intermediaries. Enabling a feature like this can actually be advantageous for the organizers, if they're expecting the venue to be full. This way, when users buy their tickets and then realize they can't attend, they have a reason to refund, making that ticket available again for the original price, making a profit for the organizer.

Another point, and the main reason to use blockchain, is to guarantee the user of any operation. In the traditional ticketing system, there can be human errors, or even fraud, and this assures users that any operation defined will never change and will be executed as expected.

1.2 Objectives

We will have a website that will allow Ticketchain to approve organizers into our system to be able create events, so that no random entity can create freely, which would lead to spam. This would also be for the event organizers themselves where, if allowed, they would be able to manage their events, and manage admins and validators associated to them.

There will also be an app for the users to check the events and manage their tickets. The events shown are gonna be the ones stored in our Ticketchain system. They will also have access to the marketplace, where they will be able to resell their tickets for a price no higher than the original one.

For the validators, we will have yet another app that allows them to validate user tickets at the entrance of the venue. These validators are selected by the organizer for the event, and their job is to truthfully verify the tickets, without any chance of fraud. These validators can be either a person with their app to operate or even some automated machine, like a turnstile.

1.3 Contributions

[user and validator app] 

1.4 Document Structure

2

Background and Related Work

This chapter presents the background and related work for the system. The background in the Section 2.1 will present the necessary information, like the blockchain concepts and the fundamentals of how the entire system works. The related work in the Section 2.2 will present projects with similarities to the system and how NFTs are more commonly used.

2.1 Background

Blockchain is a fairly recent technology and can a lot of times be hard to understand. It leverages cryptographic concepts at its core to make it all work. In this section, we will explain some of the key concepts of blockchain technology and also discuss how wallets work and how they interact with the blockchain. We will then explore some of the most popular blockchain networks and token standards, as well as the emerging trend of non-fungible tokens (NFTs). The Figure 2.1 illustrates the most common concepts of blockchain technology and we'll be mentioning the Wallets (1) in the Section 2.1.3 and what they are for, the Networks (2) in the Section 2.1.4 and what differences exist between the most popular ones, and the Smart Contracts (3) in the Section 2.1.5 and their characteristics.

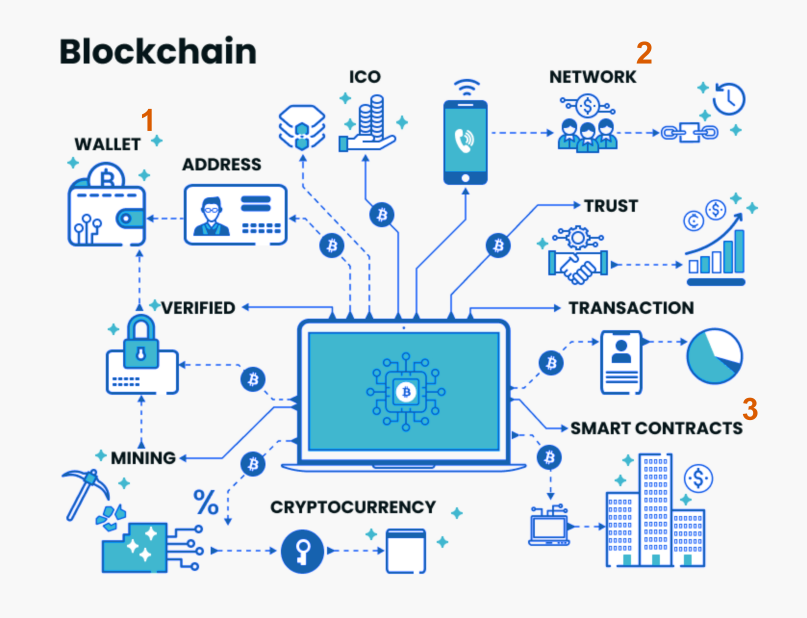

Figure 2.1: Blockchain concepts. Adapted from ...

2.1.1 Interacting with the Blockchain

As we saw, a blockchain works as a decentralized network of nodes, where each node has a copy of the entire blockchain. This means that in order to interact with the blockchain, we need to send transactions to the network, which will then be validated and added to a block by the nodes. To do this, we need to use a wallet, which is an application that allows users to manage their digital assets, interact with smart contracts, and send transactions on the blockchain. Wallets provide a user-friendly interface for accessing the blockchain network, signing transactions with private keys, and viewing account balances and transaction history.

The Figure 2.2 shows the steps of what happens on a transaction on the blockchain. To execute one, we need to sign it with our private key, which proves that we are the rightful owner of the assets being transferred. The transaction is then broadcast to the network, where it is validated by network participants and added to a block. Once the transaction is confirmed and included in a block, it becomes part of the immutable blockchain ledger, visible to all participants in the network.

This is the procedure to make a change to the blockchain state, so you always have to sign the transaction. To read information from it, you don't need to sign anything, you just need to query the network for the information you want.

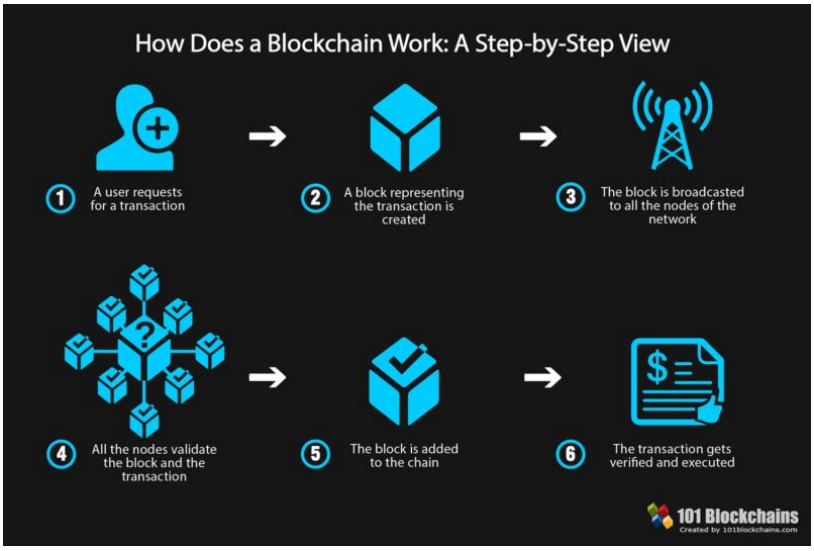

Figure 2.2: How does a blockchain work. Extracted from ...

2.1.2 Blockchain

So blockchain is a decentralized and distributed ledger technology that enables the secure recording and sharing of data across a network of computers. At its core, a blockchain consists of a series of blocks, each containing a list of transactions. These blocks are linked together in a chronological and immutable chain, forming a transparent and tamper-proof record of transactions.

Key characteristics of blockchain technology include:

Decentralization: Unlike traditional centralized systems where data is stored in a single location or controlled by a central authority, blockchain operates on a decentralized network of computers (nodes). Each node maintains a copy of the entire blockchain, ensuring that there is no single point of failure or control.

Transparency: The data recorded on a blockchain is visible to all participants in the network. This transparency fosters trust among users, as they can independently verify the integrity of transactions and the state of the ledger without relying on intermediaries.

Immutability: Once a transaction is recorded on the blockchain and added to a block, it becomes virtually impossible to alter or delete. This immutability is achieved through

cryptographic techniques such as hashing and consensus mechanisms, ensuring that the historical record of transactions remains tamper-proof.

Security: Blockchain technology employs advanced cryptographic algorithms to secure transactions and protect the integrity of the network. Transactions are verified and validated by network participants through a process known as consensus, which prevents fraudulent or unauthorized changes to the ledger.

Smart Contracts: Smart contracts are self-executing contracts with predefined rules and conditions written in code. These contracts automate the execution of transactions and enforce agreements without the need for intermediaries. Smart contracts enable the creation of decentralized applications (DApps) that run on blockchain networks, facilitating a wide range of use cases beyond simple monetary transactions.

Blockchain technology has applications across various industries, including finance, supply chain management, healthcare, and decentralized finance (DeFi). Its potential to revolutionize existing systems by enhancing security, transparency, and efficiency has led to widespread adoption and exploration of its capabilities in solving complex challenges. Some people refer to this ecosystem as Web3, which is a new paradigm for the internet that aims to decentralize control and empower users with greater ownership and privacy over their data and digital assets.

A great discussion topic as for how a system like this works is because it leverages the human nature of greed and self-interest to create a system that is secure and reliable. The network of nodes is incentivized to maintain the integrity of the blockchain by rewarding them with cryptocurrency for their efforts. As long as these cryptocurrencies have value, this creates a system where the majority of the network is honest and works together to maintain the integrity of the blockchain, making it resistant to attacks and fraud, as the cost of attacking the network would far outweigh any potential gains of joining it.

2.1.3 Wallets

Cryptocurrency wallets rely on cryptographic principles to securely manage and interact with digital assets on blockchain networks. These cryptographic techniques ensure the security and integrity of transactions while protecting the private keys that control access to cryptocurrency holdings.

Some of the key cryptographic aspects of wallets are:

Private and Public Keys: Cryptocurrency wallets utilize a pair of cryptographic keys: a public key and a private key. The public key, also known as the wallet address, is used to receive funds and is shared publicly. The private key, on the other hand, is known only to the wallet owner and is used to sign transactions and authorize the spending of funds. The relationship between the public key and the private key is based on asymmetric cryptography, where data encrypted with one key can only be decrypted with the other key. This ensures that transactions are secure and that only the rightful owner of the private key can access and control their cryptocurrency holdings.

Digital Signatures: When a transaction is initiated from a cryptocurrency wallet, it is digitally signed using the wallet's private key. This digital signature serves as proof of authorization and ensures that the transaction cannot be tampered with or altered. Digital signatures are generated using cryptographic algorithms such as the Elliptic Curve Digital Signature Algorithm (ECDSA) or the Rivest-Shamir-Adleman (RSA) algorithm, depending on the specific blockchain network and protocol.

Hash Functions: Cryptocurrency wallets use cryptographic hash functions to create a unique representation of transaction data, known as a transaction hash. These hash functions generate fixed-length strings of characters from input data, making it computationally infeasible to reverse-engineer the original data from the hash. Transaction hashes are essential for verifying the integrity of transactions and ensuring that they have not been altered or tampered with during transmission.

Seed Phrases and Mnemonic Codes: Some cryptocurrency wallets use mnemonic codes or seed phrases as a backup mechanism for restoring access to wallet funds in case the original private key is lost or compromised. These seed phrases are generated from a random sequence of words and serve as a human-readable representation of the wallet's private key. They can be used to regenerate the private key and restore access to funds on a new wallet instance.

By leveraging these cryptographic techniques, cryptocurrency wallets provide a secure and reliable means for users to store, manage, and transact with digital assets on blockchain networks. The robustness of these cryptographic mechanisms ensures the

confidentiality, integrity, and authenticity of transactions, safeguarding the value of cryptocurrency holdings against unauthorized access and fraudulent activities.

2.1.4 Networks

This technology has evolved significantly since the inception of Bitcoin in 2009. Numerous platforms have emerged, each offering unique features, capabilities, and use cases.

Some of the most prominent networks that have gained traction in the decentralized ecosystem are:

Bitcoin (BTC): Bitcoin is the first and most well-known cryptocurrency, introduced by an anonymous person or group of people under the pseudonym Satoshi Nakamoto in 2008. It operates on a decentralized network using a Proof of Work (PoW) consensus mechanism to validate transactions and secure the network. Bitcoin is designed as a peer-to-peer electronic cash system, enabling users to send and receive payments without the need for intermediaries. It has gained widespread adoption as a store of value and digital currency, with a fixed supply of 21 million coins and a deflationary monetary policy.

Ethereum (ETH): Ethereum is a decentralized, open-source blockchain platform that enables the creation and execution of smart contracts and decentralized applications (DApps). It introduced the concept of smart contracts, allowing developers to build a wide range of decentralized applications, from decentralized finance (DeFi) protocols to non-fungible token (NFT) marketplaces. Ethereum operates on a Proof of Work (PoW) consensus mechanism but is transitioning to a Proof of Stake (PoS) consensus model with the Ethereum 2.0 upgrade to improve scalability and energy efficiency.

Polygon (MATIC): Polygon is a Layer 2 scaling solution for Ethereum, designed to address the network's scalability issues by offering faster and cheaper transactions. It provides a framework for building and connecting Ethereum-compatible blockchain networks, known as sidechains, which leverage the security of the Ethereum mainnet. Polygon aims to enhance Ethereum's capabilities and support the mass adoption of decentralized applications by improving scalability, reducing transaction costs, and enhancing user experience.

Solana (SOL): Solana is a high-performance blockchain platform designed for decentralized applications and crypto-currencies. It uses a unique combination of Proof of History (PoH) and Proof of Stake (PoS) consensus mechanisms to achieve high throughput and low latency, enabling fast transaction speeds and low fees. Solana aims to provide a scalable infrastructure for decentralized finance (DeFi), decentralized exchanges (DEXs), and other high-performance applications.

These are just a few examples of the diverse range of blockchain networks that exist, each offering unique features, capabilities, and use cases. As the blockchain ecosystem continues to evolve, new networks and technologies are constantly being developed, driving innovation and expanding the possibilities of decentralized applications and digital assets.

It is common to see other networks with a similar behavior as the Ethereum and Polygon ones because they are built using Solidity. That makes them compatible with the Ethereum Virtual Machine (EVM), making it convenient for developers to deploy their smart contracts on multiple networks with the same codebase. This is a great advantage for developers, as it allows them to reach a wider audience and leverage the network effects of multiple blockchain platforms. Some other EVM compatible networks worth mentioning are the Binance Smart Chain, Avalanche, Arbitrum, and Optimism, among others.

2.1.5 Smart Contracts

Smart contracts are self-executing contracts with the terms of the agreement directly written in code. These contracts automatically execute and enforce themselves when predefined conditions are met, without the need for intermediaries such as lawyers or notaries. Smart contracts run on blockchain platforms and are stored and executed across a decentralized network of nodes.

Key characteristics of smart contracts include:

Autonomy: Once deployed on the blockchain, smart contracts operate autonomously, executing transactions and enforcing agreements without human intervention. This autonomy ensures that contract terms are upheld impartially and transparently.

Trust: Smart contracts leverage the trustless nature of blockchain technology, meaning that parties can trust the execution of the contract without relying on a trusted third party. The decentralized and immutable nature of blockchain ensures that contract terms are tamper-proof and transparent.

Security: Smart contracts are highly secure due to the cryptographic principles underlying blockchain technology. Once deployed, smart contracts cannot be altered or tampered with, providing a high level of security and reliability.

Efficiency: By automating contract execution, smart contracts eliminate the need for intermediaries, reducing costs and processing times associated with traditional contract enforcement. Transactions are executed quickly and efficiently, enhancing the overall speed and efficiency of business processes.

Versatility: Smart contracts can be programmed to execute a wide range of functions beyond simple transaction processing. They can facilitate complex conditional agreements, manage digital assets, and even interact with other smart contracts, enabling the development of decentralized applications (DApps) with diverse functionalities.

Smart contracts have numerous applications across various industries, including finance, supply chain management, real estate, healthcare, and more. They are particularly well-suited for scenarios where trust, transparency, and automation are paramount, offering a revolutionary approach to contract execution and enforcement in the digital age.

2.1.6 Token Standards

Token standards play a crucial role in defining the rules and functionalities of digital tokens on blockchain networks. These standards provide a common framework that facilitates interoperability, compatibility, and ease of use for developers and users alike.

Some of the most widely recognized token standards in the blockchain ecosystem are the ERC-20, ERC-721, and ERC-1155 standards. We can see in the Figure 2.3 a visual representation of these standards, where ERC-20 is the standard for fungible tokens, the ERC-721 is the standard for non-fungible tokens, and the ERC-1155 is a hybrid standard that supports both fungible and non-fungible tokens within the same contract.

In a videogame perspective, this means that ERC-20 is used as the currency of the game and ERC-721 is used for unique items. And since ERC-1155 covers both cases, it's usually the option for the game developers to use it as the main token standard.

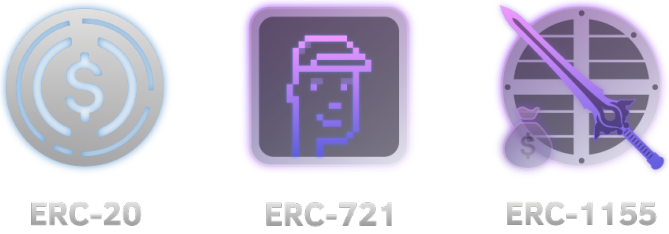

Figure 2.3: Token standards. Extracted from ...

In a more detailed explanation for each of these standards:

ERC-20 (Ethereum Request for Comment 20): ERC-20 is the most commonly used token standard on the Ethereum blockchain, governing the creation and implementation of fungible tokens. These tokens are interchangeable and have identical properties, allowing them to be traded on cryptocurrency exchanges seamlessly. ERC-20 tokens adhere to a set of standard functions, including methods for transferring tokens, querying token balances, and approving token transfers on behalf of other addresses. Many of the initial coin offerings (ICOs), token sales, and decentralized finance (DeFi) projects on Ethereum utilize ERC-20 tokens due to their widespread adoption and compatibility with Ethereum wallets and exchanges.

ERC-721 (Ethereum Request for Comment 721): ERC-721 is a token standard on the Ethereum blockchain that governs the creation and implementation of non-fungible tokens (NFTs). Unlike ERC-20 tokens, each ERC-721 token is unique and indivisible, representing ownership or proof of authenticity of a specific asset. ERC-721 tokens are commonly used to represent digital assets such as digital art, collectibles, virtual real estate, and in-game items. Each token is assigned a unique identifier (token ID), allowing it to be distinguished from other tokens within the same contract. The ERC-721 standard defines methods for transferring tokens, querying token ownership, and managing metadata associated with each token, enabling a wide range of use cases in the burgeoning NFT market.

ERC-1155 (Ethereum Request for Comment 1155): ERC-1155 is a token standard on the Ethereum blockchain that supports the creation and management of both fungible and non-fungible tokens within the same contract. This allows developers to efficiently manage multiple token types and reduce gas costs associated with deploying multiple contracts. ERC-1155 tokens are highly flexible and versatile, making them suitable for a wide range of applications, including gaming, digital collectibles, and decentralized finance (DeFi). They provide developers with the ability to create tokenized assets with varying degrees of uniqueness and scarcity. The ERC-1155 standard defines methods for transferring tokens, querying token balances, and managing batch transfers of multiple token types, offering enhanced functionality compared to previous token standards.

These token standards represent just a few examples of the diverse range of standards shaping the landscape of tokenization on blockchain networks. As blockchain technology continues to evolve, new standards are likely to emerge, offering innovative solutions and driving further adoption of digital tokens across various industries.

2.1.7 Non-Fungible Tokens (NFTs)

Since we're talking about a ticketing system for events, we can see a lot of potential in the use of NFTs to represent digital tickets, providing a secure and verifiable means of ticket issuance, transfer, and validation. NFT-based tickets can be associated with unique metadata, such as event details, seat numbers, and access permissions, providing a rich and customizable ticketing experience for event organizers and attendees. NFTs can also be used to create limited edition or VIP tickets, offering exclusive access and additional benefits to holders of these special tickets.

2.2 Related Work

In this chapter, we will present some of the most popular ticket selling platforms and how they work. We will also discuss the application of NFTs in the art and gaming industries.

2.2.1 Traditional Ticket Selling Platforms

There are many different traditional ticket selling platforms that are used today. Ticketline and Blueticket are the largest and most reputable Portuguese companies specializing in ticket sales for a variety of events, including concerts, sports games, theater productions, and exhibitions. They also have a wide range of physical outlets, like Worten, Fnac, and El Corte Inglés, making it convenient to buy tickets in person.

They have a website where they list all the events organizers are selling tickets for. The user can then open the event they are interested in and check all the details about it and it allows for the user to buy the tickets online and print them at home, or directs them to a physical outlet where they can buy them in person.

2.2.2 Application of NFTs

A few of the most popular applications of NFTs are in the art and gaming industries. In the art industry, NFTs are used to create digital art pieces that are unique and can be bought and sold. In the gaming industry, NFTs are used to create unique in-game items that can be bought, sold, and traded between players. The most popular project is the Bored Ape Yacht Club which is the famous ape *jpegs* that have been sold for millions of dollars. The project has a community of people who own these apes and they are used to allow access to exclusive events and merchandise.

3

Requirements Analysis

This chapter will present the requirements analysis for the system and is divided into two sections: use cases in the Section 3.1, and the requirements in the Section 3.2. The use cases section will present the use cases for the system while the requirements section will present the functional requirements and non-functional requirements.

[add architecture]

3.1 Use Cases

This section will present the use cases for the system, divided into four actors: system owner, organizer, validator, and user. All of them have one similar use case, the authenticate use case, which is responsible for authenticating the users on the system. On the Subsection 3.1.1 we have the use cases for the ~~system~~ owner, where it mentions the management of the system settings and the control of the organizers that have access to the system. On the Subsection 3.1.2 we have the use cases for the organizer that mention the creation and management of the events, along with the control of the validators for the event. On the Subsection 3.1.3 we have the use cases for the validator, and the only thing he can do is to validate the users' tickets. Lastly, on the Subsection 3.1.4 we have the use cases for the common user, like the purchase, gift, refund and resell of tickets.

3.1.1 System owner use cases

As we can see in the Figure 3.1, the system owner has the specific use case to control event organizers. This is important because the he needs to have control over the organizers that have access to the system. The system owner can also manage the system settings, which is important to control the system's behavior and to adapt it to the organizers' needs.

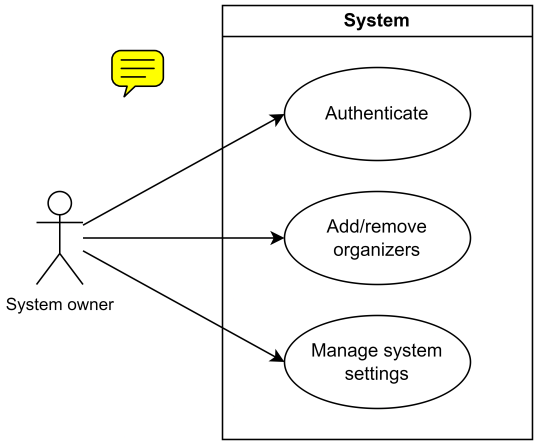

Figure 3.1: System owner use cases

3.1.2 Organizer use cases

The organizer has the use cases to create events, as we can see in the Figure 3.2. He can also control the validators for the event, which is important to select the people that have the authority to validate the tickets for each event. The organizer can also manage the event settings, like updating the event information or cancel it if really needed.

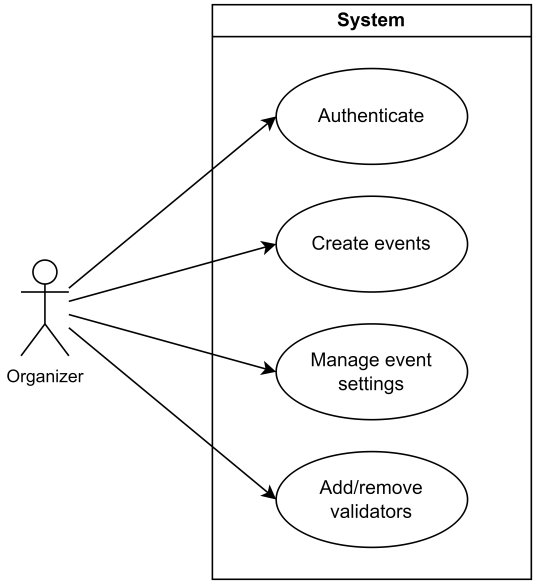

Figure 3.2: Organizer use cases

3.1.3 Validator use cases

For the validators, as we see in the Figure 3.3, the only use case is to validate the users' tickets and to allow them to enter the event. This is a necessary step to avoid users to try to bypass this security measure and to ensure that only the users that have a valid ticket can enter the event.

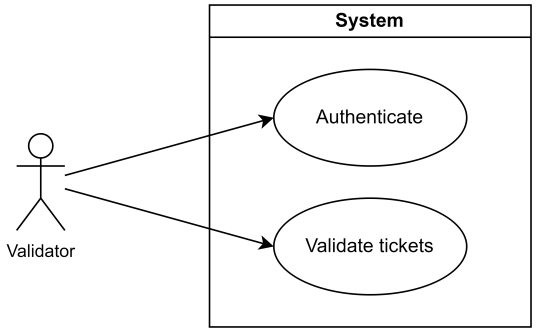

Figure 3.3: Validator use cases

3.1.4 User use cases

In the Figure 3.4 we see the use cases for the common user. The user can purchase tickets for the events, gift tickets to other users, refund tickets if he doesn't want to go

to the event anymore (depending on the configuration of the specific event), and resell tickets if he wants to sell them to other users (with the guarantee that he can't sell at a higher price than the original). All of these use cases are important to give the user the flexibility to manage his tickets and have the freedom to do what he wants with them, within the system's rules, of course

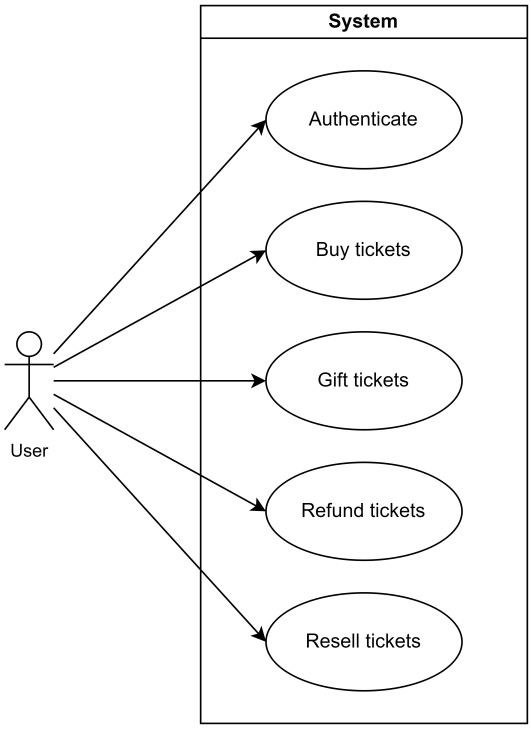

Figure 3.4: User use cases

3.2 Requirements

[brief introduction of the section]

3.2.1 Functional Requirements

The functional requirements are the features that the system must have to fulfill the users needs.

The system must have the following functional requirements:

Table 3.1: Functional Requirements

Requirement	
Connect to a wallet software	The system must be able to connect to a wallet software to intera
Interact with the smart contract	The system has to interact with the smart contract to displ
Add NFTs metadata to the IPFS	The system must be able to add the NFTs metadat

3.2.2 Non-Functional Requirements

The non-functional requirements are the features that the system must have to fulfill the users needs, but that are not directly related to the systems functionality.

The system must have the following non-functional requirements:

Table 3.2: Non-Functional Requirements

Requirement	Description
Scalable	The system must be able to handle a large number of users and events.
Low fees	The system must have low fees for any kind of operation.
Fast	The system must be fast to allow events to have the smoothest experience possible
Secure	The system must be secure to avoid any kind of fraud.
User-friendly	The system must be user-friendly to allow users to easily buy and sell tickets.

The scalable, low fees and fast requirements are essentially associated with the blockchain network choice. As we saw in the Subsection 2.1.4, different networks have different characteristics and we need to choose the one that best fits our needs. The secure and user-friendly requirements are associated with the systems design and implementation. We need to design the system in a way that is secure, to avoid any kind of fraud, and have a smooth experience for the users.

3.3 Architecture

In this section we will be mentioning the architecture of the system. As we can see in the Figure 3.5, the main system will store the organizers that are allowed to create events. For each event created by them, there will be ticket packages deployed, which contains the batch of tickets for each, and the users will buy them and, consequently,

the validators can validate them at the entrance of the venue. The validators are assigned to operate in each event and one validator can be assigned to multiple ones.

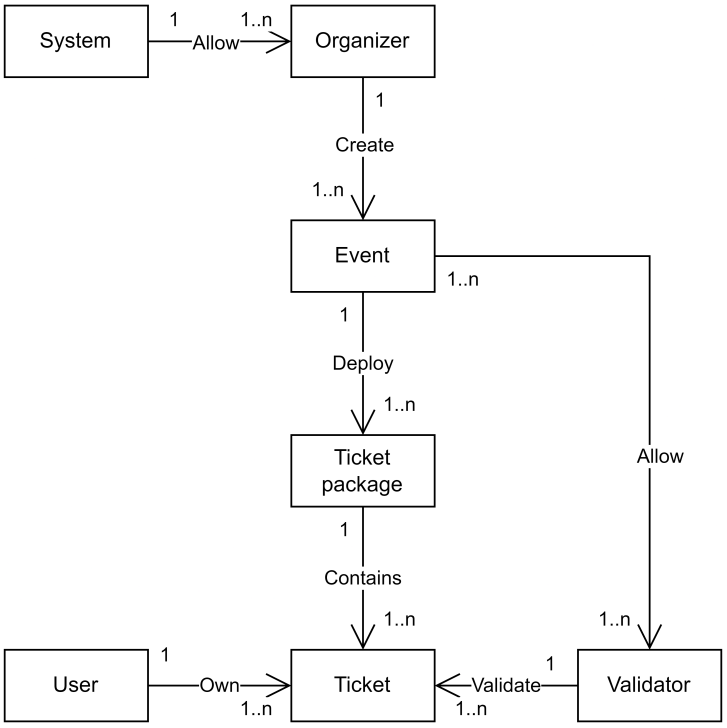

Figure 3.5: Architecture

4

Implementation

[Brief introduction of the chapter]

Being a blockchain project, the goal is to not use the so called Web2 technologies, this means the traditional approaches of having a backend running on a server and similar services, but rather to use only Web3 technologies for us to understand exactly the limitations there are by adopting solely the blockchain ecosystem. Ideally, of course, in the future this project benefits greatly by merging these two approaches.

4.1 Solidity

We'll be doing the smart contracts in Solidity, because it's the most popular language for Ethereum smart contracts, and makes it easy to deploy to any EVM compatible blockchain. This language is similar to other languages like C++, but it has some unique features that make it very powerful for smart contracts.

One important aspect is that when a contract is public, anyone can call its functions, that's why we need to restrict their access and take into account that possibility when defining the operations of the entire contract. So it's always important to have a mind-set that any function (endpoint) can be called by anyone, which is a pretty different approach from traditional web development.

Another thing is that contracts run atomically, so if something should fail or revert, the changes don't take place. Modifiers are a good example where it can be used to

prevent reentrancy attacks, which are a common security issue in smart contracts, and restrict the access to certain functions, reverting if an intruder tries to call them.

We can define, for example, a call to register an event to be restricted to only organizers or a call to validate tickets to be limited to only validators. This is possible because there are a few keywords like *msg.sender* that tell us which user address called the function and *msg.value* that tell us how much value was sent with the transaction. This value is essentially the amount of money the user sends in a transaction, necessary, for example, to buy the tickets. We'll check if the value matches the correct price and reverts otherwise.

Each blockchain network has its own currency, in which is commonly referred to as ether, on the EVM networks. This is the unit of value that is used to pay to execute transactions, but in solidity we cannot work with floating point numbers, so we have to work with the smallest unit of ether, which is called wei. Similar to how we have 1 euro is 100 cents, we have 1 ether is 10^{18} wei, same as on the bitcoin network we have the 1 bitcoin (BTC) is 10^{8} satoshis (SATS). This is the unit being used when we call the *msg.value* variable, so we always have to account for the conversion.

Another unique feature of Solidity is the *address* type, which is basically a string strict to the Ethereum address format. That's how we'll be storing the addresses of the organizers and the events on the contracts. The only difference between a user and a contract is that a contract has code associated with it, so it can execute functions and store data, while a user can only send transactions.

This way we can understand why, to send a transaction, paying for the tickets for example, we send value along with it, matching the expected price according the logic of the buy method. Essentially we're sending money to the contract, which is stored in the contract's balance, like a normal user wallet. This is how the contract can store money and manage it, for then the organizer withdraw its profit to his own wallet.

4.2 Main Smart Contract

So smart contracts are similar to C++ mainly because it lies on a class-like structure with variables to store data and methods, where the main difference is that a class is called a contract and you can extend others to integrate their functionalities. That's essentially what's gonna happen with each event, extending the ERC721 standard, making it a collection of NFTs, where each NFT is a ticket.

Since this is the behavior we want (each event being a NFT collection), we will have to

deploy (instantiate, in C++) a new contract for each event, like the Figure 4.1 shows, so we need a main contract to keep track of these events.

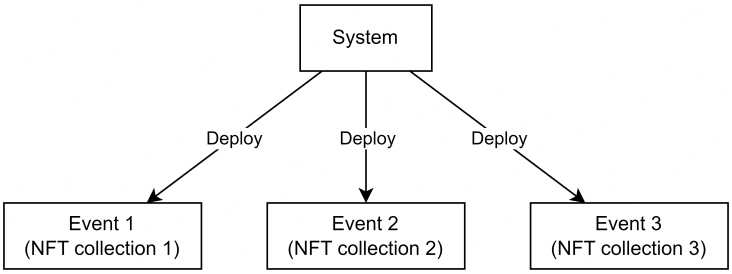

Figure 4.1: System behavior

With this in mind, like we see in the Figure 4.2, the main contract will track the organizers and the events associated with the system, along with the method to register a new event with the necessary data, restricted to only organizers (to avoid unauthorized people to interact).

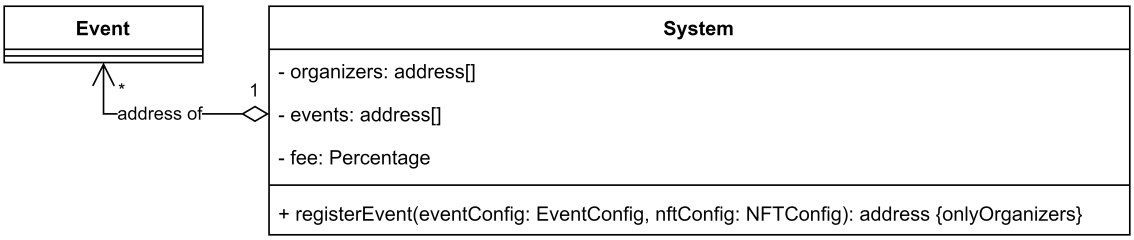

Figure 4.2: Main smart contract UML

With this structure, since we have this main contract where all the events of the system are stored, we can simply make a call to get them all, showcasing them in the app's home page for users to search. Any event that is deployed outside the system or if it gets removed from there, it won't be shown to the users.

4.3 Event Smart Contract

For the event contract, we'll be extending the ERC721 standard and adding the necessary methods to interact with the tickets, like buying, selling, and validating them. The reason to extend this standard and not implement the logic manually is because it

makes it compatible with the most common marketplaces for NFTs, which allows for users to do what they desire with them after the event. It also has the necessary methods to manage the tickets, like transferring them between users, and the necessary operations to track these operations.

4.3.1 ERC721 Structure

The standard was obtained through the OpenZeppelin library, which is a collection of secure and community-vetted smart contracts that are used by many projects in the Ethereum ecosystem. This library is a great resource for developers to build secure and reliable smart contracts.

Analyzing its source code [ref], and looking into the most important variables and methods of the standard shown in the Figure 4.3, we can understand that the NFTs are simply a mapping of the token ID to the owner address, so when you execute a transaction to get a token (this process is called minting), the token ID is then associated to your address. Then for each token it's possible set a URI, which is a link to the NFTs metadata, usually being a JSON file with the necessary information about the token, like the name, description, and image.

This link could point to anything, for example a google drive file, but the common thing is to store the metadata on the IPFS, which is a decentralized storage system, so the metadata is not stored on the blockchain itself (onchain), which would be very expensive, but rather on a decentralized storage system (offchain), which is much cheaper.

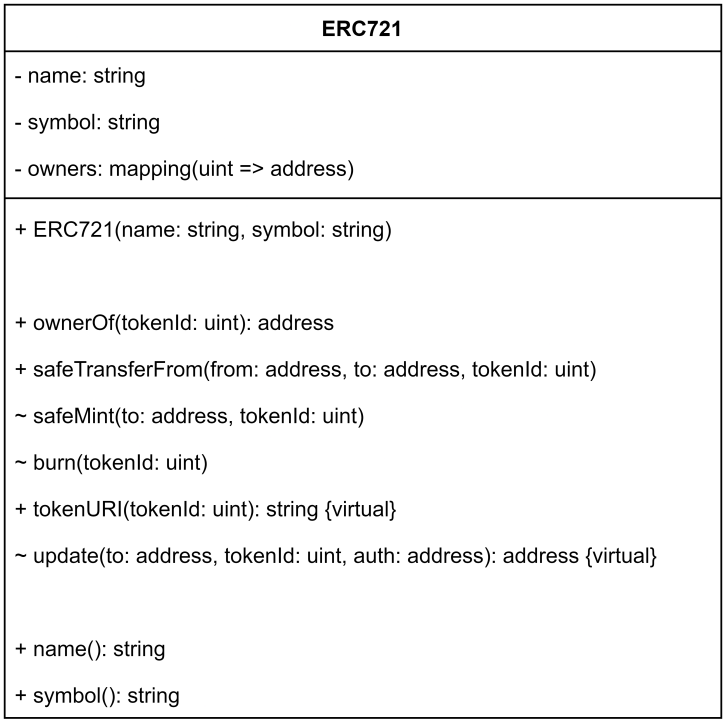

Figure 4.3: ERC721 UML

The function *tokenURI* is the one that is called by default in the marketplaces to get the NFT's metadata, being one of the main reasons to extend the ERC721 standard, because it enforces the implementation of this method. In the Figure 4.3 we see that it has the *virtual* keyword, meaning this can be overridden by the contracts that extend it, to manipulate the way to store the metadata. We'll be mentioning this again in the Section 4.7, about how the packages logic is implemented.

4.3.2 Event Behavior

So the event will be deployed and we need a certain control over the tickets. One of the aspects we need to account for is that when deploying an event, and since the event will extend the ERC721, any public functions on that standard will be possible to execute. This is a problem because we don't want the users to mint tickets whenever they want or transfer them between themselves from outside the system, so we need to restrict these operations. As we saw already on the Figure 4.3, only the *safeTransferFrom* method is public, so users could transfer NFTs between each other. We want that to be possible, just not from outside the system, since that can lead users to exploit the system and scalping the tickets easily. The minting, however, won't be an issue because

it's an internal method, so we will access it from the buy method in the event and restrict it there.

The Figure 4.4 shows the lifecycle of the event, and what restrictions are in place for the ticket operations. We will have 4 main states for the event after it has been registered, being *Open*, *No refund*, *Start*, and *End* dates. Once the event is registered, it will show up in the app for user to see, and the organizers can set a later open date to allow ticket minting (buying).

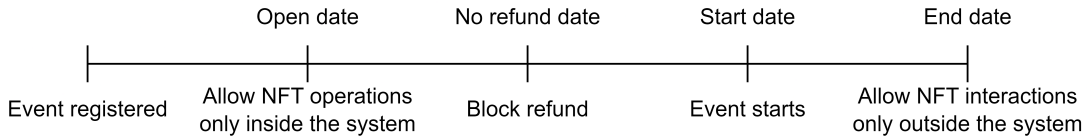

Figure 4.4: Event lifecycle

4.3.2.1 Open Date

Once it hits the *Open* date, we will allow the users to buy the tickets, which will mint the NFTs by executing the *safeMint* method of the ERC721 contract.

4.3.2.2 No Refund Date

After the *No refund* date, we will prevent the users to call the refund method, which essentially *burns* the NFTs, removing them from the user and making them available again. This is a nice operation to add because it allows the users to get their money back if they can't attend the event. The organizer decides the percentage of the refund and the deadline, which is there to prevent users to buy a big amount of tickets and then refund them last minute, which would be a way to exploit the system (in case of a 100% refund, they wouldn't risk anything). The other good thing for the organizer is when the event is expected to be sold out. Since the users will get some money back, they will have a reason to refund their tickets if they cannot attend the event anymore, making them available again for other users to buy at the original price, making the organizer a higher profit. After this deadline, the only that'll be allowed is for users to resell their tickets in the system's marketplace, which them to sell at a higher price than the refund (but never higher than the original, of course).

4.3.2.3 Start Date

The *Start* date is there to tell the users when the event starts, so basically when the gates will open. That's the date that appears in the app, so the users know when to show up.

4.3.2.4 End Date

The *End* date tells when the event is over, unlocking all the ticket operations to outside the system. So users can simply keep the tickets as a souvenir or sell them in any marketplace, without any restrictions on the tickets, including the removal of the price cap.

With this behavior in mind, we came up with the Event UML, like shown in the Figure 4.5, where we added the necessary methods for the organizer/admins to manage the event and the users to handle the tickets, which then trigger the corresponding methods of the ERC721 contract. We added a possibility to have admins so the organizer can distribute the workload of executing the necessary operations to people he trusts.

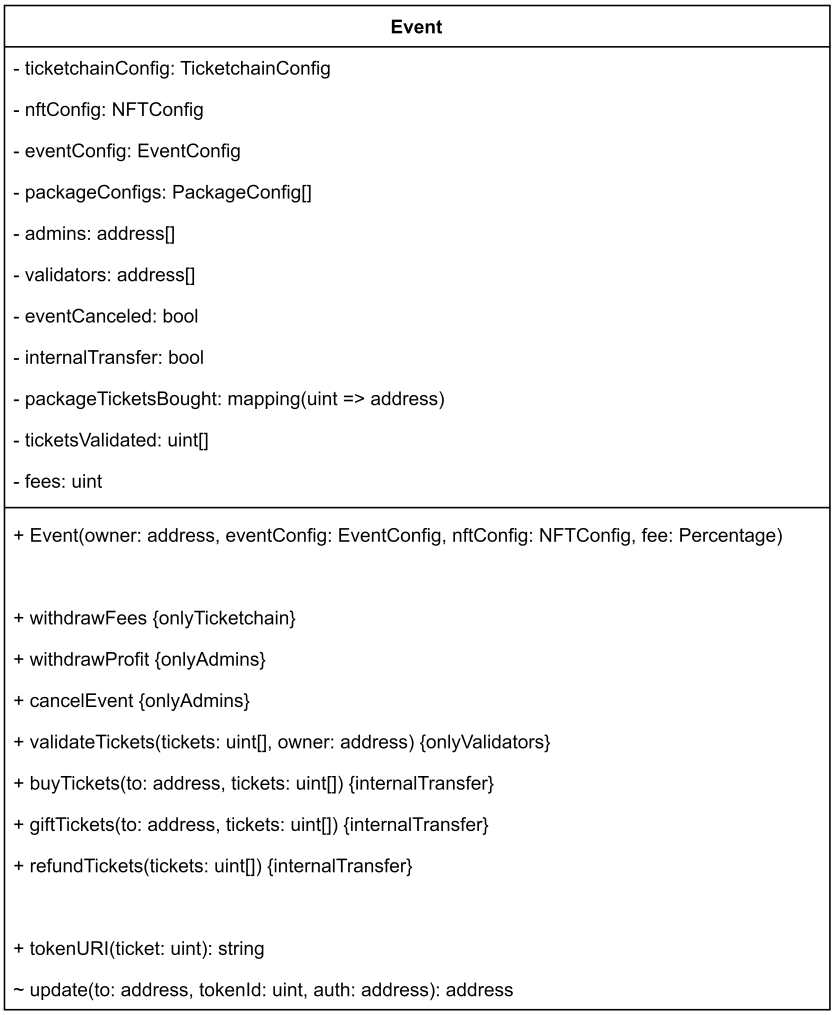

Figure 4.5: Event smart contract UML

As we can see, the *update* method has been overridden from the ERC721 standard, to restrict the interactions with the NFTs according the defined behavior. This method is the one that gets called anytime there's a change in the NFTs state, so we can implement the necessary logic to restrict the operations here and we can visualize it in the following flowchart, shown in the Figure 4.6.

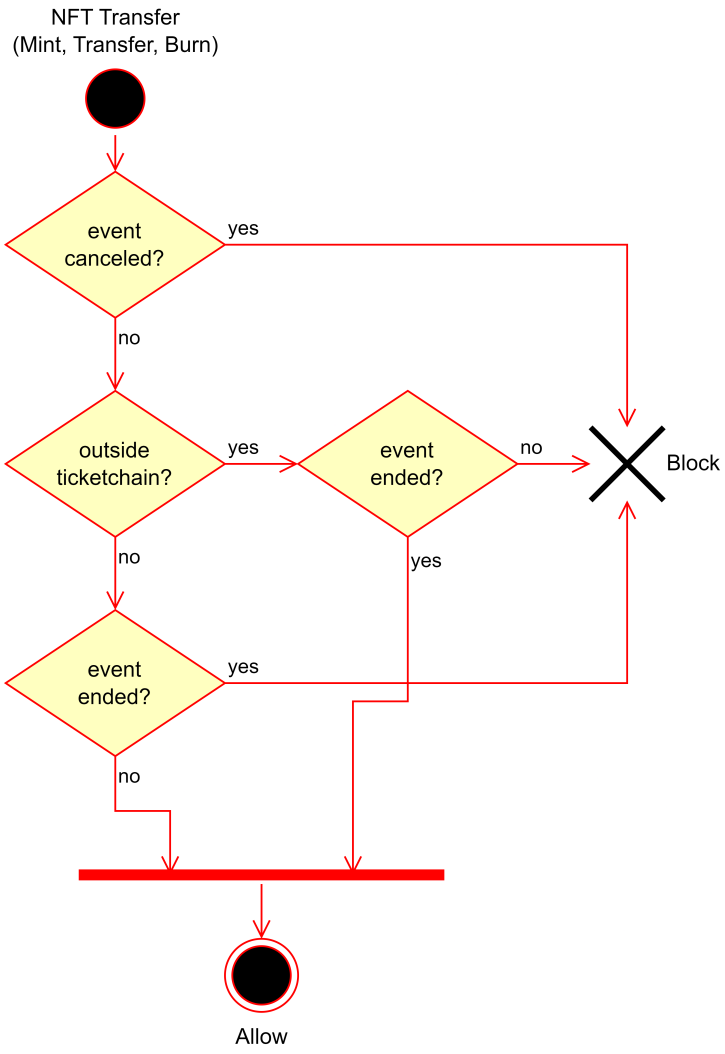

Figure 4.6: NFT flowchart

4.3.3 Structs

As is possible to see in the Figure 4.5, and also in the Figure 4.2, we have a few custom structs to organize better the data. These structs are the *Percentage*, *EventConfig*, *NFTConfig*, *PackageConfig* and *TicketchainConfig* structs.

4.3.3.1 Percentage Struct

The *Percentage* struct is necessary because in Solidity there are no floating point numbers, so we need a way to make calculations with percentages. What this struct does is it stores the value of the percentage and the amount of decimals it has, so if we want to

calculate 55.50% of a number, we would have 555 as the value with 1 decimal, or 5550 with 2 decimals.

The struct is as follows:

```
struct Percentage {
    uint256 value;
    uint256 decimals;
}
```

so to obtain a percentage of some x number, we do $y = \frac{x \times \text{Percentage.value}}{100e^{\text{Percentage.decimals}}}$.

When working with ether units, it can be common to have values like 0.00005 ether, but it's rather rare to have small values in wei, like 1000 wei, so applying this formula won't lose much precision (note that 1 ether is 10^{18} wei).

4.3.3.2 TicketchainConfig Struct

The *TicketchainConfig* struct is simply to keep it stored the system address and the system fee percentage, so we can easily access this information when applying the fees and withdrawing them, and is as follows:

```
struct TicketchainConfig {
    address ticketchainAddress;
    Percentage feePercentage;
}
```

4.3.3.3 NFTConfig Struct

The *NFTConfig* struct is just to store the NFTs basic information, like the name, symbol, and base URI, to ease the input of the NFTs information when registering the event:

```
struct NFTConfig {
    string name;
    string symbol;
    string baseURI;
}
```

The is the name of the NFT collection and the symbol is the abbreviation of it, like the name being 'Ticketchain' and the symbol being 'TCK', for example. The base URI is the link to the metadata of the NFTs, which will be used to get the information about the tickets.

4.3.3.4 EventConfig Struct

The *EventConfig* struct is to store the event's entire configuration, like the name, description, location, dates, and refund, like this:

```
struct EventConfig {
    string name;
    string description;
    string location;
    uint256 openDate;
    uint256 noRefundDate;
    uint256 startDate;
    uint256 endDate;
    Percentage refundPercentage;
}
```

4.3.3.5 PackageConfig Struct

Lastly, the *PackageConfig* struct is there to store each package information, to keep track of the ones that are available for the event:

```
struct PackageConfig {
    string name;
    string description;
    uint256 price;
    uint256 supply;
    bool individualNfts;
}
```

This structure will be better discussed in the next Section 4.3.4.

4.3.4 Ticket Packages

It's common to see events with different types of tickets, like VIP, standard, or even 3-day passes, each with its own price and benefits. We want to implement this feature in the system, so we can have a better control over the tickets and the users can choose the one that fits them better.

For that, we will allow the organizer to add packages, indicating the supply of each one, and as we saw already, the NFTs are a mapping of the ID to the owner, so we can organize the packages as a list, where the supply and order of the package defines the ID of each NFT, like the Figure 4.7 demonstrates.

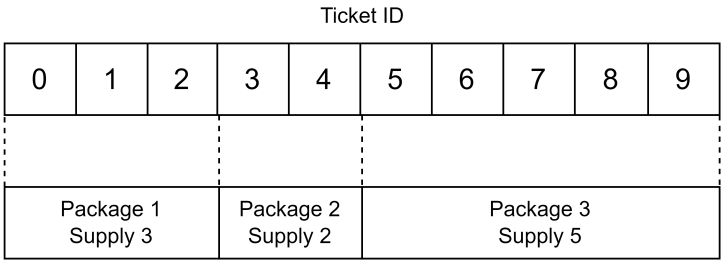

Figure 4.7: Package logic

This way, whenever we need to get a ticket for a certain package, we can go through the packages and see in which one the ID is. One only limitation with this is if the event is already open (users can buy tickets), the only thing we can allow the organizer to do is to add packages, neither remove or change their order, because that would change the ID of the ticket, which would be a problem for the users that already bought them.

Now we just have to make sure the information obtained with the *tokenURI* method corresponds to the ticket, according to its package. For this we will have a different metadata file for each package, with the necessary information about the tickets. The *individualNfts* boolean in the *PackageConfig* struct is there to indicate if the organizer wants each ticket on the package to have its own metadata, or if they can share it.

According to this, the *tokenURI* will return an URI like 'baseURI/packageId/ticketId' for a package with individual NFTs, and 'baseURI/packageId' for a package with shared metadata. Like this, when we store the metadata on the IPFS, we store the metadata for each ticket inside a folder of the packages with individual NFTs, and only a metadata file for each package without individual NFTs.

4.3.5 Metadata Storage

As we mentioned before, we'll be using the InterPlanetary File System (IPFS) for storing the NFTs metadata. The IPFS is a decentralized storage system where the data is stored in a distributed network of nodes (decentralized), making it very secure and reliable. This is a great solution for storing the metadata of the NFTs because it's very

cheap and easy to use, and it's a common practice in the blockchain ecosystem. Other options would be to store the data on some kind of server, but that would be more expensive to maintain, and since we are dealing with NFTs, it's good practice to store the data in a decentralized manner, to avoid any kind of alteration on its contents, if the tickets possibly become valuable collectibles.

This kind of issue was something that has happened before, where people bought NFTs with the idea of them being somewhat valuable, but then the owner changed the contents of the metadata, executing what was called of a *rug pull*, which is a scam that made the NFTs worthless, keeping the money for himself.

To store the data on the IPFS, we will be using the Pinata service, which does the heavy lifting for interacting with the storage itself. To accomplish this, we just need to arrange the files according to the ticket packages, like mentioned before in the Section 4.3.4, and then upload them to the IPFS, getting the link to the metadata, which we'll then store on the contract as the base URI. The files would be stored like the Figure 4.8 shows, being the packages 1 and 3 with shared metadata, and the package 2 with individual NFTs.

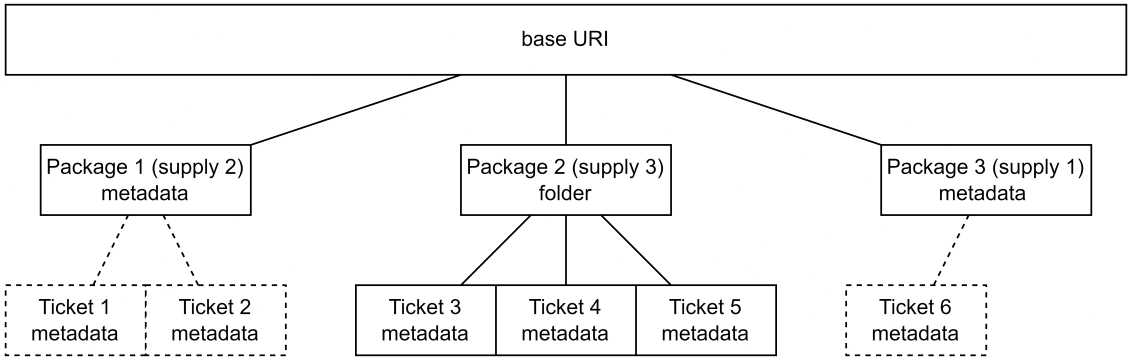

Figure 4.8: Metadata storage

4.3.6 System Fees

One of the most important aspects of a system like this is the business model we have to take into account. Since this is a service we want to deploy for event organizers, we need to make this sustainable and profitable. This kind of service aims to do some heavy lifting, with its own features, so we could set a fee lower than the usual on the traditional marketplaces and ticket selling platforms, since the organizers need to pay for each service.

These low fees are possible because with the system being deployed on the blockchain, it stays there while the network is running, so the only extra cost are the network fees when interacting with the event. For the users, each interaction is paid by them, so when a user buys a ticket, the only thing to take into account are the network fees, which depending on the network can be super low.

We'll set a fee on the main smart contract, where will be stored in the event when registering it, so that if we decide to change it, the previous events aren't affected. This is also because we want to abstract the user of any extra fee, so the price the organizer sets, is the price the user pays, and the system fee is taken from the tickets price. In case an event gets cancelled, or a user decides to get a refund, the ticket fee is returned to the user (proportional to the refund), making the system less profit, but guarantees the users of a fair process. With this, we need to restrict the system to only withdraw any profit when the event is over. Since this is rather an uncommon case, the less profit for the system is worth the trust the users and organizers will have in it.

4.3.7 Ticket Validation

For the ticket validation, we must take into consideration a lot of aspects, because it's not just checking if the user address has a ticket associated to him. This is because, since the data is on the blockchain, anyone can see the addresses where each ticket belongs to, and pretend he's the owner of the ticket. For this to be secure, we need to guarantee the user is actually the owner of the address, and here is where the cryptographic message signature comes in, the same process that happens when executing a normal blockchain transaction.

Essentially the user will have to sign a message for the validator to check if the signature is valid, and if it is, it means the user is the actual owner of the ticket. There's a cryptographic method to recover the address of the signer using the original message and it's signature.

Knowing this, both parties need to know the original message so it matches. We could just use a default message for everyone, so the users would just need to give the signature to the validator, but this could become a security issue, in case the signature gets leaked, anyone who has it, could pretend to be a different address. The idea here is to have a unique message for each user at the time of the event, so it forces the user to sign it in the moment.

The process defined is shown in the Figure 4.9, where the user reads the QR with a generated message from the validator, signs it with his wallet, and generates a JSON

with the signature and useful information like the tickets to validate, the event, and the user address. After the validator reads the QR code with the JSON, he checks if the parameters match the ones on the blockchain, gets the address from the signature and the message, and checks if the user is the owner of the tickets. If everything matches, the validator will trigger a transaction to mark the tickets as validated, to avoid people sharing the accounts and using the same ticket multiple times.

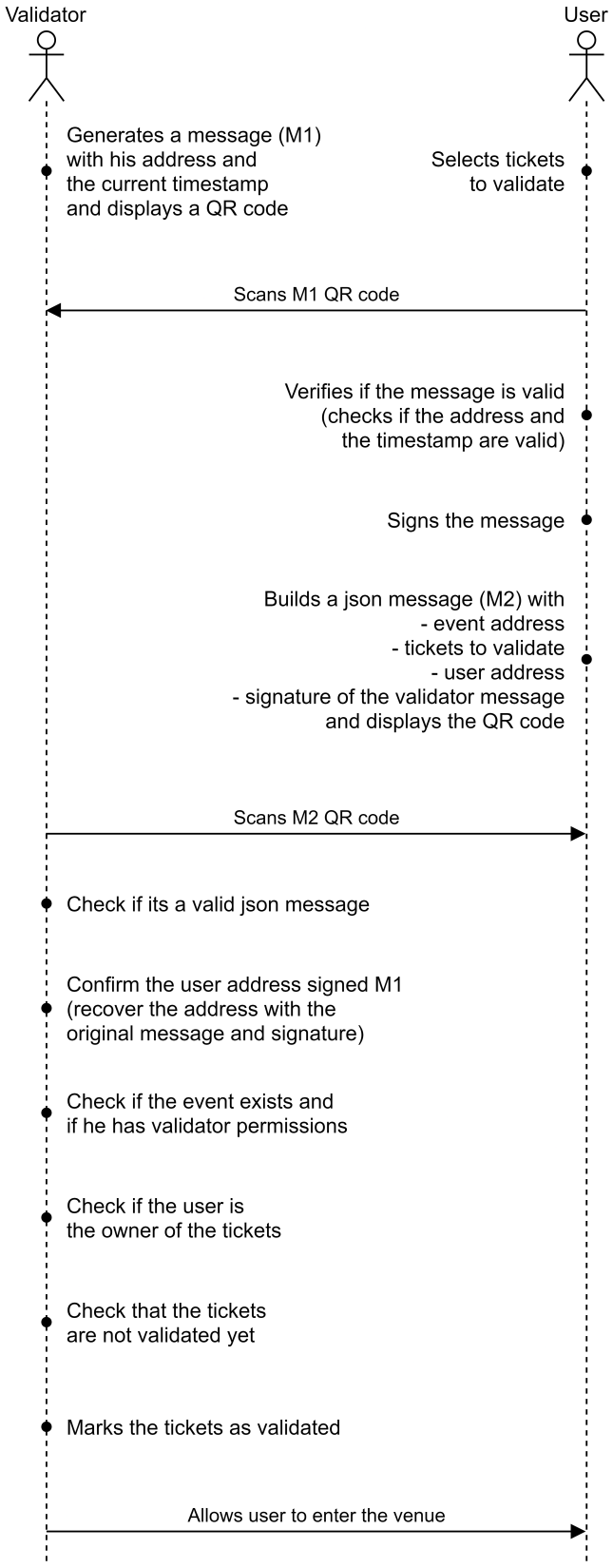

Figure 4.9: Ticket validation

All this is reduced to a single transaction on the blockchain because, depending on the network, the finality of a transaction can take a while (the time it takes for a transaction to be fully registered on the blockchain). This will be discussed in the SECTION. This was planned to be done in a single transaction to avoid congestion at the entrance of the venue, so the users can enter the event with the least amount of delay.

[NOTES]

[should i mention marketplace and dashboard?] [should i include the setters and getters in the UMLs?]

[todo mention network fees and network choice]

[user app] [validator app]

4.4 Project Features

5

Results

6

Conclusions

6.1 Limitations

[network fees] [finality]

6.2 Future Work

[marketplace (resell)] [website dashboard]