COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

MÁSTER UNIVERSITARIO EN INGENIERÍA DE TELECOMUNICACIONES

TRABAJO FIN DE MÁSTER

DESIGN AND IMPLEMENTATION OF A SOFTWARE SOLUTION USING BLOCKCHAIN TECHNOLOGY FOR DECENTRALIZING A BOOKING PLATFORM BY IMPLEMENTING SMART CONTRACTS COMPLIANCE WITH ETHEREUM STANDARDS. DEVELOPMENT OF A DAPP FOR TESTING A REAL CASE STUDY.

Autor: Álvaro Gericke Parga

Director: Jesús Ruiz Martinez

Co-Director: Carlos Pastor Matut

Co-Director: Edward Chlebus

Madrid

Agosto de 2019

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título

……..……..……

……

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el curso académico ……. es de mi autoría, original e inédito y no ha sido presentado con anterioridad a otros efectos. El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido tomada de otros documentos está debidamente referenciada.

Fdo.: (Nombre del alumno) Fecha: ……/ ……/ ……

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO

Fdo.: (Nombre del Director) Fecha: ……/ ……/ ……

AUTHORIZATION FOR DIGITALIZATION, STORAGE AND DISSEMINATION IN THE NETWORK OF END-OF-DEGREE PROJECTS, MASTER PROJECTS, DISSERTATIONS OR BACHILLERATO REPORTS

1. Declaration of authorship and accreditation thereof.

The author Mr. /Ms. ÁLVARO GERICKE PARGA

HEREBY DECLARES that he/she owns the intellectual property rights regarding the piece of work: Design and Implementation of a software solution using Blockchain Technology for decentralizing a booking platform by implementing smart contracts compliance with ethereum standards. Development of a DApp for testing a real case study.
that this is an original piece of work, and that he/she holds the status of author, in the sense granted by the Intellectual Property Law.

2. Subject matter and purpose of this assignment.

With the aim of disseminating the aforementioned piece of work as widely as possible using the University's Institutional Repository the author hereby **GRANTS** Comillas Pontifical University, on a royalty-free and non-exclusive basis, for the maximum legal term and with universal scope, the digitization, archiving, reproduction, distribution and public communication rights, including the right to make it electronically available, as described in the Intellectual Property Law. Transformation rights are assigned solely for the purposes described in a) of the following section.

3. Transfer and access terms

Without prejudice to the ownership of the work, which remains with its author, the transfer of rights covered by this license enables:

a) Transform it in order to adapt it to any technology suitable for sharing it online, as well as including metadata to register the piece of work and include "watermarks" or any other security or protection system.

b) Reproduce it in any digital medium in order to be included on an electronic database, including the right to reproduce and store the work on servers for the purposes of guaranteeing its security, maintaining it and preserving its format.

c) Communicate it, by default, by means of an institutional open archive, which has open and cost-free online access.

d) Any other way of access (restricted, embargoed, closed) shall be explicitly requested and requires that good cause be demonstrated.

e) Assign these pieces of work a Creative Commons license by default.

f) Assign these pieces of work a HANDLE (*persistent* URL). by default.

4. Copyright.

The author, as the owner of a piece of work, has the right to:

a) Have his/her name clearly identified by the University as the author

b) Communicate and publish the work in the version assigned and in other subsequent versions using any medium.

c) Request that the work be withdrawn from the repository for just cause.

d) Receive reliable communication of any claims third parties may make in relation to the work and, in particular, any claims relating to its intellectual property rights.

5. Duties of the author.

The author agrees to:

a) Guarantee that the commitment undertaken by means of this official document does not infringe any third party rights, regardless of whether they relate to industrial or intellectual property or any other type.

b) Guarantee that the content of the work does not infringe any third party honor, privacy or image rights.
c) Take responsibility for all claims and liability, including compensation for any damages, which may be brought against the University by third parties who believe that their rights and interests have been infringed by the assignment.
d) Take responsibility in the event that the institutions are found guilty of a rights infringement regarding the work subject to assignment.

6. Institutional Repository purposes and functioning.

The work shall be made available to the users so that they may use it in a fair and respectful way with regards to the copyright, according to the allowances given in the relevant legislation, and for study or research purposes, or any other legal use. With this aim in mind, the University undertakes the following duties and reserves the following powers:

a) The University shall inform the archive users of the permitted uses; however, it shall not guarantee or take any responsibility for any other subsequent ways the work may be used by users, which are non-compliant with the legislation in force. Any subsequent use, beyond private copying, shall require the source to be cited and authorship to be recognized, as well as the guarantee not to use it to gain commercial profit or carry out any derivative works.
b) The University shall not review the content of the works, which shall at all times fall under the exclusive responsibility of the author and it shall not be obligated to take part in lawsuits on behalf of the author in the event of any infringement of intellectual property rights deriving from storing and archiving the works. The author hereby waives any claim against the University due to any way the users may use the works that is not in keeping with the legislation in force.
c) The University shall adopt the necessary measures to safeguard the work in the future.
d) The University reserves the right to withdraw the work, after notifying the author, in sufficiently justified cases, or in the event of third party claims.

Madrid, on 25 of August of 2019

HEREBY ACCEPTS

Signed.....

Reasons for requesting the restricted, closed or embargoed access to the work in the Institution's Repository

COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

MÁSTER UNIVERSITARIO EN INGENIERÍA DE TELECOMUNICACIONES

TRABAJO FIN DE MÁSTER

DESIGN AND IMPLEMENTATION OF A SOFTWARE SOLUTION USING BLOCKCHAIN TECHNOLOGY FOR DECENTRALIZING A BOOKING PLATFORM BY IMPLEMENTING SMART CONTRACTS COMPLIANCE WITH ETHEREUM STANDARDS. DEVELOPMENT OF A DAPP FOR TESTING A REAL CASE STUDY.

Autor: Álvaro Gericke Parga

Director: Jesús Ruiz Martinez

Co-Director: Carlos Pastor Matut

Co-Director: Edward Chlebus

Madrid

Agosto de 2019

Agradecimientos

En primer lugar quiero agradecer a mis padres y mis hermanos por todo el apoyo que me han dado durante estos años y por apoyarme en todo esos proyectos que he llevado a cabo en combinación con los años de Universidad.

También quisiera agradecer a mis tutores Jesús Ruiz y Carlos Partor por haberme ayudado y acompañado en el desarrollo de este trabajo fin de máster.

DISEÑO E IMPLEMENTACIÓN DE UNA SOLUCIÓN BASADA EN TECNOLOGÍA BLOCKCHAIN PARA LOGRAR LA DESCENTRALIZACIÓN DE UN SISTEMA DE ALQUILERES APLICANDO SMART CONTRACTS CONFORMES A LOS ESTÁNDARES DE ETHEREUM. DESARROLLO DE DAPP PARA TESTEAR UN CASO DE USO REAL.

Autor: Gericke Parga, Álvaro.
Director: Pastor Matut, Carlos.
Co-Director: Ruiz Martínez, Jesús.
Co-Director: Schlebus, Edward.
Entidad Colaboradora: ICAI – Universidad Pontifica Comillas.

RESUMEN DEL PROYECTO

Mediante este proyecto se pretende desarrollar y desplegar un conjunto de contratos inteligentes que permitan migrar la lógica asociada a un sistema de alquileres a una arquitectura basada en tecnología Blockchain, así como elaborar una aplicación descentralizada que sirva como prueba de concepto. Esto nos permitirá cambiar la industria de los alquileres, actualmente controlada en exceso por plataformas como Booking o Airbnb, las cuales imponen condiciones estrictas que resultan en el detrimento de los servicios.

Palabras clave: Blockchain, Ethereum, Smart Contracts, Booking, Dapp, Standards.

1. Introducción

En los últimos años la industria de los alquileres ha crecido año tras año a un ritmo más acelerado del acostumbrado. Esto se debe en parte al cambio de mentalidad, especialmente entre las generaciones más jóvenes, en cuanto a la idea de poseer productos. Se da prioridad al concepto de compartir y alquilar productos que resultan costosos y se resta importancia a la necesidad de poseerlos.

Este escenario ha provocado que empresas que se dedican a poner en contacto oferta y demanda en esta industria, hayan crecido exponencialmente y hayan adquirido un papel esencial en estas transacciones. Este es el caso de empresas como Airbnb o Booking. Sin embargo, el poder adquirido se ha visto reflejado en comisiones cada vez más altas y condiciones más estrictas que han provocado que las empresas finales que son propietarias de los productos de alquiler, hayan tenido que recortar gastos hasta el punto de que se ha visto perjudicado el cliente.

Sin embargo, la irrupción de la tecnología Blockchain ha abierto las puertas hacia un futuro muy diferente. Las características que proporciona esta tecnología, tales como la descentralización, confianza y seguridad en un entorno que carece de un ente central o automatización de procesos, plantea un escenario nuevo en el que estos marketplaces pierdan en cierta medida ese poder dando paso a un mercado más eficiente.

2. Definición del Proyecto

Esta tesis de máster tiene como objetivo principal diseñar y desplegar un sistema de alquileres que haga uso de la red Blockchain. Así mismo, pretende desarrollar una

aplicación basada en un marketplace existente, que haga uso de dicho despliegue y sirva como caso de uso y prueba de concepto de la idea.

Se han desarrollado un conjunto de contratos inteligentes que, desplegados en la red de Ethereum, una de las redes blockchain públicas más utilizadas, actúe como elemento que implemente la lógica de una aplicación de alquileres. El diseño de estos contratos inteligentes estará enfocado en utilizar estándares de Ethereum ya existentes que permitan dotar de mayor interoperabilidad al sistema, así como revisar la eficiencia de las transacciones generadas por dicha solución.

Finalmente, se ha desarrollado una aplicación descentralizada, que no es más que una interfaz de usuario a través de la que interactuar con los contratos inteligentes, que permite testear el alcance y validez de dicho sistema. En caso de obtener resultados positivos, podría suponer una innovación en la industria de los alquileres, ya que implicaría que la gran cantidad de intermediarios que existen hoy en día en dicho sector, podrían verse directamente afectados.

3. Descripción del sistema

El Sistema desarrollado consta de dos partes bien diferenciadas, pero perfectamente integrables entre sí. Por un lado, tenemos los contratos inteligentes que se encargarán de implementar toda la lógica que es necesaria para llevar a cabo un sistema de alquileres. Y por otro lado tenemos la aplicación descentralizada, que se apoya en dichos contratos inteligentes desplegados en la blockchain, para ofrecer a sus clientes una interfaz sencilla e intuitiva que abstraiga los procesos de interacción con dichos contratos y la tecnología blockchain, y permita ofrecer a los usuarios servicios de alquiler de diferentes tipos de productos.

La arquitectura de dichos contratos puede verse en la Ilustración 1. Tal y como podemos ver, estará compuesta de dos tipos de módulos. El primero y más importante, identificado como *ProducsManagement Smart Contract*, es el que incluirá prácticamente toda la lógica. Dicho módulo esta compuesto por un único contrato inteligente. Éste actuará como puerta de entrada de cualquier transacción o aplicación que quiera interactuar con el sistema. A través de él se registrarán los productos y se realizarán las peticiones de alquiler y cancelaciones. Este contrato, por cada producto registrado, desplegará un contrato de tipo *BookingManagement,* en el cual se llevará un registro de todos los alquileres relativos a dicho producto.

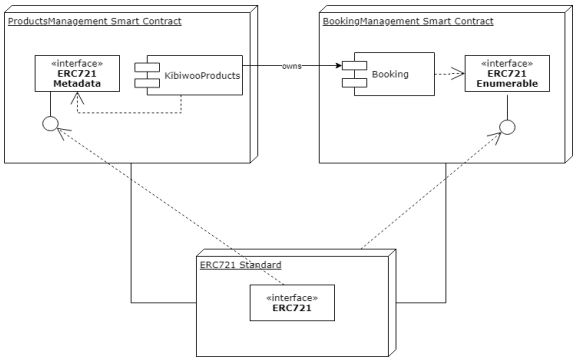


Ilustración 1 – Arquitectura de los contratos inteligentes.

De esta manera, se introduce el segundo módulo referenciado como *BookingManagement Smart Contract.* Este módulo estará compuesto por tantos contratos como productos registrados haya en la plataforma. Estos contratos serán mucho más simples y únicamente se encargarán de registrar las reservas y cancelaciones de cada producto y guardar dicho estado en el propio contrato.

Por lo tanto, simplemente teniendo un mapeo de identificadores de productos a direcciones de contratos, el primer módulo será capaz de comunicarse con el contrato adecuado con cada petición de alquiler de un determinado producto que reciba, traspasar dicha petición al contrato correspondiente, y éste último aceptar o rechazar dicha reserva en función de la disponibilidad.

Por último, destacar que todos los contratos inteligentes implementan las interfaces del estándar conocido como ERC-721 dentro de la blockchain de Etheruem, por la cual permite identificar los productos y los derechos de reserva mediante tokens los cuales pueden ser intercambiados, transferidos, prestados e incluso incluidos en un exchange.

Así mismo, tenemos la aplicación descentralizada que hace uso de múltiples herramientas para desarrolladores existentes en su arquitectura. Podemos ver dicha arquitectura en la Ilustración 2.

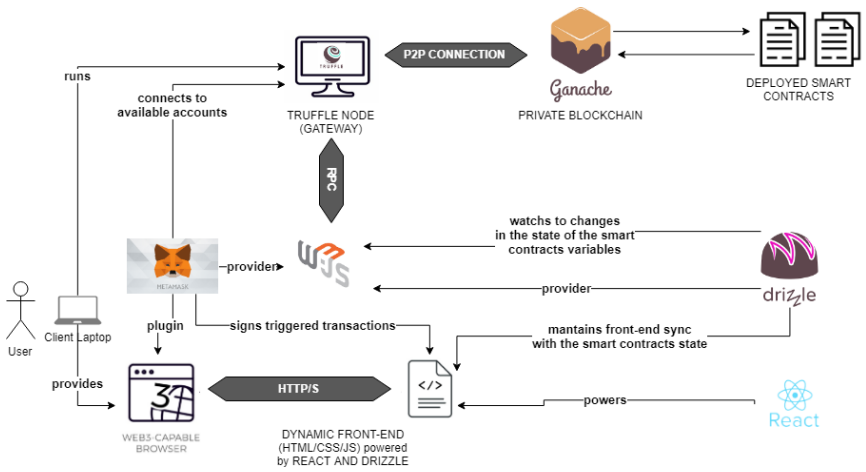


Ilustración 2 – Arquitectura Aplicación Descentralizada

Podemos ver que hay múltiples herramientas software. En primer lugar, tenemos la aplicación desarrollada con React. Esta aplicación es ejecutada en un navegador instalado en el ordenador. A su vez, será necesario tener instalado un nodo que actúe de portal entre la aplicación descentralizada y la Blockchain. Así mismo, dicho nodo estará conectado a la red Blockchain, en la cual estarán desplegados los contratos inteligentes previamente descritos.

Podemos ver también un plugin del navegador que se conecta al nodo. La función de este plugin es permitir el acceso a la cuenta que el nodo registra en la blockchain, para poder gestionarla y mandar transacciones desde el propio navegador de manera sencilla, sin necesidad de recurrir a una consola. Además, observamos una herramienta denominada Drizzle. Esta herramienta permite conectar nuestra interfaz a los contratos

inteligentes, y mantener nuestra interfaz sincronizada con la información contenida en los contratos inteligentes. Esta herramienta será la que refresque y actualice nuestra aplicación cada vez que se produzca algún cambio en los contratos inteligentes, ya sea por la interacción de otros usuarios o por transacciones enviadas por nosotros mismos. Por último, tanto la herramienta Drizzle como Metamask actúan como proveedores de un objeto web3, el cual hace referencia a una librería Javascript para poder enviar acciones al nodo usando un lenguaje de alto nivel, que posteriormente será traducido por el nodo a lenguaje específico de Ethereum para enviarlo a la red y que dicha transacción sea procesada.

4. Resultados

El resultado final ha sido una prueba de concepto exitosa que ha demostrado que la lógica de un sistema de alquileres puede ser gestionada por una serie de contratos inteligentes en la blockchain de Ethereum sin necesidad de utilizar tokens privados.

Se han desplegado dos versiones de aplicaciones descentralizadas, y su vez se han implementado casos de uso que han permitido la gestión de productos de distintas categorías, evitando así tener un despliegue ligado a un único sector como es el caso de la mayoría de las soluciones existentes. Así mismo, ha quedado patente el potencial de implementar los estándares de Ethereum para identificar los productos y las reservas.

También ha dado buen resultado el diseño adoptado de diferenciar entre tokens que representan posesión de los productos y tokens que representan derechos de uso durante un periodo de tiempo de un producto específico, en lugar de tener un único token que identifica al producto y es transferido a otra cuenta a lo largo del período de alquiler.

5. Conclusiones

A raíz del trabajo realizado a lo largo de este proyecto podemos concluir que las soluciones existentes que abusaban de implementaciones con elementos privados desplegados en blockchain privadas con la intención de obtener cierto control sobre el sistema. La solución desarrollada sin utilizar ningún tipo de token creado o blockchain privada ha sido capaz de dar el mismo servicio. Además, la implementación de los estándares ha permitido extender la usabilidad del sistema y crear mayor valor añadido para los usuarios. Pues los tokens utilizados en nuestros contratos inteligentes tienen mucho potencial que se extiende más allá de los objetivos iniciales fijados.

Se ha proporcionado una variante a las soluciones existentes que abre nuevos retos para ser investigados. Es necesario extender el alcance de este proyecto y desplegar en la red de Alastria dicha aplicación y contratos inteligentes para probar dicha solución en un entorno enfocado que está más enfocado al negocio.

Por último, destacar que tal y como hemos visto a lo largo del proceso, es necesario no centrarse únicamente en exprimir al máximo la tecnología Blockchain, sino hacer un balance entre requerimientos técnicos y de negocio. De esta manera, alcanzamos una solución intermedia y lo más eficiente posible en la que tenemos una parte de los

procesos gestionados en la red y otras off-chain. Es muy importante determinar la separación entre lo procesos que son gestionados bajo un entorno descentralizado y los que se mantienen privados.

6. Referencias

[1] Andreas M, Antonopoulos and Wood, Gavin. *Mastering Ethereum: Building Smart Contracts and Dapps*. O'Reilly Media, 2018.

[2] Entriken, W.; Shirley, D.; Evans, J and Sachs, N. EIP 721: ERC-721 Non-Fungible Token Standard, January 2018. https://eips.ethereum.org/EIPS/eip-721.

[3] Zibin, Z.; Shaoan, X.; Hongning, D.; Xiangping, C. and Huaimin, W. *An overview of blockchain technology: Architecture, consensus, and future trends*. In 2017 IEEE International Congress on Big Data (BigData Congress), pages 557-564. IEEE, 2017.

DESIGN AND IMPLEMENTATION OF A SOFTWARE SOLUTION USING BLOCKCHAIN TECHNOLOGY FOR DECENTRALIZING A BOOKING PLATFORM BY IMPLEMENTING SMART CONTRACTS COMPLIANCE WITH ETHEREUM STANDARDS. DEVELOPMENT OF A DAPP FOR TESTING A REAL CASE STUDY.

Autor: Gericke Parga, Álvaro.
Supervisor: Pastor Matut, Carlos.
Co-Supervisor: Ruiz Martínez, Jesús.
Co-Supervisor: Schlebus, Edward.
Entidad Colaboradora: ICAI – Universidad Pontifica Comillas.

ABSTRACT

This Project aims to develop and deploy a series of smart contracts that allow to migrate a booking system logic to a decentralized environment based in Blockchain technology. It also develops a decentralized application that acts as a proof of concept of the solution. This will allow to disrupt the booking industry, which is highly controlled by big marketplaces such as Booking or Airbnb. These actors have leverage that power in order to impose tough conditions which have adversely affect the quality of services provided within this industry.

Key Words: Blockchain, Ethereum, Smart Contracts, Booking, Dapp, Standards.

1. Introduction

The booking industry has experienced a continuous rapid growth during last years. This is especially due to the change of mind young people have experienced and the possibilities internet provides for all type of renters. The idea of ownership is not so well-established among young generations such as millennials. Instead, they embrace more the concept of sharing, renting or booking items.

This new scenario has led into the exponential growth of several companies that provide services in this industry and that connect offer and demand. Growth has turned into power and they play an essential when talking about transactions. This is the example of enterprises such as Airbnb or Booking. Nevertheless, this power has meant higher commissions and tougher conditions imposed to the owners of the rental products, who now are struggling. They have needed to cut costs, until the point that this fact has adversely affect the client.

However, the irruption of Blockchain technology has open new possible paths to a very different future. Blockchain characteristics such as decentralization, trust in a trust-less environment, security in an environment which lacks a central authority and process automatizations suggests new scenarios in which these marketplaces cannot leverage those network effects which as a consequence will enable a more efficient and fair market.

2. Project Definition

This end of master's project sets as its main objective the design and development of a booking system that works in a blockchain. In addition, it sets as its goal the development of a decentralized application, based on an existing marketplace, which connects and

interacts with the previous system deployed in the Blockchain, and that serves as a proof of concept of our idea.

Several smart contracts have been developed and deployed to Ethereum Blockchain, which is one of the most known as used public blockchains. These smart contracts implement all the processes bounded to a booking system but in a decentralized environment. The implementation of existing standards has been set as a main requirement of the system. This approach will provide our solution with a free interoperable infrastructure. The system should look forward for having a design that considers thoroughly the cost of transactions.

Finally, a decentralized application has been developed, which is no more than a user interface through which we can interact with the deployed smart contracts and allows to test the validity of the system. In case the system gives positive results, it could mean a great innovation in the booking industry as the numerous existent intermediaries would be directly affected.

3. System Description

The system consists of two different parts which can be perfectly connected. In one hand, we have the smart contracts which will be in charge of implementing the necessary logic for being able to give support to a booking system. In the other hand, we have the decentralized application, which provides a simple and friendly user interface for interacting with the deployed smart contracts. This interface will hide the complicated logic and processes of interacting with the smart contracts and the blockchain network while still offering a great user experience and offers them a system that supports the booking of different types of products.

We can see the smart contracts architecture in Figure 1. As it can be observed, the system is composed of two types of modules. The first one, referred as *ProductManagement Smart Contract*, is the most important one. It will include nearly all the logic for enabling the booking service. This module has only one smart contract which will act as a gateway for every transaction o application that wants to interact with the system. It will handle all the products registrations, booking and cancel transactions. This smart contract will deploy a *BookingManagement* smart contract for every registered product. The *Booking Management* smart contract will handle and maintain a register with all the bookings for that specific product.

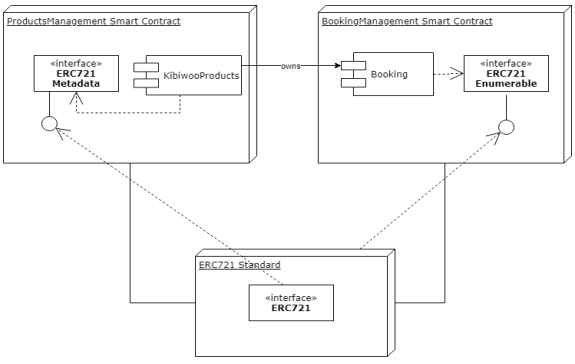


Figure3 – Smart Contracts Architecture.

This way, it has been introduced the second module type, the *BookingManagement Smart Contract.* This module will contain as many smart contracts as products the system has. These smart contracts will be lighter and simpler and will handle only the registration and cancellations of the reservations made of the specific product they represent and maintain the schedule for the product inside the contract.

Therefore, by having a simple mapping which links products ids to contract addresses inside the ProductsManagement smart contract, this main module will be able to communicate and redirect reservation transactions of specific products to the correct BookingContract that references the product. Then, the Bookingcontract will accept or reject the reservation by consulting its state.

Lastly, it should be outlined that as we can see in Figure 1, every developed smart contract implements the interfaces from the Ethereum contact standard ERC721. This standard allows the unambiguous identification of products and reservation rights through ERC721 tokens. In addition, these tokens can be traded or transferred.

Then, we have the decentralized application which is backed by several developer tools in its architecture as it can be seen in Figure 2.

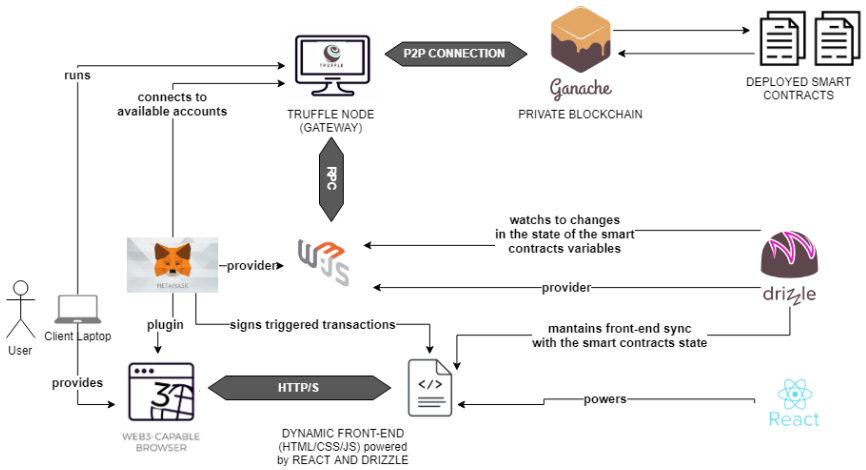


Ilustración 4 – Arquitectura Aplicación Descentralizada

Several software tools are implemented and used. First of all, we use React for building the application, which is run in a browser. It is necessary to install and run a client which acts as a gateway between the decentralized application and the blockchain, as it's the case of the Truffle node in Figure 2. This node will be connected to the blockchain in which the smart contracts have been previously deployed.

A browser extension known as Metamask will be also connected to the client. This extension will act as a wallet, allowing access to the blockchain account stored in the client through the browser. This way we will be able to manage the account though a simple interface in the browser as well as signed and send transactions, avoiding using a console. Drizzle is another powerful developer tool which connects our application front-end to the smart contracts and maintains the interface in sync with the information form the contracts. This tool will be the one that updates the webpages when a modification is

perceive in the state of the smart contracts or a transaction to the smart contracts is sent. Finally, both Drizzle and Metamask expose a web3 object, which is a Javascript library that enables the communication with the node with a high level language, which is translated into Ethereum Virtual Machine language by the client prior to sending it to the network.

4. Results

The final result has been a successful test that has proved that a booking system logic can be managed by a series of smart contracts in the Ethereum Blockchain without needing private tokens.

Two different decentralized applications have been deployed, and several use cases have been implemented for testing the ability of the system to handle the booking of different type of products, avoiding a solution bounded to a specific use case, which is the case of the majority of the existing solutions. Likewise, it has been proved the high potential of having our smart contracts comply with some Ethereum standards for identifying through standard tokens both the ownership and the rental rights of the products.

Also, the selected design of distinguishing between ownership and rental rights tokens has resulted really positive against the approach of implementing a unique token that represents the products and is transferred every time the producto is booked.

5. Conclusions

As a result of the work done throughout this project, it can be concluded that the existing solutions made use of private elements such as private tokens, when it was not strictly necessary with the mere objective of acquiring some control over the system. The provided solution without any element of that type has been able to provide the same or even a better service. What is more, conforming to the Ethereum standards has allowed the system to gain in usability and interoperability, which results in added value for the customer. Actually, the tokens used within the smart contracts have a potential which greatly exceeds the initial goals set.

An alternative to the existing solutions has been provided, which creates new challenges for future research. It is necessary to extent the scope of this project and deploy the system to Alastria's Blockchain. This will enable to test the solution in a more business focused environment.

Lastly, it should be outlined as detected throughout the project, that the focus should not be put only in squeezing out Blockchain Technology, but instead make an equilibrium between technical requirements and business necessities. This way, we can obtain a more efficient solution that fulfils both sides. Also, we should determine until which processes are interesting for being processed on-chain and which ones should be remain off-chain. It is important to establish which processes are worthy for being manage in a decentralized environment and which ones are preferably to maintain private.

6. References

[1] Andreas M, Antonopoulos and Wood, Gavin. *Mastering Ethereum: Building Smart Contracts and Dapps*. O'Reilly Media, 2018.

[2] Entriken, W.; Shirley, D.; Evans, J and Sachs, N. EIP 721: ERC-721 Non-Fungible Token Standard, January 2018. https://eips.ethereum.org/EIPS/eip-721.

[3] Zibin, Z.; Shaoan, X.; Hongning, D.; Xiangping, C. and Huaimin, W. *An overview of blockchain technology: Architecture, consensus, and future trends*. In 2017 IEEE International Congress on Big Data (BigData Congress), pages 557-564. IEEE, 2017.

Contents

Agradecimientos **IX**

Contents **XXI**

List of Figures **XXIII**

List of Tables **XXV**

1 Introduction **1**

1.1 Context and Motivation 1

1.2 Structure of this document 2

2 Enabling Technologies **5**

2.1 Git 5

2.1.1 GitHub 6

2.2 Blockchain 6

2.2.1 Blockchain Technology Description. 7

2.2.1.1 How a Blockchain Works. 13

2.3 Ethereum 17

2.3.1 Ethereum Accounts. 18

2.3.2 Ethereum Transactions. 18

2.3.3 Ethereum Clients 20

2.3.4 Ethereum Virtual Machine 21
2.3.5 EIPs - Ethereum Improvement Proposals 25
2.4 Smart Contracts 28
2.5 Solidity 28
2.5.1 Open-Zeppelin 29
2.6 Truffle Suite 29
2.6.1 Ganache 30
2.6.2 Drizzle 30
2.7 WEB3.js 31
2.8 MetaMask 31
2.9 JavaScript Libraries 32
2.9.1 TDD/BDD Coding Practices 32
2.9.2 MOCHA 33
2.9.3 CHAI 33
2.10 REACT 33

3 State of the Art **35**

3.1 Introduction 35
3.2 Booking Platforms using Blockchain Technology. 36
3.3 Booking Standardization and Implementations in Ethereum. 37

4 Project Definition **41**

4.1 Justification 41
4.2 Project goals 42
4.3 Methodology 43

5 Developed System **47**

5.1 Overview 47
5.2 ERC721 Ethereum Token Standard 50
5.2.1 ERC721Metadata Token Standard Extension 55
5.2.2 ERC721Enumerable Token Standard Extension 57
5.3 KibiwooProducts Smart Contract 58
5.3.1 Create New Product Use Case 64
5.3.2 Book Product Use Case. 65
5.4 BookingContract Smart Contract 66
5.5 Environment Configuration 78

6 Decentralized Application **81**
6.1 Introduction 81
6.2 Overview and Architecture 82
6.3 Decentralized Applications Implementations 84
6.3.1 Simple Dapp Implementation 84
6.4 Full Dapp Implementation 87

7 Results Analysis **95**
7.1 Results Analysis and Final Thoughts 95

8 Conclusions And Future Work **97**
8.1 Conclusions 97
8.2 Future Work 99

Bibliography **100**

A ANEXO A **I**

List of Figures

2.1 Git and GitHub Workflow Example 6

2.2 Centralized Ledgers vs Blockchain Ledgers Scenario. 8

2.3 Bitcoin Signed Transaction Overview. 11

2.4 Blockchain Structure Overview. 12

2.5 Description of the blockchain process for including a transaction. . . . 14

2.6 Public Key Cryptography Overview. 14

2.7 Block Header Formation. 16

2.8 Overview of Mining Process. 16

2.9 Broadcast of a new Founded Block. 17

2.10 Ethereum State Transition Function. 19

2.11 Overview of Ethereum client architecture. 21

2.12 EVM State Transition Example. 22

2.13 Transaction EVM Bytecode Example. 23

2.14 EVM Contract Bytecode Example. 24

2.15 Ethereum Architecture Design. 24

2.16 Ethereum Improvement Proposal Workflow 26

2.17 Redux Design Pattern 34

3.1 BTU protocol specification. 40

5.1 Smart Contracts Design. 49

5.2 Create New Product Use Case. 65

5.3 Book Product Use Case. 66

5.4 checkAvailability function test results. 78

5.5 Development Environment Architecture. 79

6.1 Decentralized Application Environment Architecture. 82

6.2 Metamask Functionality Examples. 83

6.3 Ganache-Cli Functionality Examples. 84

6.4 Simple Dapp Webpage. 85

6.5 Create a New Product Transaction. 86

6.6 After Booking Layout. 86

6.7 React App Initialization Process. 91

6.8 React App Landing Page. 92

6.9 React App Calendar Example. 93

List of Tables

2.1 Important EIPs and ERCs 27

CHAPTER 1

Introduction

This chapter introduces the context and motivation of the project, including a brief overview of all the different parts that are discussed. It also covers all the objectives that this project pursues and introduces the content that is covered in chapter.

1.1 Context and Motivation

Nowadays there are plenty of online platforms where we can buy or rent services and goods. These so called Marketplaces, have benefit from Internet and allow their users to compare many different providers and products just with one simple search. Nevertheless, while these platforms have provided added value to local stores and companies in terms of reach, global visibility and even market size, the dominance they have gained has led to a centralized scenario in which local stores have lost nearly all their negotiation power.

We are experiencing a situation in which due to the networks effects created by Marketplaces, the renting stores, hotels or any other type of services companies are totally subject to the limitations or conditions imposed by these big matchmakers.

This way, providers are struggling as commissions are getting higher and conditions are getting tougher, which ends in them needing to reduce costs. This fact, adversely affects the end user, as normally this reduction in costs by the providers means a reduction in quality of their services or products.

This way, although at first these matchmakers introduced a lot of advantages both to clients and providers, now the situation is being reversed and the size and importance these platforms have acquired is resulting in a negative impact. They are leveraging the dependency stores and clients have on them, to increase the returns they collect and impose some rules or conditions that are difficult for the providers to comply with. This way, the main winners from this scenario are only the online platforms while the other users are taking the other loser side. Even the rise in competition of marketplaces we have gone through in the last years has not been able to prevent this.

Nevertheless, we are going through a transition thanks to the new Blockchain technology. This new scenario created thanks to the emerging Blockchain technology, is characterized by enabling a decentralized environment in which dominant players such as the big marketplaces will not be able to impose so much power and will enable a more efficient market in which stores are not buried in commissions from every intermediary.

For this reason, our project aims in developing and deploying a series of smart contracts that live in the blockchain and benefit from its features, which implement a booking system that is backed by this decentralized technology. In addition, we will develop an application as a proof of concept of our implementation.

1.2 Structure of this document

In this section we introduce the different chapters presented on this document:

Chapter 1 introduces the project and its context. We explain briefly the structure of this document.

Chapter 2 introduces and explains the main technologies we have used on this project to achieve its goals.

Chapter 3 makes a study of all the related work done so far in the field. Presents advantages and improvements of the analysed projects.

Chapter 4 justifies why our project is valuable and what deficiencies detected in other works we will be solved by our system and why.

Chapter 5 describes in detail the design, architecture and development of the smart contracts that have been deployed to the blockchain. It also explains the procedures we have followed in our implementation and the standards we have used.

Chapter 6 presents the development and design of the Decentralized application that will interact with the smart contracts and will serve as a proof of concept of our design. It will show how shops can use the deployed system and in which ways they benefit from it.

Chapter 7 presents an analysis of the systems, the development process and the obtained results from the proof of concept.

Chapter 8 is a final overview of the project and achieved goals. Also future work for extending the system functionality is proposed.

CHAPTER 2

Enabling Technologies

In this chapter, we are going to describe the different technologies used to carry out this project. We will start explaining git and then we will dive into Blockchain and all software that enables the interaction with Blockchain techonology.

2.1 Git

Git is a free and open source distributed version control system [3]. As we are implementing a software project which will be developed following an incremental approach in different versions and deployments, each of them adding new content and functionality, we will use Git as our tool for tracking all this process.

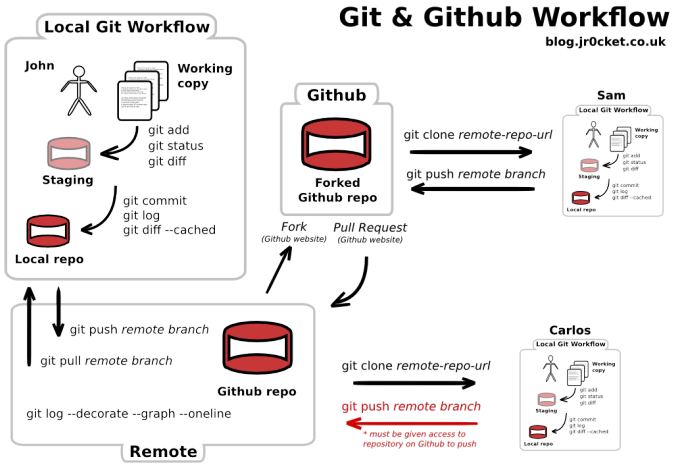
This tool will allow us not only to organise the incremental deployment but also to recover from failures, be able to have a stable version under one branch and test the new functionalities in another branch until completeness is reach. In addition, this tool will enable the collaboration between different people and make sure everyone is working under the same version. Also, we will be able to maintain independent the developed software and necessary libraries from the computer it is being implemented

on, in order to make sure that anyone who downloads the project is able to run it.

2.1.1 GitHub

GitHub is the Git hosting service implementation we have used during the development of the project. This implementation is one of the most widely used between developers and allows us to publish our git repositories to the cloud, making sure everyone can access it and download it with only an internet connection.

We can see an example of how GitHub works and its workflow in Fig.2.1.



(*Source:* http://jr0cket.co.uk/2013/08/getting-to-grips-with-git-understanding-the-simple-workflow.html.html)

Figure 2.1: Git and GitHub Workflow Example

2.2 Blockchain

Blockchain is a disruptive technology introduced in 2008 in Bitcoin's whitepaper: 'Bitcoin: A Peer-to-Peer Electronic Cash System' [15]. There were several publications an attempts before Satoshi Nakamoto's paper to implement a software solution that provided similar characteristics to a Blockchain. Actually, the concept that Bitcoin represents was conceived inside the Cypherpunk mailing-list [10] back in 1998. Nevertheless, it was not until Satoshi's publication when it became universally famous

and a real use case was put in practice, igniting this technology credibility and usage in other fields.

Initially, Bitcoin was no more than a distributed ledger which enabled the exchange of bitcoins, the unit of digital currency, between peers that participated in Bitcoin's Blockchain. Blockchain technology allows an environment with no necessity for intermediaries such as banks or other trusted third parties, which usually are in charge of assuring transactions' validity, as it provides with security and trust thanks to its consensus mechanisms based in cryptography. These algorithms allow the network to avoid the double-spending problem as well as control the order of money transactions.

Over a decade after, Blockchain has extended its usage, and multiple versions of the initial Bitcoin's proposal have been deployed with new functionalities until the point that Blockchain technology is considered the greatest software disruption and innovation since the appearance of Internet back in the beginning of the 21^{th} century [17]. We are experiencing a transition from the Internet as a source of Information to the Internet of Value [14]. Blockchain is one of the main characteristics and causes of this transformation regarding the way we perceive Internet's value. It is also one of the main foundations of the fourth Industrial Revolution.

Beyond the financial sector, Blockchain technology is and will revolutionize many other fields. Nowadays, there have been big changes and innovations regarding sectors such as healthcare, education, real estate, IoT, Supply Chain Management, or Electronic voting systems [11, 4]. In the future, much more areas will introduce Blockchain in their processes in order to improve their activities and services. As a fact, huge investment has been done in research regarding Blockchain technology and its applications.

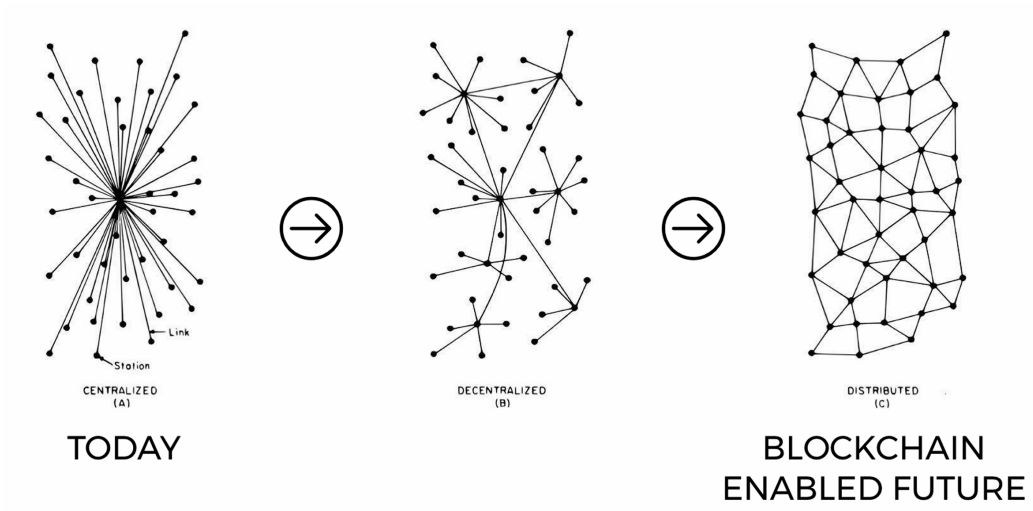
2.2.1 Blockchain Technology Description.

Blockchain is no more than a distributed ledger with some unique characteristics. A ledger is distributed when it has been securely replicated across several geographic locations or computers. This distributed ledger represents some kind of shared information and state. Blockchain uses blocks as the security mechanism for maintaining and updating the network state.

This way, we could explain Blockchain as a simple concept: **It is simply a distributed ledger, where transactions are stored in blocks and these blocks are chained together one after the other**. Every participant of the Blockchain can see both the transactions and the blocks. The way transactions and new blocks are introduced in the Blockchain is known by everyone and must be agreed by every member. This process relies in cryptographic algorithms in order to provide to the network with decentralization, security and trust at the same time. Thus, there is no necessity of a central authority that validates transactions.

A blockchain is a chain of chronological blocks. Each block is an aggregated set of data that is collected, processed and verified according to the consensus mechanisms. Every block is identified by a cryptographic hash of the information it contains and a timestamp. Each of them contain the hash of the previous block, so that blocks can form a chronologically ordered chain. This way, data in the blockchain is immutable. [1].

In Fig. 2.2 we can see how centralized databases differ from those proposed by Blockchain Technology.



(*Source:* Consensys Inc. Academy 2018)

Figure 2.2: Centralized Ledgers vs Blockchain Ledgers Scenario.

As a Distributed Ledger, Blockchain comes to a consensus of the state of the database through peer to peer protocols. These features are achievable thanks to the cryptographic infrastructure that Blockchain relies on. Depending on the Blockchain,

there are different consensus mechanism. We will focus in this section in explaining the basics, and later we will dive deeper in the specific characteristics of Ethereum, the Blockchain we have used throughout our project.

So, Blockchain as a whole is a distributed ledger formed by blocks that are chained together one after the other. Each of this blocks includes the transactions that are triggered by the participants of the network and that generate a change of the state of the Blockchain. This way, every block represents and image of the state of the Blockchain in a specific time. The way transactions and blocks are generated and validated are determined by the consensus mechanisms.

Consensus Mechanisms

Any distributed datasystem must define certain rules so that every participant has the same ledger and can agree upon the state of the network. Consensus mechanisms facilitate the agreement process among participants to guarantee data consistency. Then, the consensus mechanism sets the economic methodology implemented to achieve this goal.

As Vitalik Buterin, who is one of the founders of Ethereum, explains:
"The purpose of a consensus algorithm, in general, is to allow for the secure updating of a state according to some specific transition rules, where the right to perform the state transitions is distributed among some economic set."

There are several different types of Consensus Mechanisms. All of them share that they rely on cryptography for combating issues such as the double spending problem. They provide with robustness, integrity of data and trust in a trustless peer to peer network. Some of the most used consensus mechanisms are:

- ***PBFT:*** Practical Byzantine Fault Tolerance algorithm can stand about a third of its participants being dishonest or absent and still reach consensus.
- ***PoW:*** Proof of Work algorithm is based on repeatedly hashing the information of a block. Miners, who are the ones that hash the information of a block and introduce new blocks and transactions in the Blockchain, must guess and check hashes until a block hash less than a target number is found. This target hash is called block difficulty. This method is the one used by Bitcoin, but is known to be computationally expensive and cannot provide with high transaction volume.

- ***PoS:*** Proof of Stake algorithm is more efficient than PoW. In Proof of Stake, currency holders stake some currency for the chance to determine a block. Network nodes will accept blocks only from the selected validator for a specific time frame. The chance of being selected is proportional to the stake. Ethereum, although initially used the PoW algorithm, now has shifted to PoS.

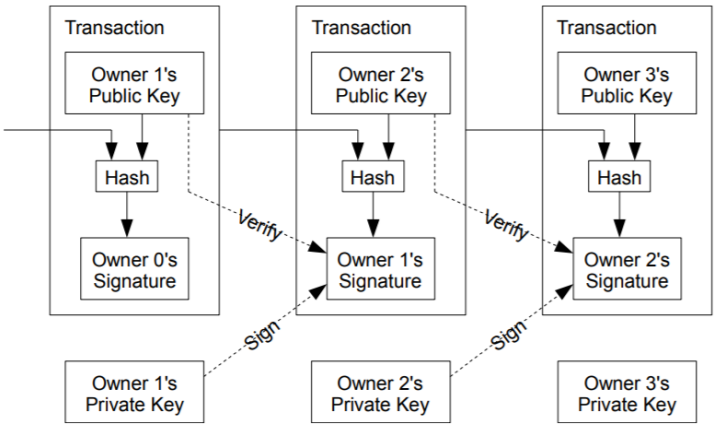
We can have different types of Blockchains depending on its characteristics and the permissions each node has. There are *three types of Blockchains:*

- ***Public:*** Anyone can join, read or write data in public Blockchains. Everyone is considered a trustless participant, but the proper consensus mechanism provides the network with trust so members can be confident. Nowadays, the most important Blockchains such as Bitcoin or Ethereum, are public Blockchains.
- ***Permissioned:*** Only certain participants of the Blockchain are allowed to participate on the network. There is also another version of this Blockchains in which anyone can participate in the Blockchain as regular nodes, but only certain nodes are allowed to validate transactions and introduce new blocks.
- ***Private:*** This type of Blockchain is typically the one used by private enterprises, or during the development and test of Decentralized applications, as we will show you through this paper. They are faster, provide with rapid app development and instant deployment. Participants are known and trusted.

Once we have understand what a consensus mechanism enables in a Blockchain and what type of Blockchains we have we are going to introduce the essential parts that form it. It is composed basically by:

- ***Nodes:*** These are the participants of the network. Depending on the Blockchain there can be different types of nodes. A computer can be turn into a Blockchain node by running the appropriate software. This software is called a blockchain client. Generally there are different versions of clients written in different languages, but all of them must comply with the rules specified by the consensus mechanism. Nodes are the gateway through which a computer, application, smart contract or any other entity can communicate with the network.

- ***Transactions:*** In simple words, a transaction is a signed message sent from one account to another account. A transaction can represent from simple transfers of digital currency between accounts, to the execution of multiple smart contracts deployed in the Blockchain. It is important to notice that a transaction is a **signed** data package. Transactions determine the change of state of the blockchain. Normally, transactions are grouped into blocks. We can see in Fig. 2.3 an overview of the structure of a valid signed transaction in Bitcoin.

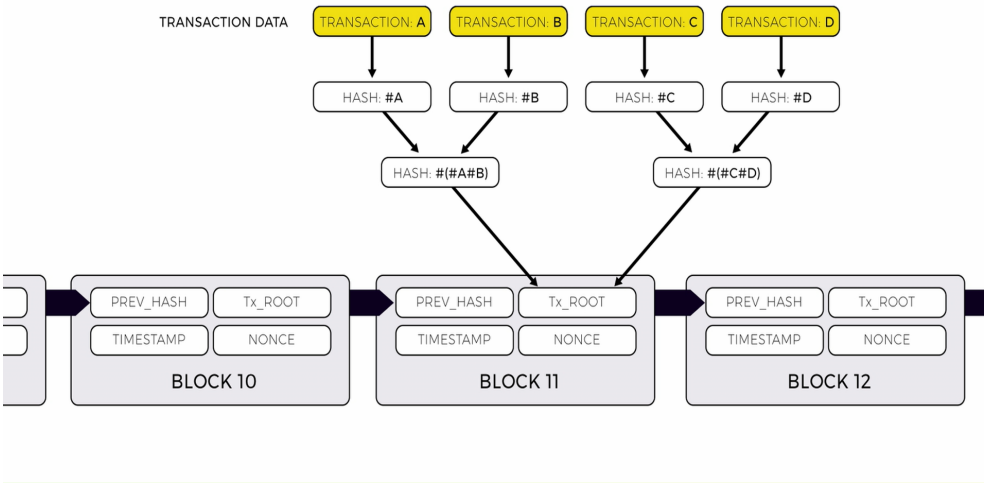


(*Source:* Bitcoin White Paper - https://bitcoin.org/bitcoin.pdf)

Figure 2.3: Bitcoin Signed Transaction Overview.

- ***Blocks:*** Blocks generally are formed by a block header and a group of transactions. The block header is a collection of relevant pieces of information that allow the blockchain to be chronologically ordered as well as secured. Blocks are the elements that are introduced into the Blockchain by miners. A transaction can be added to the Blockchain only as part of a block.

We can see in Fig. 2.4 an overview of the blockchain structure.



(*Source:* Consensys Inc. Academy 2018)

Figure 2.4: Blockchain Structure Overview.

Blockchain Features [20, 18]

Once we have briefly covered a Blockchain from a technical point of view, we can outline what all Blockchains have in common and what are their characteristics in terms of value to its users. Although there are several types and implementations of Blockchain technology, being this way Blockchain technology modifiable to fit specific purposes, there are several features that are common to every Blockchain implementation [1]:

- ***Consensus:*** Refers to the ability of all participants to agree in the state of the network according to some established rules. This rules are defined by the consensus algorithm.

- ***Distributed Computation:*** There is no central authority, computer or node in which the networks relies for maintaining the Blockchain or validating transactions. Every full node downloads a full copy of the blockchain which includes the data from all the transactions processed until the moment. Also, every full node can run independently, validate all old and new transactions and broadcast to its peers any work that is proved.

- ***Information Storage:*** All transactions included in the Blockchain cannot be erased, so transactions data will be stored in the blockchain forever.

- ***Provenance & Transparency:*** As every transaction is recorded in the blockchain, you can always track the origin of all digital assets that are traded or exchange through a blockchain. This is why blockchain technology has a high potential for supply chain applications. Every full node can audit transactions.

- ***Immutability:*** Once a transaction has been included in the blockchain, there is no way of altering or modifying the information included in that transaction, even if you made an error when sending the transaction. The only way to go back and reject a group of blocks in a blockchain is by making a fork of the blockchain. This must be agreed by the majority of the blockchain's participants. One example is the DAO attack in Ethereum, which ended in a fork of the Ethereum Blockchain [16].

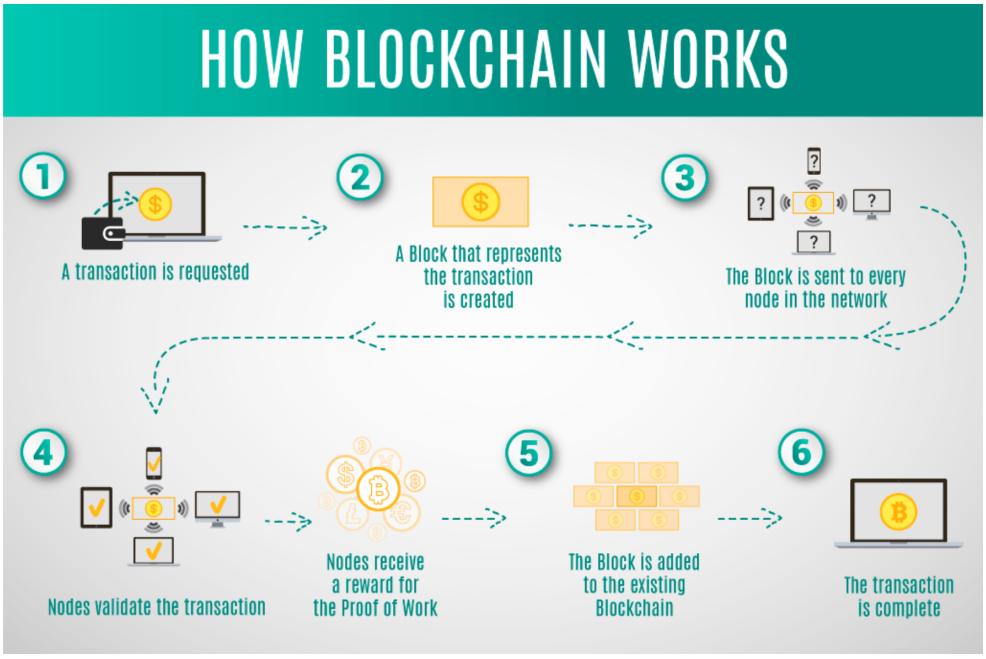
- ***Access Control:*** As we have seen, there are different types of Blockchains that can set different roles to distinct types of participants.

2.2.1.1 How a Blockchain Works.

In Fig. 2.5 we can see a description of how a blockchain works in terms of the process since a transaction is sent until it is included onto the blockchain. Specifically we show the process of a bitcoin transaction.

Public Key Cryptography

In order to be able for the participants of the network to communicate between them through transactions, while maintaining privacy of the information inside transactions, and allowing miners or any other participant to confirm those transactions without needing to know exactly the content of a transaction, public key cryptography or asymmetric cryptography algorithms are used. As we have seen in Fig. 2.3, all transactions are signed with the private key of an account and every other member of the network uses the public key to validate the signature of the transaction. This way, the network provides both with **privacy through encryption** and **trust** in a

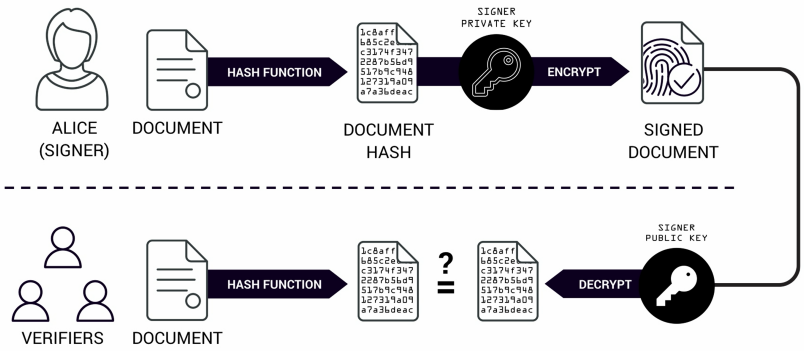


(*Source:* https://mlsdev.com/blog/156-how-to-build-your-own-blockchain-architecture)

Figure 2.5: Description of the blockchain process for including a transaction.

trustless environment **through authentication**. Normally, the public key coincides with the address of an account or can easily be derived from this address.

We can see in Fig. 2.6 how this process works.



(*Source:* Consensys Inc. Academy 2018)

Figure 2.6: Public Key Cryptography Overview.

As we can see, besides hashing the initial document so that the content that relies in the blockchain is the hash of the document instead of the document itself, that hash is signed using the private key of the sender.

Anyone that wants to verify the signature of the document can use the public key of the sender to decrypt the document and verify the signature. If, in addition, you want to check the validity of the original content, you can hash again the original document and compare the result against the decryption you obtained from the signed document using the public key of the sender. If they coincide, you can assure the equality of the documents.

Mining Process

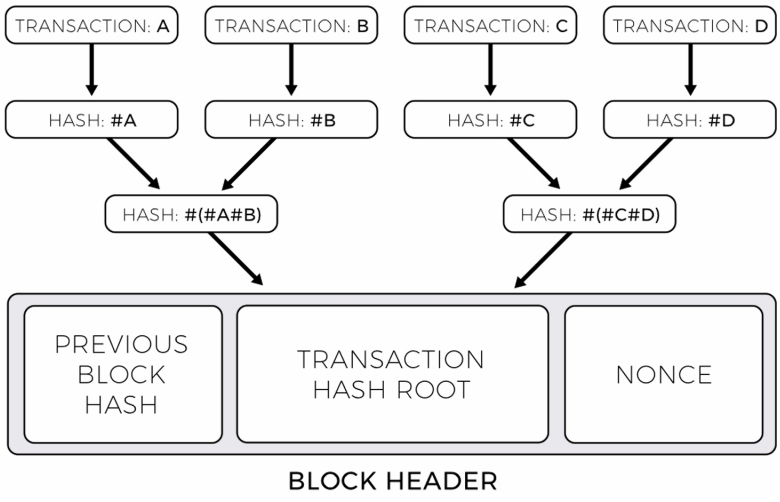
Nodes that run a full version of the software and try to include pending transactions in a new block chained to the last block of the chain are called *miners*. There are different mining processes depending on the cryptographic functions and rules a blockchain uses, but we are going to focus in PoW and PoS mining processes, as they are the ones used by Ethereum blockchain. Although these two differ in the way a miner is selected, the process of mining is nearly equal in both of them.

The mining process consist of constructing a block using the information from the last block included in the chain, the transactions that are going to be included in the new block and the information of the miner's account.

First of all, the miner should select the transactions that have been broadcast by users of the network. Normally the miner selects these transactions from a 'bag' of pending transactions. The selected ones should be processed and validated by the miner software. Once they have been validated, the miner should determine the new blockchain state and build the block header.

The block header usually includes metadata used for allocating the block in the chain and maintaining the chronological order. This metadata also includes the roots of the merkle trees that are used for representing the state of the blockchain in a specific time as well as save disk space (this is a big issue regarding public blockchains specially). Once the block header has been formed, it should be hashed. Now that the miner has the block, he should start what seems as a guessing game. We can see in Fig. 2.7 an example of how a block header is computed.

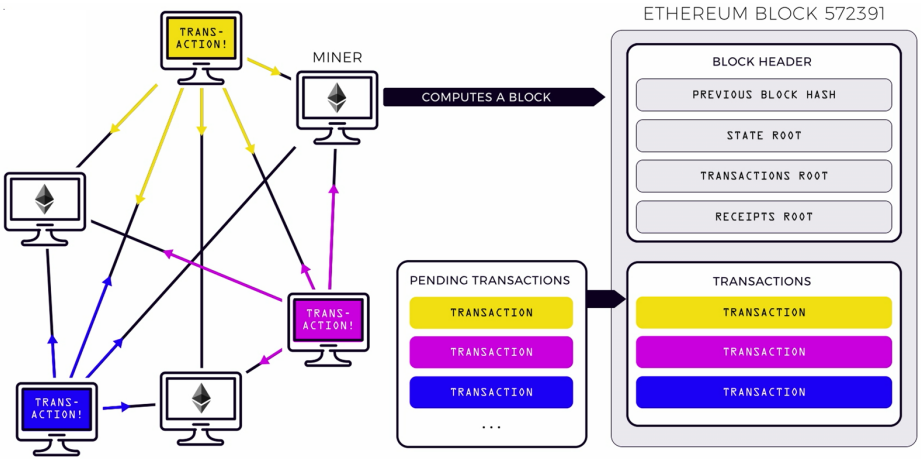
This guessing game consists of continuously hashing all the transactions with the block header and compare the result against a threshold. If the result is not less than the threshold, the process should be repeated. In order to avoid creating a



(*Source:* Consensys Inc. Academy 2018)

Figure 2.7: Block Header Formation.

block header again and processing other transactions, a field is provided in the block header, **the nonce field**, so that miners can only change this field while maintaining the rest of the block fields, so that the result of the generated hash of all the block content changes by only altering the nonce. We can see n Fig. 2.8 an illustration of this process.

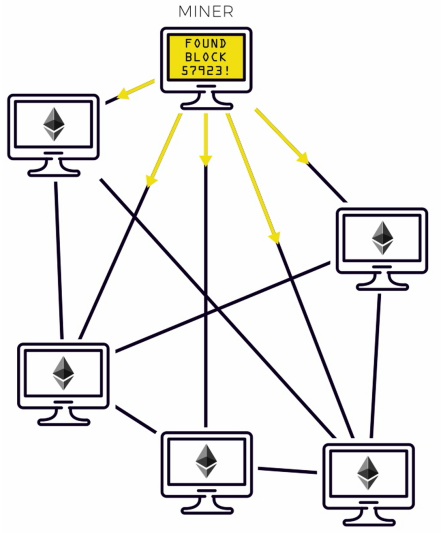


(*Source:* Consensys Inc. Academy 2018)

Figure 2.8: Overview of Mining Process.

So, this nonce field should repeatedly be changed until a valid hash of the block is found. Once a valid hash is found, the miner broadcasts the new version of the

chain with its founded valid block as the last block. Then all the others validators and participants of the network check that the this new block is valid and start mining a new block from this last founded block. The process of validating a new proposed block is really fast and computationally cheap while the process of creating and founding a new block really expensive in terms of computing power. We can see an image of this in Fig. 2.9.



(*Source:* Consensys Inc. Academy 2018)

Figure 2.9: Broadcast of a new Founded Block.

2.3 Ethereum

Ethereum is a **public** opened blockchain that introduces the concept of smart contracts inside the Blockchain. It was first introduced by Vitalik Buterin in Ethereum's white paper[7]. It extends the functionality of its ancestor Bitcoin, so that **transactions can represent** not only **exchange of digital currency** but also **any type of information or value exchange**. What is more, **applications can be deployed to the network** [6].

Ethereum can be seen as a transaction-based state machine. Its basic unit is the **account** instead of the digital currency as in Bitcoin. Every account not only stores a balance but a state also. And the world state is a **mapping between addresses** (160-bit identifiers) and **account states**. **Ether** is the token that is used

in Ethereum. A valid state transition is one which comes about through a transaction [19]. State transitions represent transfers of value and information between accounts.

2.3.1 Ethereum Accounts.

The Ethereum state is represented by the state of all accounts in Ethereum. Ethereum accounts are the basic object of the Blockchain and are identified by a 20-Byte address. Every account has a **persistent key-value pair** mapping 256-bit words to 256-bit words called **storage**. In addition, every account has a **balance in Ether**.

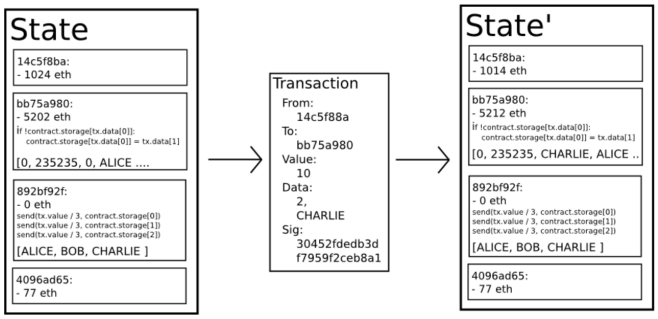
There are two types of accounts, **Externally Owned Accounts (or EOAs)** and **Contract Accounts**. Nevertheless, this difference is going to be removed in the next release. Both type of accounts have a nonce field, a storage and a balance. Contract accounts in addition have a codeHash which is the hash of the code it contains.

Every account is defined by a private key, a public key and an address. The state of an account is formed by only their balance in case of EOAs, and their balance and storage in case of a Contract Account. An EOA can send transactions and initiate transactions by their own whereas a Contract Account cannot initiate transactions. Their code is triggered by transactions or messages receive from other accounts. Although the code of a contract account can generate new transactions or call other contracts accounts, the origin transaction that initiated the sequence must be sent by an EOA.

2.3.2 Ethereum Transactions.

They are the method to update the state (either an ether balance or a contract storage). There are two types of Ethereum transactions, **MESSAGE CALLS** and **CONTRACT CREATION** transactions. Fig. 2.10 shows a transition state due to a transaction.

Every transaction specifies a TO address it sends to, unless it is a contract creation transaction, in which case the TO address is the ZERO address. If the TO address is an EOA address, the transaction represents simply a transfer of ether between two accounts. On the other hand, if the TO address references an account with code,



(*Source:* Ethereum White Paper [7])

Figure 2.10: Ethereum State Transition Function.

the transaction is triggering some function or part of code from that account. Every transaction contains [7]:

- ***Recipient:*** A TO address. If a contract is being created we introduce the ZERO address.
- ***Signature:*** Identifies the sender and is used for validation. It is diseminated in three values identified as *v, r and s*.
- ***Value:*** Represents the amount of ether to be transferred with the transaction. It can be 0.
- ***STARTGAS:*** Maximum amount of computational steps the transaction is allowed to take.
- ***GASPRICE:*** Value the sender is willing to pay per computational step.
- ***DATA:*** Optional field. Here we can specify what function we want to trigger from a specific smart contract, the input parameters of that function and additional values. The way we interact with Smart Contracts and its functions is by using this field. We need to use RLP-Enconding for specifying functions and parameters. [19].

Not every transaction costs Gas and Ether. If you send a message call to a contract just to read data or make some calculation that does not require or cause a change in the state of the Ethereum blockchain, then no gas or ether is required. This way you can make reads of contract storage without needing to spend Ethers in each read.

Gas and Fees

In the previous section we have introduced the concept of gas. Gas and fees are used as a way of controlling what users execute in the Ethereum Blockchain as well as a way of incentivizing miners. **Gas is the metering unit of the cost of each computational step.**.

Gas is paid for with Ether. Every executed operation in Ethereum has a specific cost or a specific amount of gas. Each computational step the Ethereum Virtual Machine does has a specific gas cost which is specified in the Ethereum Yellow Paper. Miners collect all gas used in a block as a reward.

This way, gas and fees protects the network as they reduce the spam broadcasted to the network, as it will cost ether to the attackers, and infinite loops are avoid as a transaction will run out of gas or the account that sends the transaction will run out of funds.

2.3.3 Ethereum Clients

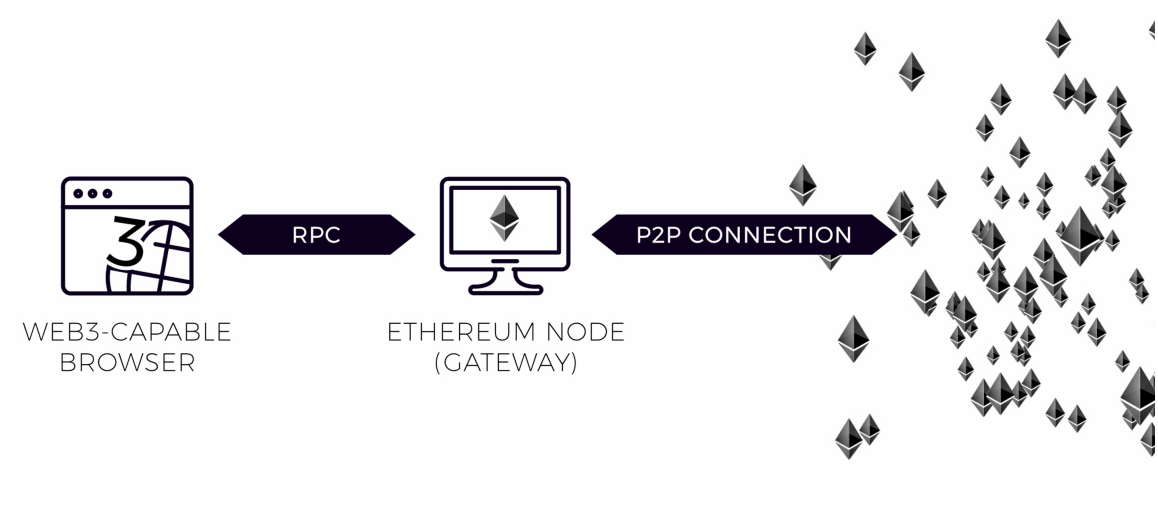
An Ethereum client consists of a node that installs a software application which implements the Ethereum specification and communicates over the peer-to-peer network with other Ethereum Clients.

There are many different clients written in various programming languages. All of them can interoperate as long as they comply with the Ethereum specification and the standardized communication protocols. The most used Ethereum clients are Geth [1], written in Go programming language [2], and Parity [3], written in Rust programming language [4] by Dr. Gavin Wood, one of the founders of Ethereum. There are many other clients such as cpp-ethereum, written in C++, or py-ethereum, written in Python [2].

A client is the Gateway through which any application, computer or any other type of Hardware can coommunicate with the network. It is not necessary to run

[1] https://github.com/ethereum/go-ethereum/wiki/geth
[2] https://golang.org
[3] https://www.parity.io/ethereum/
[4] https://www.rust-lang.org

a client locally as there are cloud solutions, such as Infura [5], that provide with an Ethereum node to which you can connect through an API. We can see an overview of how a client enables the communication with the Blockchain in Fig. 2.11



(*Source:* Consensys Inc. Academy 2018)

Figure 2.11: Overview of Ethereum client architecture.

In our case, we will run a client using the Parity implementation.

2.3.4 Ethereum Virtual Machine

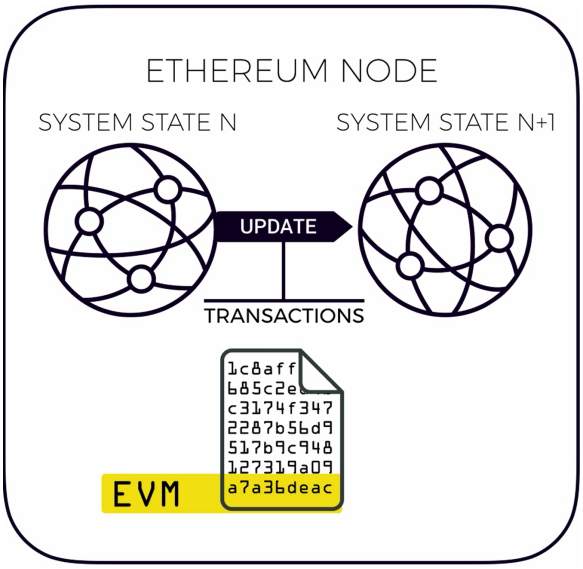
Thanks to the element known as the Ethereum Virtual Machine, Ethereum Blockchain can be seemed as a world computer. The Ethereum virtual machine has the following features:

- It runs in every node of the peer-to-peer network.
- Handles all the transaction processing to determine the current state of the network.
- It is a quasi-Turing-complete machine. This is because the computation is intrinsically bounded through gas [19], as we have already seen. It is a 256-bit words size machine.

[5] https://infura.io

- It operates on Bytecode.
- Is is relatively slow. It can manage between 10-20 transactions per second approximately.
- Every transaction is processed by the EVM of every node. Each operation must be processed simultaneously by every full node of the network in order to achieve a trustless environment.

We can see in Fig. 2.12 an example of how a full node would determine the state transition by running all the EVM Bytecode of the transactions contained in a block.



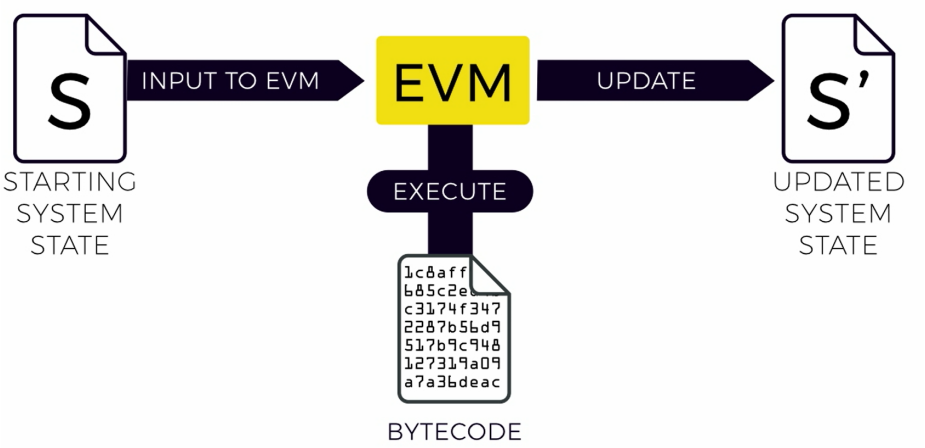
(*Source:* Consensys Inc. Academy 2018)

Figure 2.12: EVM State Transition Example.

The EVM is a simple stack-based architecture. It forms the key part of the execution model for an Account's associated bytecode [19]. When a transaction is executed, all the bytecode of that transaction or EVM code is executed by the EVM of each node. The EVM code represents all the computational operations that the EVM need to run in order to complete the transaction objective and change the state of the blockchain. We can see in the yellow paper, all the costs associated to each type computational step. This way, we can somehow approximate the gas cost of a specific transaction before it is executed.

The EVM is the main difference and advantage Ethereum has compare to Bitcoin. The EVM offers an execution environment through which the usage of the Blockchain can be extended to be able to handle and deploy smart contracts in the Blockchain. This makes Ethereum a worldwide distributed computer.

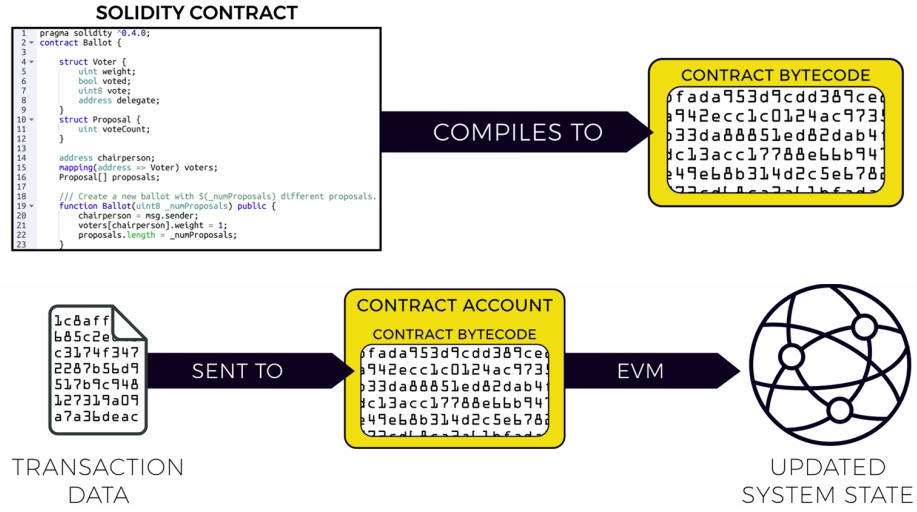
The EVM runs on Bytecode specifically designed for Ethereum. Bytecode is a low level stack based language which specifies how state transitions are applied to the netowrk's state. All transactions are translated into EVM bytecode, which is interpreted by the EVM as we can see in Fig. 2.13.



(*Source:* Consensys Inc. Academy 2018)

Figure 2.13: Transaction EVM Bytecode Example.

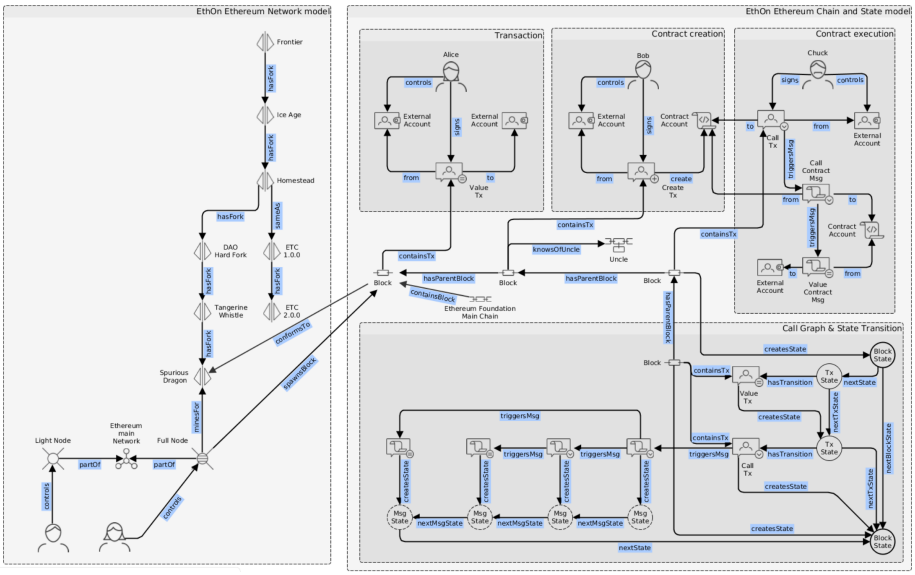
In order to provide instructions to the EVM, Smart Contracts exists as bytecode in contract accounts storage.The EVM executes the appropriate bytecode specified by RLP-Enconding as we have seen previously, when a transaction is sent to a contract with transaction data, and it updates the state of the system. We can see this process in Fig. 2.14.



(*Source:* Consensys Inc. Academy 2018)

Figure 2.14: EVM Contract Bytecode Example.

We finally show in Fig. 2.15 an overview of all Ethereum architecture, its elements and relations between them.



(*Source:* Consensys Inc. - EthOn ontology)

Figure 2.15: Ethereum Architecture Design.

2.3.5 EIPs - Ethereum Improvement Proposals

The yellow paper gathers all the formal specifications of Ethereum that a client must comply with. Although this specification is updated with the new releases, Ethereum counts also of an additional way of updating the specification. This way of including new standards or updating the existing protocols of Ethereum is called the Ethereum Improvement Proposals.

Ethereum Improvement Proposals are documented in a public github repository [6]. **The EIPs describe standards for the Ethereum platform, including core protocol specifications, client APIs, and contract standards** [9].

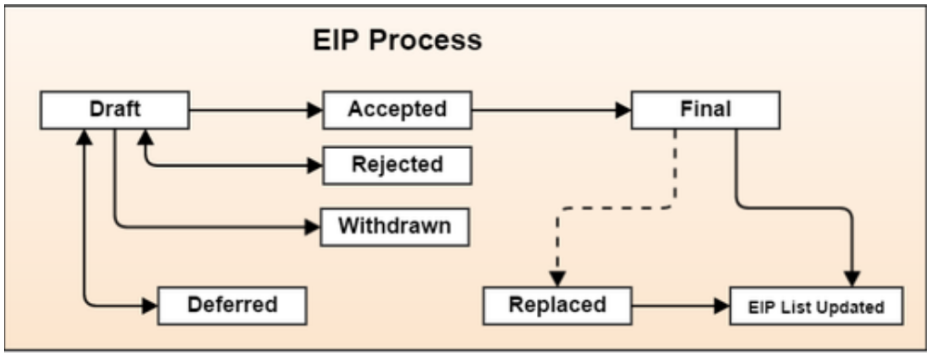
The intend and rationale of the EIPs is described in the first review of EIP-1 [7]. As described in this EIP:

"EIP stands for Ethereum Improvement Proposal. An EIP is a design document providing information to the Ethereum community, or describing a new feature for Ethereum or its processes or environment. The EIP should provide a concise technical specification of the feature and a rationale for the feature. The EIP author is responsible for building consensus within the community and documenting dissenting opinions."

EIPs are the main mechanism that developers of the Ethereum community have for proposing new features or standards, and provide a forum in which participants can discuss and come to an agreement of what are the specifications that need to be updated of the Ethereum technical specification. Any Ethereum developer is able to submit a new EIP, providing all the details of the proposal following a provided guideline for EIPs proposals. As they are maintained as text files in a versioned github public repository, their revision history is the historical record of the feature proposal. This way, implementers and other members of the community, are able to track the current status of their implementations. Every EIP goes through a workflow which is illustrated in Fig. 2.16, to determine if it becomes final or the community rejects it.

Eips are classified using numbers. Depending on the type of protocol an EIP refers to (Core, Networking, Interface, ERC), a different identifier is used. In our case, it is

[6] https://github.com/ethereum/EIPs
[7] https://eips.ethereum.org/EIPS/eip-1



(*Source:* [6])

Figure 2.16: Ethereum Improvement Proposal Workflow

important to outline the ERC EIPs. This type, refers to application-level standards and conventions, including contract standards. This standards are the way of Smart Contract and Decentralized applications developers of establishing standards that certain contracts should follow, so that applications and smart contracts can interact between them following a specific standard, instead of having every smart contract independent of each other, which would make the interaction really difficult and would limit the potential of using smart contracts in Ethereum. This standards allow interaction between different interfaces, dapps, smart contracts, wallets, etc. We can see in Table 2.1 an overview of the most important EIPs and ERCs. We will see that we will make use of ERC standards throughout the development of our project in order to make it more accessible and cross-platform.

EIP/ERC #	Title/Description	Author	Layer	Status
EIP-1	EIP Purpose and Guidelines	Martin Becze, Hudson Jameson	Meta	Final
EIP-2	Homestead Hard-fork Changes	Vitalik Buterin	Core	Final
EIP-5	Gas Usage for RETURN and CALL*	Christian Reitweissner	Core	Draft
EIP-6	Renaming SUICIDE Opcode	Hudson Jameson	Interface	Final
EIP-7	DELEGATECALL	Vitalik Buterin	Core	Final
EIP-8	devp2p Forward Compatibility Requirements Homestead	Felix Lange	Networking	Final
EIP-20	ERC-20 Token Standard. Describes standard functions as token contract may implement to allow DApps and wallets to handle tokens across multiple interfaces/DApps.			

Table 2.1: Important EIPs and ERCs

2.4 Smart Contracts

They can be defined as a computer protocol to digitally execute terms of a contract. Every Smart contract should be:

- ***Trustless:*** Two or more parties can act on an agreement without needing any kind of intermediary. Also it needs to be universally accessible. This means that anyone that forms part of the smart contract should not need to rely in anyone for participation.
- ***Trackable:*** All transactions should be able to be traced. This way, auditability is available and disputes can be resolved.
- ***Irreversible:*** Transactions must be final, so that participants cannot revoke their decisions. This feature ensures security of the smart contract.
- ***Self-executing:*** Smart contracts should be self-executing. This allows to reduce costs or increase speed for example.

Although the smart contracts existed way before Blockchain technology appeared, the perfect environment that Blockchain offers for Smart contracts has lead to a dramatic increase in the implementation and development of them. Referring strictly to a blockchain ecosystem, a smart contract is code stored on the blockchain which self-executes using the trust and security of the blockchain network.

2.5 Solidity

There are several programming languages for writing Smart Contracts in Ethereum. Solidity is the most used programming language for this purpose [5].

It is an object-oriented, high level language for implement smart contracts in Ethereum. It is really similar to Javascript and is specifically focused for the Ethereum Virtual Machine.

It is a statically typed programming language which supports inheritance, libraries and complex user-defined types such as Structs or Enums.

2.5.1 Open-Zeppelin

OpenZeppelin is a private company which builds developer tools for writing, deploying and operaing decentralized applications. It is a global benchmark regarding smart contracts development, and has set industry standards for building secure distributed systems.[8]

Openzeppelin provides open source implementations of ERC standards. In addition, they have built several well tested libraries which cover and implement many use cases which are really useful when designing and programming your smart contracts in solidity. This way, we can minimize the risk of vulnerabilities in our implementations and make them more secure. Considering that when a Smart contract is deployed to the Blockchain it cannot be modified, it is one of the most important aspects.

We will use implementations of ERC standards mainly, specifically we will focus in the different ERC-721 secure and well-tested implementations they provide. In addition, we will use other libraries that will provide us with additional functionalities that will be critical in our implementation.

2.6 Truffle Suite

Truffle suite is the most popular development framework for Ethereum. It will provide several tools which will make all the development workflow process of Smart contracts a whole lot easier. It is a development environment, testing framework and asset pipeline for Ethereum blockchain [9].

A world class development environment, testing framework and asset pipeline for blockchains using the Ethereum Virtual Machine (EVM), aiming to make life as a developer easier. Truffle provides with tools for:

- Network management for connecting to public or private blockchains.
- Built-in smart contract compilations and deployment.
- Interact with deployed contracts.

[8] https://openzeppelin.com
[9] https://www.trufflesuite.com

- Automated test environment using both Javascript libraries or solidity testing libraries.

- Configurable build pipeline with support for customized build processes.

2.6.1 Ganache

Ganache is part of the Truffle suite for developer tools for Ethereum and works great in conjunction with the Truffle Framework. [10]

It allows to quickly create a private blockchain which will provide us with an environment for rapid development and testing of our Smart Contracts before deploying them to an Ethereum testnet or the mainnet. It makes the whole process of deploying and testing our smart contracts totally free instead of needing to pay the gas associated to those transactions every time. We can run tests, execute commands and inspect the blockchain state while controlling how the chain operates.

It starts with 10 pre-funded accounts with 100 Ether per account. It can also easily be integrated and configured with Metamask and Remix. Additionally it includes a built-in blockchain explorer which we can use to inspect in depth transactions and blocks. Blockchain log outputs of the Blockchain can be viewed and it offers and advanced configuration of mining options.

2.6.2 Drizzle

Drizzle is the Front-End solution of Truffle suite. They are a collection of front-end libraries that make writing dapp user interfaces easier and more predictable [11].

It can easily be integrated with React. In fact, it is based on a Redux store. A Redux store enables us to have a dynamic webpage which quickly synchronizes the data according to the user actions. It makes it easier to interact with our Smart contracts through our Front-End and includes web3. In addition, it provides with a repository with commonly used React components which can easily be installed and integrated. It is based in a modular architecture.

[10] https://www.trufflesuite.com/ganache
[11] https://www.trufflesuite.com/drizzle

2.7 WEB3.js

WEB3.js is a collection of Javascript libraries that allow you to communicate with local or remote Ethereum nodes using an HTTP or IPC connection. It makes interaction with the node much easily as it provides an API that automatically is converted to JSON-RPC for communicating with the node and interacting with smart contracts [12].

WEB3.js has all of the functionality of the node that is connected to. It makes it easy to access node functions from within your applications by making a WEB3 object available in your Javascript files. It offers a collection of modules which contain specific functionality for the ethereum ecosystem.

2.8 MetaMask

Metamask is simply a Chrome and Firefox browser extension that stores accounts and private signing keys. It stores them locally in your computer instead of storing them in a database in the cloud, this way users have total control over their keys [13].

It directly connects to an Ethereum gateway. You can configure which specific gateway you want to connect to, the mainnet, the Ropsnet Testnet or even a localhost port that a private blockchain is listening to. What is more, Metamask injects the WEB3 object into the webpage so that it can interact with the Ethereum gateway it connects to. It makes really easy some actions such as creating accounts, funding them or sending and signing transactions through a really simple front-end. We can even deploy contracts by sending a transaction with the compiled contracts as the content of the data field and the ZERO address as the TO address of the transaction. When interacting with deployed contracts, whenever the page requests a transaction to be signed, MetaMask will display a prompt asking you to approve the transaction. This way, it makes all the interaction process with smart contracts in the blockchain easy, even to people that do not have a technical knowledge of the Ethereum blockchain.

Metamask acts as a bridge between your application front-end and the gateway that connects to the blockchain.

[12] https://web3js.readthedocs.io/en/v1.2.1/
[13] https://metamask.io

2.9 JavaScript Libraries

JavaScript is a high-level, interpreted scripting language with object-oriented capabilities [13]. Regarding web applications, JavaScript is the element that enabled the transition from static webpages to dynamic webpages. It enabled webpages to interact with servers, databases or other elements. It is the most used programming language among web applications.

Although it was mainly used in the front-end part of web applications, the appearance of NodeJS [14] extended its use to the server side also. So now, you are able to develop a complete web application using JavaScript, with no requirements of other server side programming language.

There are multiple libraries that have been written in JavaScript for different purposes. Testing has been widely covered using Javascript and multiple libraries have been develop towards this goal. We are going to use JavaScript not only for our Decentralized application but also for testing in depth our Smart Contracts.

2.9.1 TDD/BDD Coding Practices

As we have already outlined, all applications should be extensively tested before final deployment, but this aspect is even much more important if we are talking about smart contracts in a blockchain. The deployed smart contracts must be deeply tested and checked for possible security vulnerabilities, as if not it could ended in all our clients loosing their funds or digital assets.

This is why we approach a TDD/BDD coding style. TDD/BDD stands for *Test driven Development* and *Behavior Driven Development*. These are test-first development approaches. With TDD, you write first tests so that they fail, and then write the necessary code in order to make them pass. If necessary you refactor the code so that it does not causes already written tests to fail. This way every line of code that is written is tested also. This method focuses in functionality tests, but does not cover behaviour or specific situation test cases. BDD on the other hand, extends TDD ideas and combines them with software design methods and business processes

[14] https://nodejs.org/es/

use cases in order to improve sorftware development. With BDD we do not test only aisle functionalities or functions, we run tests of scenarios and interoperation of the different functions and classes in order to fullfil several real possible use cases.

There are several JavaScript libraries for this TDD/BDD development approach but we will use the following two as they are the most used ones, their community is really big and they are well documented.

2.9.2 MOCHA

Mocha is a feature-rich Javascript test framework running on Node.js and the browser [15]. It allows us to run asynchronous and synchronous tests really simple. In addition, it offers several tools for the execution and debugging of tests. It allows us to design our tests in a really ordered and structured way.

2.9.3 CHAI

Chai is a BDD/TDD assertion library for node and the browser also. It provides multiple interfaces and allows us to make tests using an easy human readable language. Multiple plugins for Chai framework are available for download and integration. It has several interfaces, such as *assert*, *expect* and *should*, which allow the developer to select the style it adapts better to its requirements.

We will use Mocha for setting the structure and environment of our tests, and Chai for the specific tests.

2.10 REACT

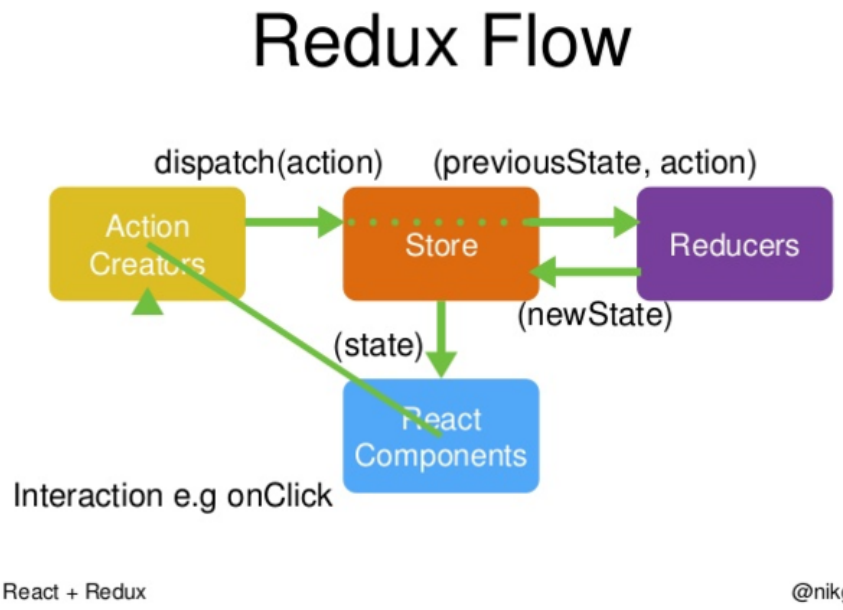
React is a really powerful and extensively used Javascript library for building user interfaces [16]. It focuses in interactive user interfaces that create dynamic webpages that are constantly changing and reacting to user actions.

It fosters a modular approach through which we can separate our user interface

[15] https://mochajs.org
[16] https://reactjs.org

in simple components and then compose the entire webpage by simply adding and integrating those components. Additionally, it enables the communication of those components through variables, props and states.

It is also easy to learn and not very slow. It is recommended also to use JSX files. JSX is an extension of Javascript that allows to run Javascript code throughout the html webpages. We can use React in combination with Redux for managing the state of the application. Redux allows the user interface to update only specific elements that are altered due to a user action instead of needing to reload the entire webpage. We can find a basic architecture of a Redux component in Fig. 2.17.



(*Source:* https://jobs.zalando.com/tech/blog/design-patterns-redux/?gh_src=4n3gxh1)

Figure 2.17: Redux Design Pattern

CHAPTER 3

State of the Art

In this chapter, we describe approaches, technologies and applications that have been implemented in related works. We will first talk about already existing solutions for implementing a booking system using blockchain technology and end talking about current proposals for standardization of renting and booking implementations in Ethereum.

3.1 Introduction

The booking industry has continuously risen during the past years. This is specially due to the change of mind young people have experienced and the possibilities internet provides for all type of renters. The idea of ownership is not so well-established among young generations such as millennials. Instead, they embrace more the idea of sharing, renting or booking items, which has led to the appearance of new booking sectors and the revolution of others. This is one of the main reasons why renting or sharing marketplaces such as Airbnb, BlaBlaCar, Booking or Uber have rocketed in the last years both in terms of number of users and market value.

Nevertheless, the vast majority of them act only as an intermediary platform

between renters and guests. The appearance of Blockchain is changing the current paradigm, as it enables the lack of intermediaries, and it has caused this 'big fishes' to review their business models. We will go briefly through already proposed solutions concerning booking systems running on the blockchain. Finally, we will cover all the current standards for enabling booking platforms to migrate to the blockchain and benefit from its features.

3.2 Booking Platforms using Blockchain Technology.

Nowadays, there are several use cases that have been tested and deployed to a blockchain. It has been proved that it is possible to run a booking solution in the blockchain. Although several different solutions are available, most of them focus in hotel booking systems. Instead of being more opened, they strictly focus in this sector without trying to extend their functionality so that it can fit other areas. Some of the decentralized applications we refer to are:

- ***Noctorus*** [1]***:*** A blockchain hotel booking middleware. They are still raising funds through an ICO. They have partner with big companies of the hotel sector such as Marriot, Renaissance or the Ritz-Carlton. They offer a middle pace between a total decentralization system that runs completely on a blockchain and the current scenario, through a middleware that hotels can install in their webpages and interact with smart contracts leaving still some parts of the process off-chain. Anyone that wants to implement their solutions should buy noctorus tokens.

- ***Coze*** [2]***:*** Is a decentralised hotel booking and rewards platform which uses a protocol powered by an Ethereum based token system. Again, is necessary to have Coze tokens for being able to participate on the system.

- ***Locktrip*** [3]***:*** Offers a hotel booking and rental marketplace that runs on the Blockchain with 0% commissions. They run their own Blockchain that uses Proof-os-Stake as their consensus mechanism.

[1] https://us.nocturus.com
[2] https://coze.io
[3] https://locktrip.com

- ***HotEngine*** [4]**:** Makes it simpler by offering an engine which in the background is using decentralized technology supported by a blockchain. This engine is easily embedded to any html web site, making it really simple for rental shops or companies for installing their software. Although this is one of the most extensible and general implementations, the demo they have deployed focuses also in hotel booking sector.

There some other implementations and work in progress. Nevertheless, we can see how many of the solutions focus in hotel booking systems. In addition, most of them require the usage of their own tokens, which although it is true that they are decentralized approaches that get rid of middlemen, you are still bounded to their tokens. What is more, some of them are implemented in their own blockchains which cannot be so reliable as public blockchains such as Ethereum.

3.3 Booking Standardization and Implementations in Ethereum.

As we have seen, Ethereum has a way of proposing standardization of processes for building more valuable applications that can interact with each other and allow the usage of several tokens or other implementations that comply with particular standards, by Ethereum Improvement Proposals or EIPs. We already have important token standards such as the ERC20 or the ERC721 which allow all these tokens to be exchangeable among other usages. Although still there is not an accepted and final Ethereum standard for the booking process, there have been several proposals that we are going to cover here.

ERC-809: Renting Standard for Rival, Non-Fungible Tokens

One of the most used and important standards of Ethereum is the ERC-20, which allows tokens that comply with this standard to be exchangeable. Nevertheless, this are fungible tokens. With the objective of having a similar standard that could be used with real assets and non-fungible tokens, which represent each of them different

[4]https://hotengine.co

assets and are all of them distinct, it was created the ERC721 Ethereum standard. This sets an interface for non fungible tokens.

The ERC-809 EIP, as its name describes, tries to extend the functionality of the ERC721 standard including functions that enable the booking or renting of these tokens and what they represent. This issue was opened in the ethereum Github on the 27th of December of 2017 by Github user slee981. [5]

As the abstract of the issue states, "This EIP suggests a standard API for renting access to rival non-fungible tokens within smart contracts." This way what is being tokenized for renting purposes is the access to a specific rival NFT or Non-Fungible token that may represent a real asset. This issue was discussed and slightly edited until the 2nd of September of 2018. Since then, no more comments or additions were done and it was not included in EIP official repository so it was not accepted. Along the comments we can see some example implementations of the interface such as `https://github.com/BookLocal/BookLocal-contracts-v2.0`.

ERC-1201: Two Tiered Token Structure for Non-fungible Asset Ownership and Rental Rights

This new issue came motivated from its predecessor, the ERC-809. It was opened in the 6th of July of 2018 by Github user zemingyu. The innovation that introduces with respect to ERC-809 proposal was the differentiation in two tiers of the ownership rights and the rival rental rights. Unlike ERC-809, these rights are tokenised introducing the possibility of exchanging not only the ownership rights but also the rental rights [6].

The last comment on the thread was posted on the 16th of October of 2018, and as the other one, never made it as an official ethereum standard. Some examples of implementations of this EIP were provided, such as `https://github.com/vincentshangjin/uhoodchain/blob/erc1201/contracts/uhood.sol`.

BTU Token Protocol

BTU Token protocol or Booking Token Unit protocol is a building block for any

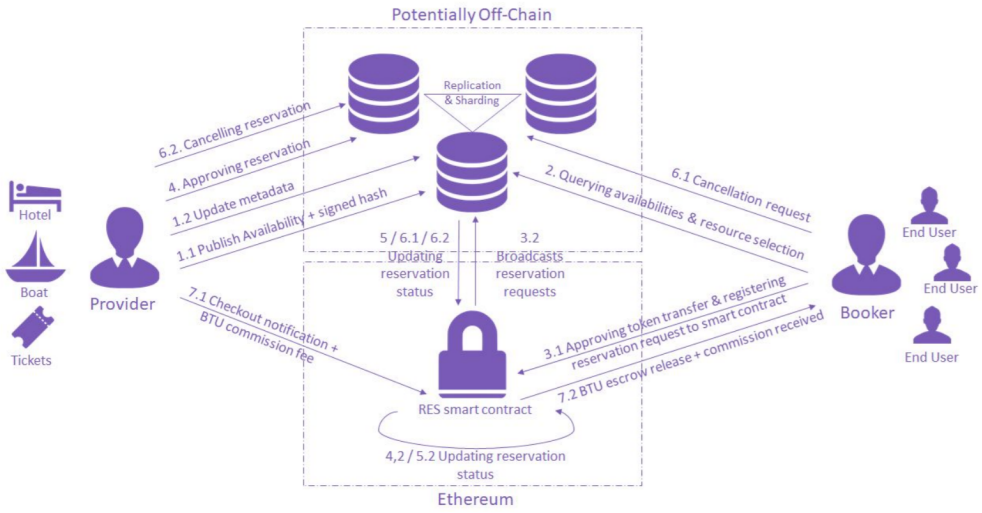
[5] https://github.com/ethereum/EIPs/issues/809
[6] https://github.com/ethereum/EIPs/issues/1201

decentralized application (dApp) or web site willing to implement booking features for their end-users [8]. It is also an issue in Ethereum EIPs github under the name of ERC-808. It was opened on the 26$^{\text{th}}$ of December of 2017, first with the number 770 by Github user appyhour770, but then was changed to EIP-808 as a requirement of ethereum. The official name is 'EIP 808 - ERC for a Standard Decentralized Booking Protocol [7]. The last comment was posted on the 10$_{\text{th}}$ of May of 2019. It was reviewed on the 5$^{\text{th}}$ of November of 2018 and all checks passed. Currently, appyhour770 has made a pull request for including it on the official EIPs repository but still has not been included.

This is the most serious implementation regarding standardization of booking processes in Ethereum as it has not only been an open issue in Github, but there has been a lot of work and progress done externally to this issue. Use cases regarding Hotel Bookings and Tours Bookings have been implemented and there are available demos for testing them and a whitepaper with all protocol specifications has been published. We can see an illustration of the specification in Fig. 3.1.

It is true that it is a really interesting approach and thinks not only in the technical aspect but also covers business aspects. It offers an hybrid solution in which some actions are done off chain and others rely in the blockchain technology. It provides also with an easy way of integrating the system in existing solutions or webpages although this implementation is still under tests. Nevertheless, the downside of this approach is the necessity of relying in BTU tokens, which are ERC-20 Ethereum compliance tokens that are used as the currency for the renting system. This ERC-20 token, are tokens controlled and issued by the company behind the BTU Protocol.

[7] https://github.com/ethereum/EIPs/pull/808



(*Source:* BTU Protocol Whitepaper)

Figure 3.1: BTU protocol specification.

CHAPTER 4

Project Definition

In this chapter, we will explain the motivation and purpose of our project. We will answer questions such as why our project is different from the actual implementations and what additional value provides. We will expose a list of goals to be achieved during the development of the project and we will describe the methodology that we will follow for the elaboration.

4.1 Justification

We have seen several implementations that focus mainly in Hotel Booking systems. These platforms and versions of decentralized booking systems not only have not been tested under other use cases but do not make use of the public well-tested blockchains such as Ethereum. They use in excess private tools which put in doubt the decentralized aspect. What is more, they are really bounded in terms of interoperability, scalability and standardization.

On the other hand, there have been some applications that have focus on utilizing the Ethereum blockchain, which is a far better decision. Nevertheless, most of these

have focused only in trying to develop a smart contract that squeezes the most of smart contracts and decentralization, but have forgotten the business part. They should have made a requirement analysis from a business perspective, and try to adopt a solution which makes the transition to the Blockchain of already existing platforms easy and progressive, instead of making a huge jump. In addition, the first two described implementations have reach a dead end and finally ended only being failed proposals. The last proposal covered, promotes a solution which is thoughtful for existing platforms that want to migrate to decentralized solutions and also has been well tested. The problem with it is its interoperability with other elements of the blockchain, as it relies in using the BTU protocol which is a ERC20 token that is controlled, distributed and minted by the company that developed the BTU protocol.

This is why we want to reach a solution that combines the best part of each of these implementations and takes advantage of blockchain features while maintaining an eye on the business necessities and possibilities. We will develop a series of smart contracts which do not rely in any private token as the currency within the system, and which prioritizes the use of existing ethereum standards such as ERC20 or ERC721 in its design and smart contracts, so that the resulting system is interoperable and does not rely excessively in any component. We will take into account economic aspects for deciding what parts must operate exclusively on the Blockchain and what parts should be maintained off-chain.

4.2 Project goals

The final purpose of this end-of-master's project is the design and development of a series of Smart Contracts in Ethereum that allow the migration of a renting system, in which there is an interaction between two or three different types of users, to a decentralized solution that relies in Blockchain techonology.

In addition, we will try to comply and use existing standards for Ethereum Smart Contracts, in order to build a more flexible, reliable and trusty software. Although there are several proposals regarding booking standards for Ethereum Smart Contracts, none of them has been globally accepted by the Ethereum community, so we will try to avoid using those proposals as they are still work in progress.

This goals will be achieved after all of the following sub-goals are completed:

- Design and Development of Smart Contracts using Solidity programming language.
- Study and implementation of ERC-721 Ethereum Standard.
- Fully test all Smart contracts by using TDD/BDD coding style.
- Set-up and configuration of a local Blockchain.
- Dapp development for interacting with the deployed Smart Contract via a Front-end.
- Implementation of a real use case using the Dapp.
- Set-up and configuration of an Ethereum Full-Node using Parity.
- Deployment of the Smart Contract in Ropsten Ethereum Testnet.
- Adjustment and deployment in Alastria Blockchain.

4.3 Methodology

We have followed an agile software development methodology, specifically we have adopted a Scrum process. We have planned several sprints, each of them with certain objectives and at the end of each one we should delivered a final product that could be tested somehow. Every week, the work and process state was reviewed. This way, we were able re-adapt rapidly the objectives and identify possible problems that emerged during the project, as well as what solutions or decisions were made to rapidly resume the development. We have also used Github as a tool for controlling the versions of the project and checking the incremental additions to it.

We considered 4 sprints initially, with the following general objectives for each of them:

- During first sprint, we did a research and study of the available solutions for decentralized booking systems in the Blockchain. In addition, we went through

the Ethereum standards and determine which could be used and were suitable for the scope of our project as well as learning in depth all Ethereum architecture and functionalities. The final deliverable for this sprint was a summary of the available solutions and a description of which were potential usable standards for our work.

- Second sprint objectives focused in the set-up of a development framework for the implementation of Smart contracts and the set-up of a private blockchain for being able to deploy those smart contracts. We went also deeply through solidity documentation for acquiring all the necessary knowledge and coded following best practices of coding styles and comments. The goal was to develop several Smart Contracts that implemented the ERC-721 Ethereum standard, deploy them to our private Blockchain and have those Smart contracts implement the minimal amount of functions and functionalities so that we were able to have a Minimal Viable Product (MVP) that we could deploy to the blockchain and interact through a console with it.

- Third sprint is going to be the last to be fully completed. It was focused in extending the smart contracts functionality, and refactoring them in order to implement all the available smart contracts and libraries such as the ones from openzeppelin. In addition, we would extend the way we interacted with the smart contracts by developing a simple Dapp (Decentralized Application). We developed a Front and Back-End using React and Web3.js so that we could interact with the Blockchain through an application running on the browser. This aspect required some more learning of React, dive deeper in Javascript, and get to know well Web3.js library. This Dapp includes minimal functionality to test the core of our project. In addition, we will set up an Ethereum node using parity and deploy our smart contract to the Ropsten Ethereum Testnet, and then connect our Dapp first to the private blockchain and then to the Ropsten Testnet.

- Finally, last sprint objectives focuses in extending the functionality of our Dapp to fully implement a booking Marketplace and connect it to the Ropsten Testnet. In addition, after we are 100% sures that it works perfectly we will start the migration of the Dapp and the smart contracts to Alastria's Blockchain. This sprint will not be finished by the presentation of this project due to some

delays in the previous sprints and the decision of testing and developing more the dapp in a private blockchain and the Ropsten Testnet before migrating it to Alastria's network.

CHAPTER 5

Developed System

In this chapter, we will cover the design and implementation of the smart contracts. We will explain the purpose of the system, our approach for enabling booking services in the Blockchain, how every element integrates with the system and what value we are providing with it. We will go also through several standards and libraries that we use and will give an idea on how our implementation may be useful for several use cases, not only the one that we will deploy or a Hotel Booking system, but to plenty of other case studies.

5.1 Overview

As we have outlined through this document, our idea is to implement a solution that has as its primary goals standardization, co-integration and business value. Following these requisites, we have taken some ideas provided by already deployed systems, as the ones we have seen in Chapter 3, and design an architecture that includes the following characteristics:

- Uses existing Ethereum Smart Contracts standards, transforming our system in an interoperable infrastructure.
- Include well tested Solidity libraries, providing higher security to our code.
- Development done with business requisites always in mind.
- Provide an easy way to connect already existing applications.
- Leave the door open for implementing simple interfaces which allow connection of universal software tools such as Wordpress.
- Economic aspects have been studied, and expenses related to Ethereum transactions have been reduced.

It is important to differentiate between what smart contracts and the Decentralized Applications or Dapp provide. The smart contracts will enable to migrate the business logic of a booking system from a traditional centralized and independent environment to a decentralized environment based in Blockchain technology. On the other hand, a decentralized application is a system, webpage or any other kind of application that provides a user interface for clients to search and book products, and which interacts with the deployed smart contracts for being able to provide a booking service.In this section we will focus only in explaining the smart contracts, not how a Dapp interacts with them, as we are going to explain that part in Chapter 6. Our smart contracts must be able to handle any type of the Dapp that need a booking service logic in the Blockchain. This way, we will try to generalize the software instead of bounding it to a specific use case.

This approach, although it may seem that it will not provide with all the necessary services for specific case studies, following our requirements is the best design. We have preferred to include in our smart contracts the minimum code for enabling the booking system logic while leaving specific functionalities for off-chain solutions. This way we reduce the complexity of our Smart Contracts as well as we reduce the cost of the transactions in the Blockchain, while maintaining the minimum required functionality so that the essential processes and information for supporting the booking functionality is stored in the Blockchain.

Having said this, we present in Fig. 5.1 a general design of the Smart Contracts that will be described in depth throughout this chapter.

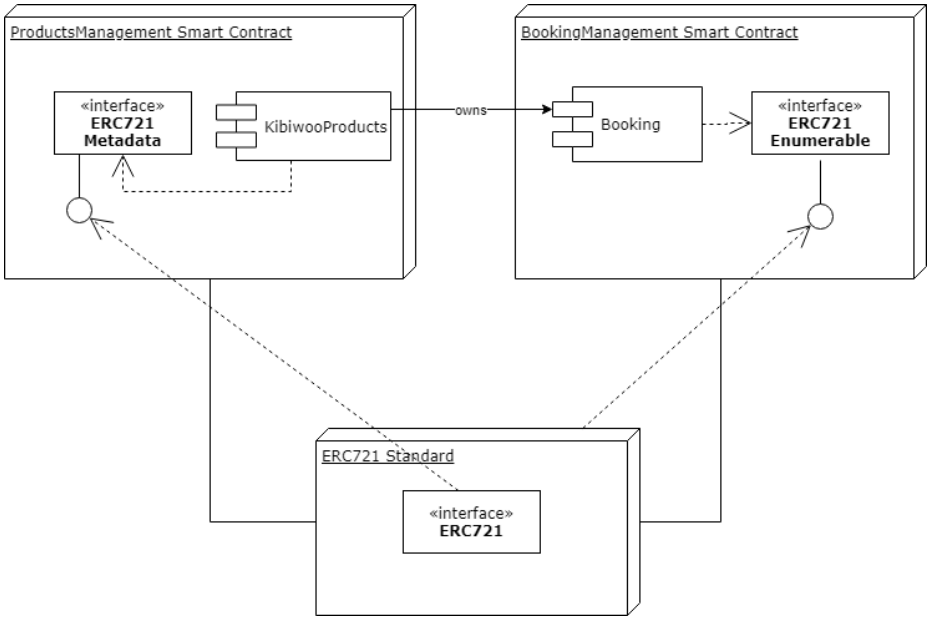


Figure 5.1: Smart Contracts Design.

As we can see, we will differentiate between two main modules or smart contracts, both of them implementing the Ethereum Standard ERC-721. The products management module, will be in charge of identifying and tracking products. Each of the products will have an associated ERC721 token which represents the ownership of the specific asset the token refers to. On the other hand, each of the tokens within the products' management smart contract will be associated to a different booking smart contract. The booking smart contract will also conform with the ERC721 Ethereum standard, but its tokens will represent reservation rights of the asset the booking smart contract is associated to. Each token will represent a block of time for booking a product.

This way we separate the managing and ownership of the products from their reservations rights. This allows us even to set specific configurations for each token such as how much a block of time represent (for a house it may be a day while for a surfboard it may be an hour), cancellation policies, etc. We have followed the approach suggested in the 1201 EIP github, but introducing some changes to it. The decision of differentiating between ownership tokens and rental rights tokens has been crucial. If no separation was made, if a token that represented a specific asset was transferred to a client that rents that asset for a specific amount of time, during that time, the client will be seen by the system as the owner of the asset when in

reality this was not the case. Having that token even only for a small amount of time, will enable that client to modify the asset and make other actions that should be restricted to the owner. So although it could also be introduced the idea of roles with specific restrictions to what each role can do inside the Smart Contract, the decision of generating a separate ERC721 token that represents rental rights, is simpler, cheaper and allows those tokens also to benefit from complying with the ERC721 standard.

In the following subsections we will cover the used interfaces and standards and we will explain in detail each of the types of smart contracts developed.

5.2 ERC721 Ethereum Token Standard

The ERC-721 Ethereum Standard was inspired by its predecessor the ERC-20. The ERC-20 represents fungible tokens, that can be transfer totally or partially and with each of them equal to each other. Nevertheless, when talking about assets that cannot be exchange partially and all of them are unique, the ERC-20 does not work. So it appeared the ERC-721, also known as the Non-Fungible Token Standard [12].

As the official eip website describes:

"The ERC-721 token standard allows for the implementation of a standard API for NFTs within smart contracts. This standard provides basic functionality to track and transfer NFTs."

NFT is the acronym for Non-fungible tokens. They are also refer to as deeds. A NFT can represent any type of asset, either physical or digital, and these assets are all of them distinct. These tokens, can represent ownership over them or other type of things such as rental rights. We can see the flexible nature of this type of tokens in our implementation, as we have one type of NFTs that represent the ownership of the actual products being rented, and other NFTs with each of them representing a specific block of time associated to a certain product. Ownership of the second type of tokens, provides you booking access to that product in that time. This way we clearly differentiate between owning the NFT and renting it over a period of time.

As our smart contracts conform to the ERC721 token standard, this enables our tokens to be exchangeable, as well as allows wallets brokers or even auctions applications to work with our tokens, boosting the value of our smart contracts. This is one

of the best applications of composing token standards, as they provide your smart contracts with free interoperable infrastructure. We will have both NFTs, the ones that represent ownership and the ones that represent the proper bookings, to automatically be exchangeable and tradable by numerous wallets, token exchanges and many other applications, as well as give rights to other Ethereum accounts to manage the tokens also. This way, we proof how reusing existing libraries and implementing token standards is best decision and best practice among Ethereum smart contracts development.

In addition, not including any more added code or configuration to our smart contract, leaves the door open to a huge range of possible applications to benefit from our solution and make use of it. What is more, maintaining it simple will resilt in cheaper transactions costs. Due to the importance of the ERC721, we present the full interface in Listing 5.1.

Listing 5.1: "EIP 721: ERC-721 Non-Fungible Token Standard "

```
1  pragma solidity \^0.4.20;
2
3  /// @title ERC-721 Non-Fungible Token Standard
4  /// @dev See https://eips.ethereum.org/EIPS/eip-721
5  ///  Note: the ERC-165 identifier for this interface is 0x80ac58cd.
6  interface ERC721 /* is ERC165 */ {
7  /// @dev This emits when ownership of any NFT changes by any
       mechanism.
8  ///  This event emits when NFTs are created (`from` == 0) and
       destroyed
9  ///  (`to` == 0). Exception: during contract creation, any number of
       NFTs
10 ///  may be created and assigned without emitting Transfer. At the
       time of
11 ///  any transfer, the approved address for that NFT (if any) is
       reset to none.
12 event Transfer(address indexed _from, address indexed _to, uint256
       indexed _tokenId);
13
14 /// @dev This emits when the approved address for an NFT is changed
       or
15 ///  reaffirmed. The zero address indicates there is no approved
```

```
    address.
16  ///  When a Transfer event emits, this also indicates that the
    approved
17  ///  address for that NFT (if any) is reset to none.
18  event Approval(address indexed _owner, address indexed _approved,
    uint256 indexed _tokenId);
19
20  /// @dev This emits when an operator is enabled or disabled for an
    owner.
21  ///  The operator can manage all NFTs of the owner.
22  event ApprovalForAll(address indexed _owner, address indexed
    _operator, bool _approved);
23
24  /// @notice Count all NFTs assigned to an owner
25  /// @dev NFTs assigned to the zero address are considered invalid,
    and this
26  ///  function throws for queries about the zero address.
27  /// @param _owner An address for whom to query the balance
28  /// @return The number of NFTs owned by `_owner`, possibly zero
29  function balanceOf(address _owner) external view returns (uint256);
30
31  /// @notice Find the owner of an NFT
32  /// @dev NFTs assigned to zero address are considered invalid, and
    queries
33  ///  about them do throw.
34  /// @param _tokenId The identifier for an NFT
35  /// @return The address of the owner of the NFT
36  function ownerOf(uint256 _tokenId) external view returns (address);
37
38  /// @notice Transfers the ownership of an NFT from one address to
    another address
39  /// @dev Throws unless `msg.sender` is the current owner, an
    authorized
40  ///  operator, or the approved address for this NFT. Throws if `_from
    ` is
41  ///  not the current owner. Throws if `_to` is the zero address.
    Throws if
42  ///  `_tokenId` is not a valid NFT. When transfer is complete, this
    function
43  ///  checks if `_to` is a smart contract (code size > 0). If so, it
    calls
```

```
44 ///  `onERC721Received` on `_to` and throws if the return value is
       not
45 ///  `bytes4(keccak256("onERC721Received(address,address,uint256,
       bytes)"))`.
46 /// @param _from The current owner of the NFT
47 /// @param _to The new owner
48 /// @param _tokenId The NFT to transfer
49 /// @param data Additional data with no specified format, sent in
       call to `_to`
50 function safeTransferFrom(address _from, address _to, uint256
       _tokenId, bytes data) external payable;
51
52 /// @notice Transfers the ownership of an NFT from one address to
       another address
53 /// @dev This works identically to the other function with an extra
       data parameter,
54 ///  except this function just sets data to "".
55 /// @param _from The current owner of the NFT
56 /// @param _to The new owner
57 /// @param _tokenId The NFT to transfer
58 function safeTransferFrom(address _from, address _to, uint256
       _tokenId) external payable;
59
60 /// @notice Transfer ownership of an NFT -- THE CALLER IS RESPONSIBLE
61 ///  TO CONFIRM THAT `_to` IS CAPABLE OF RECEIVING NFTS OR ELSE
62 ///  THEY MAY BE PERMANENTLY LOST
63 /// @dev Throws unless `msg.sender` is the current owner, an
       authorized
64 ///  operator, or the approved address for this NFT. Throws if `_from
       ` is
65 ///  not the current owner. Throws if `_to` is the zero address.
       Throws if
66 ///  `_tokenId` is not a valid NFT.
67 /// @param _from The current owner of the NFT
68 /// @param _to The new owner
69 /// @param _tokenId The NFT to transfer
70 function transferFrom(address _from, address _to, uint256 _tokenId)
       external payable;
71
72 /// @notice Change or reaffirm the approved address for an NFT
73 /// @dev The zero address indicates there is no approved address.
```

```
74 ///  Throws unless `msg.sender` is the current NFT owner, or an
       authorized
75 ///  operator of the current owner.
76 /// @param _approved The new approved NFT controller
77 /// @param _tokenId The NFT to approve
78 function approve(address _approved, uint256 _tokenId) external
       payable;
79
80 /// @notice Enable or disable approval for a third party ("operator")
        to manage
81 ///  all of `msg.sender`'s assets
82 /// @dev Emits the ApprovalForAll event. The contract MUST allow
83 ///  multiple operators per owner.
84 /// @param _operator Address to add to the set of authorized
       operators
85 /// @param _approved True if the operator is approved, false to
       revoke approval
86 function setApprovalForAll(address _operator, bool _approved)
       external;
87
88 /// @notice Get the approved address for a single NFT
89 /// @dev Throws if `_tokenId` is not a valid NFT.
90 /// @param _tokenId The NFT to find the approved address for
91 /// @return The approved address for this NFT, or the zero address if
        there is none
92 function getApproved(uint256 _tokenId) external view returns (address
       );
93
94 /// @notice Query if an address is an authorized operator for another
        address
95 /// @param _owner The address that owns the NFTs
96 /// @param _operator The address that acts on behalf of the owner
97 /// @return True if `_operator` is an approved operator for `_owner`,
        false otherwise
98 function isApprovedForAll(address _owner, address _operator) external
        view returns (bool);
99 }
100
101 interface ERC165 {
102 /// @notice Query if a contract implements an interface
103 /// @param interfaceID The interface identifier, as specified in ERC
```

```
    -165
104 /// @dev Interface identification is specified in ERC-165. This
    function
105 ///  uses less than 30,000 gas.
106 /// @return `true` if the contract implements `interfaceID` and
107 ///  `interfaceID` is not 0xffffffff, `false` otherwise
108 function supportsInterface(bytes4 interfaceID) external view returns
    (bool);
109 }
```

In our case, we will use the ERC721 implementation of OpenZeppelin, which as already explained, will provide with higher security, robustness and well-tested code. We can find this implementation in `https://docs.openzeppelin.com/contracts/2.x/api/token/erc721`.

We can see in Fig. 5.1 that we reference two more ERC721 related interfaces. Both of them, although they do not strictly form part of the ERC721 token standard, add certain functionality to your application if implemented. While a ERC721 compliant smart contract must implement both the ERC721 and ERC165 interfaces, they are not obliged to implement the other two interfaces showed in Fig. 5.1. Nevertheless, the proper Ethereum community recommends their implementation in order to provide with added value to the system. We will cover both of them now and justify their use.

5.2.1 ERC721Metadata Token Standard Extension

This extension is **optional**. It allows the smart contract to provide information about its name, symbol or details about the assets which the NFTs represent. We show in Listing 5.2 the code for this extension.

Listing 5.2: "ERC721 Metadata Extension"

```
1  /// @title ERC-721 Non-Fungible Token Standard, optional metadata
      extension
2  /// @dev See https://eips.ethereum.org/EIPS/eip-721
3  ///  Note: the ERC-165 identifier for this interface is 0x5b5e139f.
4  interface ERC721Metadata /* is ERC721 */ {
```

```
5   /// @notice A descriptive name for a collection of NFTs in this
        contract
6   function name() external view returns (string _name);
7
8   /// @notice An abbreviated name for NFTs in this contract
9   function symbol() external view returns (string _symbol);
10
11  /// @notice A distinct Uniform Resource Identifier (URI) for a
        given asset.
12  /// @dev Throws if `_tokenId` is not a valid NFT. URIs are defined
        in RFC
13  ///  3986. The URI may point to a JSON file that conforms to the "
        ERC721
14  ///  Metadata JSON Schema".
15  function tokenURI(uint256 _tokenId) external view returns (string);
16  }
```

As we can see, it introduces the idea of having a token point to a specific URI. In this URI we can specify any information we want. A common approach is to have all the information about a specific asset stored off-chain, such as images for the underline asset or other types of data, and only an URI that points to that information stored inside the blockchain. This is a method for reducing the cost of storing assets in the blockchain and add more flexibility and simplicity. This way a shop can manage their product in their own website without needing to configure or download any software for being able to interact with the blockchain, and just have a specific URI that points to an URL where that information is persisted. This is a way of integrating part of the logic off-chain and other part on-chain. With this method, we reduce to the minimum the required information needed to be store in the blockchain, which is expensive.

Of course we will benefit from this aspect, and the purpose of using this metadata is for allowing our products to be pointed by an URI that is stored on chain, while the rest is managed by the proper shops or owners. This is why only the Product‘s Management contract implements this interface. We could have also the bookings contracts to implement this interface, but we have considered it was better only the product to be referenced.

Following the standardization objective Ethereum developers have, they have de-

fined a ERC 721 Metadata JSON Schema, that provides information about the ERC-721 asset. This Json Schema can be found in eip721 official website [12].

5.2.2 ERC721Enumerable Token Standard Extension

This extension is also **optional**. What we acquire by implementing this extension, is to allow our smart contract to publish its full list of NFTs and make them discoverable. We can see the coe for this interface in Listing 5.3.

Listing 5.3: "ERC721 Metadata Extension"

```
1  /// @title ERC-721 Non-Fungible Token Standard, optional enumeration
       extension
2  /// @dev See https://eips.ethereum.org/EIPS/eip-721
3  ///  Note: the ERC-165 identifier for this interface is 0x780e9d63.
4  interface ERC721Enumerable /* is ERC721 */ {
5  /// @notice Count NFTs tracked by this contract
6  /// @return A count of valid NFTs tracked by this contract, where
       each one of
7  ///  them has an assigned and queryable owner not equal to the zero
       address
8  function totalSupply() external view returns (uint256);
9
10 /// @notice Enumerate valid NFTs
11 /// @dev Throws if `_index` >= `totalSupply()`.
12 /// @param _index A counter less than `totalSupply()`
13 /// @return The token identifier for the `_index`th NFT,
14 ///  (sort order not specified)
15 function tokenByIndex(uint256 _index) external view returns (uint256)
       ;
16
17 /// @notice Enumerate NFTs assigned to an owner
18 /// @dev Throws if `_index` >= `balanceOf(_owner)` or if
19 ///  `_owner` is the zero address, representing invalid NFTs.
20 /// @param _owner An address where we are interested in NFTs owned by
       them
21 /// @param _index A counter less than `balanceOf(_owner)`
22 /// @return The token identifier for the `_index`th NFT assigned to `
       _owner`,
```

```
23 ///   (sort order not specified)
24 function tokenOfOwnerByIndex(address _owner, uint256 _index) external
       view returns (uint256);
25 }
```

As we see in Fig. 5.1, the Bookings smart contracts will be the ones that implement this interface. We will use the concept of publishing the existing NFTs as a way of checking availability for certain periods of time. If we take into account that each NFT represent rental rights of the product in a specific block of time, we will have that the existing NFTs are the existing block of times occupied or already rented. This is because for each booking, we will mint the associated tokens for the period of time of the renting, but if it the booking is cancelled, they will be burn. this way, we will have that the existing NFTs are going to represent the specific days, hours or weeks that are already rented.

This way, if we want to check the availability for a specific period of time, as this interface makes all NFTs discoverable, we see if the NFTs or tokens associated to the desired block of times exist. If they do not exist, then that period of time is available. This design allows us to still maintain the characteristics and benefits an ERC721 standard gives us while also reducing the costs and complexity of the logic to check availability.

5.3 KibiwooProducts Smart Contract

We present in Listing 5.4 all the KibiwooProducts' smart contract but without including the implementation of the functions to make it lighter.

Listing 5.4: "Kibiwoo Products Interface"

```
1 pragma solidity ^0.5.0;
2
3 import 'openzeppelin-solidity/contracts/ownership/Ownable.sol';
4 import 'openzeppelin-solidity/contracts/math/SafeMath.sol';
5 import 'openzeppelin-solidity/contracts/drafts/Counters.sol';
6 import 'openzeppelin-solidity/contracts/token/ERC721/ERC721Metadata.
      sol';
7 import './BookingContract.sol';
```

```
8
9  /// @author Alvaro Gericke
10 /// @title A contract for managing Kibiwoo's products registration.
11 contract Kibiwoo is Ownable, ERC721Metadata {
12
13   using SafeMath for uint256;
14   using Counters for Counters.Counter;
15
16   /// @notice Product object definition.
17   /// @param sku - The productId that identifies uniquely each
       product.
18   /// @param category - Product category.
19   /// @param min_rent_time - Integer to set the minimum rental time
       for the product.
20   /// @param name - The name of the product set by the owner of the
       product.
21   struct Product {}
22
23   /// @notice Complement object definition
24   /// @param productId - Identifier of the product to which this
       complement belongs.
25   /// @param subcategory - Subcategory of the complement.
26   /// @param name - Name of the complement.
27   struct Complement {}
28
29   /// Enum type for defining products categories.
30   enum Categories {Surf, Cycling, Ski}
31
32   /// Enum type for defining complements subcategories
33   enum Subcategories {Wetsuit, Helmet, Boots}
34
35   address payable private kibiwooAdmin;
36
37   /// The number of digits that the identifier of the product will
       have.
38   ///     The first 16 digits correspond to the shop identifier.
39   ///     The last 20 digits correspond to the product identifier.
40   uint256 constant productIdDigits = 36;
41   uint256 productIdModulus = 10 ** productIdDigits;
42
43   /// Dynamic arrays to persist stores, products and complements.
```

```
44  Product[] public products;
45  Complement[] public complements;
46
47  /// Mapping from tokenId to booking contract
48  mapping(uint256 => address payable) private _tokenToContractAddress
        ;
49  /// Mapping from complement to product.
50  mapping (uint256 => uint256)  private _complementToToken;
51  /// Mapping from productID to complements count.
52  mapping (uint256 => Counters.Counter) private _tokenComplementCount
        ;
53
54  event NewProduct();
55  event NewComplement(uint256 tokenId, uint256 complementId, uint256
        subcategory, string name);
56
57  constructor (string memory name, string memory symbol)
        ERC721Metadata(name, symbol) public {
58    kibiwooAdmin = msg.sender;
59  }
60
61  ///@notice special function for allowing the Samrt contract receive
        ether in case any of the
62  ///         existing functions is called
63  function() external payable {
64  }
65
66  /// @notice Withdraws all the balance in the contract back to owner
        of the contract
67  function withdraw() external onlyOwner {}
68
69  /// @notice Returns all the products with its complements that are
        owned by a specific shop.
70  /// @param _owner Address of the shop to retrieve the products from
        it.
71  /// @return An array with all the IDs of the products and their
        complements owned by that shop
72  function getProductsByShop (address _owner) external view returns (
        uint[] memory) {}
73
74  /// @notice Require that the owner of the product is the caller of
```

```
    the function.
75  modifier onlyOwnerOf (uint _productId) {}
76
77  /// @notice Creates a new product with id the consecutive one and
        name specified by caller.
78  /// @dev If no name is assigned, it will be assigned an empty
        string. This function sets
79  ///  isComplement to false.
80  /// @param _name String identifying the  name of the product.
81  /// @param _category An integer that represents the category of the
         product.
82  /// @return The id that uniquely identifies the registered product.
83  function createNewProduct(string memory _name, uint256 _category)
        public returns(uint256) {}
84
85  /// @notice Function for handling the booking of a product.
86  /// @param _productId Id of the product which we want to book.
87  /// @param _startTimeBlock start time of the booking.
88  /// @param _stopTimeBlock stop time of the booking.
89  function book(uint256 _productId, uint256 _startTimeBlock, uint256
        _stopTimeBlock) public {}
90
91  /// @notice Function for cancelling a specific booking of a product
        .
92  /// @param _productId Id of the product which we want to book.
93  /// @param _startTimeBlock start time of the booking.
94  function cancel(uint256 _productId, uint256 _startTimeBlock) public
         {}
95
96  /// @notice Adds a new complement to the actual product.
97  /// @dev If no name is assigned, it will be assigned an empty
        string.
98  /// @param _productId The unique identifier of the product to which
         a complement will be added.
99  /// @param _name String identifying the  name of the complement.
100 function addComplement(uint256 _productId, uint256 _subcategory,
        string memory _name) public onlyOwnerOf(_productId) returns (
        uint256) {}
101
102 /// @notice Gets the actual kibiwoo Administrator.
103 /// @return address representing Kibiwoo's Administrator address.
```

```
104   function getAdmin() public view returns(address) {}
105
106   /// @notice Get the contract booking address for a specific token.
107   /// @return uint256 address of the contract.
108   function getContractBookingAddress(uint256 tokenId) public view
          returns(address payable) {}
109
110   /// @notice Gets the total amount of products created.
111   /// @return uint256 representing the amount of products created.
112   function getProductsCount() public view returns(uint256) {}
113
114   /// @notice Gets the productId for a specific complement Id.
115   /// @return uint256 productId of a complement's product.
116   function getProductOfComplement(uint256 complementId) public view
          returns(uint256) {}
117
118   /// @notice Gets the number of complements a product has.
119   /// @return uint256 number of complements of a product.
120   function getComplementsCount(uint256 tokenId) public view returns(
          uint256) {}
121
122   /// @notice Creates a new product with id the consecutive one and
          name specified by caller.
123   /// @dev If no name is assigned, it will be assigned an empty
          string.
124   /// @param _name String identifying the  name of the product.
125   /// @param _sku The 36 digits identifier of the product to be
          created.
126   /// @param _category An integer that represents the category of the
           product.
127   /// @return The id that uniquely identifies the registered product.
128   function _registerProduct(string memory _name, uint256 _sku,
          uint256 _category) internal returns (uint256) {}
129
130   /// @notice Creates a new complement with id the consecutive one
          and name specified by caller.
131   /// @dev If no name is assigned, it will be assigned an empty
          string.
132   /// @param _productId Id of the product which the complement is
          added to.
133   /// @param _subcategory An integer that represents the category of
```

```
        the product.
134     /// @param _name String identifying the  name of the product.
135     /// @return The id that uniquely identifies the registered product.
136     function _registerComplement(uint256 _productId, uint256
            _subcategory, string memory _name) internal returns (uint256) {}
137
138     /// @notice Generates Random sku as an identifier for the product.
139     /// @param _name String identifying the  name of the product.
140     /// @return The last `productIdModulus`of the generated random
            number
141     function _generateRandomSku (string memory _name) internal view
            returns (uint256) {}
142 }
```

As we outlined before, we can see how these smart contracts inherits from ERC721Metadata which inherits from ERC721. Also we can observe how we import openZeppelin implementations for those contracts. By implementing these standards, we transform our products in ERC721 tokens which inherit all the functionality.

WE have defined also several complex struct types. Specifically, we have defined a Product struct type and a Complement struct type. The idea is that these structs will not be necessary in the future, as we have already explained that we will limit to store by each token only an URI that points to an URL from where we can fetch the data of the specific product. Nevertheless, for being able to implement a Dapp for testing our implementation, we decided to define these structures to store products in the Blockchain.

One of the most important data for a product is the minimum rental time. This variable will define the duration of each block time for each reservation token for each product, and will be used in the moment of creation of the product to initialize the Booking smart contract related to that product, so that each timeblock duration is correctly determined and set during the booking contract creation. With this design, our system gains in generalization, as it can support several types of products that require different block durations. This is reached thanks to the idea of having each product its own Booking contract. Although it may be a little more expensive, the trade-off between cost and functionality in this case was clear.

We have also decided to contain nearly all the logic inside this contract address,

so that the booking contract address is the lightest and consequently we can be more efficient in terms of cost. As we will see, the booking contract address practically only has three main functions, the reserve function, the cancel function and the checkAvailability function. Even in the main smart contract, some logic related to those three functions is done so that we reduce workload from those functions in the booking smart contracts.

It is also really important the contract mapping variable that matches product ids to contract address. This way, by a simple lookup to this mapping we can obtain the booking contract address for any token or product. The products smart contract will need to instantiate a booking smart contract every time a booking needs to be registered or a cancellation is done. Also, during the creation of a new product, a new booking contract is created and associated to that specific product. The booking contract address is also output in the form of an event, which are registered in the Blockchain and can be searched and filtered. This is another extreme method of reducing cost in our Smart Contract, as we could avoid storing contract addresses of Booking contracts, and just filtering all the new product events that the contract has emitted, and search for the one that matches the specific product Id. All this process would be done off-line, although some reads will be done on the blockchain. In our case, we prefer to have a mapping variable as it is faster and it allows us to implement easier a front-end that is sync with the contract's data and state.

We now present some UML diagrams to explain the logic of the main actions that may be done with these smart contracts.

5.3.1 Create New Product Use Case

This use case explains the process of registering a new product. We have set-up a private blockchain through Truffle and ganache-cli that provides us with 10 pre-funded accounts with 100 Ether each. We use one of these accounts to send a transaction to the private blockchain in which we specify that we want to call the createNewProduct method and send the function parameters within the transaction. We do this through the truffle console which acts as the gateway to the private blockchain.

This transaction is sent to Kibiwoo's smart contract, which calls the necessary internal functions to create the product and update its variables. Once it has the

productId, it creates a new BookingContract with the specific parameters of that product. When this contract is finally deployed to a blockchain address, this address is stored in Kibiwoo's smart contract and associated to the productId. This way we enable future interactions with that smart contract. Finally a newProduct event is triggered and stored in the blockchain with the initial transaction receipt. The client's node, which is listening to specific events triggered by Kibiwoo's smart contract, views the transaction receipt and can examine it to view the its results.

This is how the process works. It is important to notice that with each product creation transaction, a new BookingContract is created and link to the new product id. Of course, all this logic also checks that the product still does not exist and many other requirement. We can see the diagram for this process in Fig. 5.2.

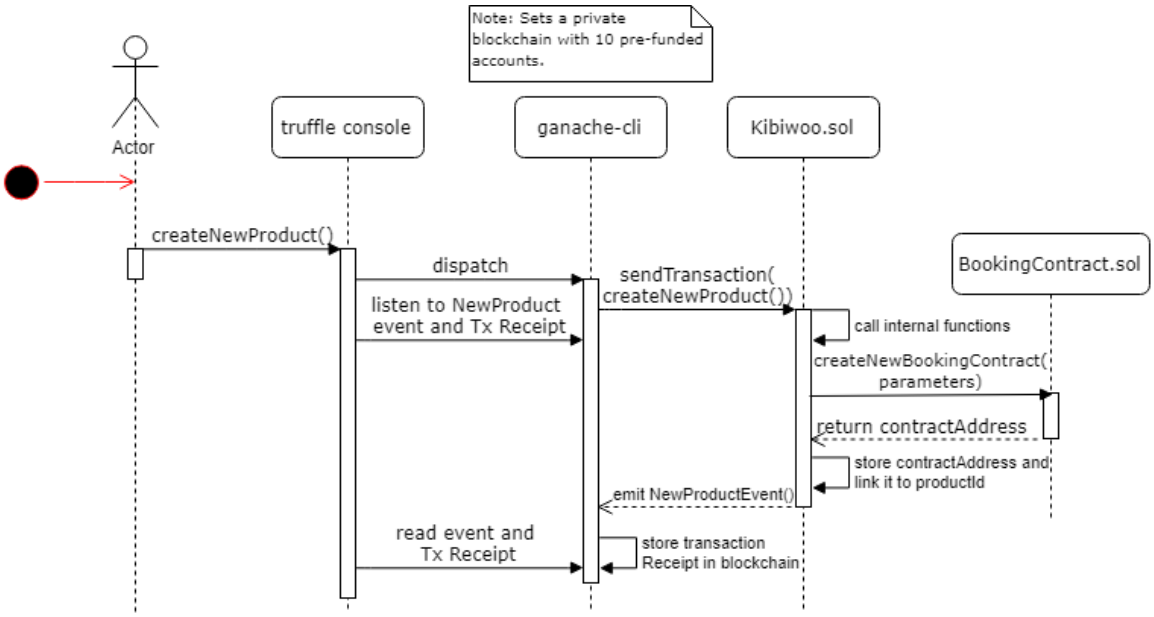


Figure 5.2: Create New Product Use Case.

5.3.2 Book Product Use Case.

For this use case, we initially call Kibiwoo's smart contract, but this contract after some checks relegates the logic of the booking to the BookingContract. In order to achieve this, we instance a BookingContract by passing the address of the smart contract that was created during the creating of the product we want to book. Once it has been initialized, we can call any functions inside the BookingContract. The

method for checking the availability of the product being booked will be explain in detail in section 5.4.

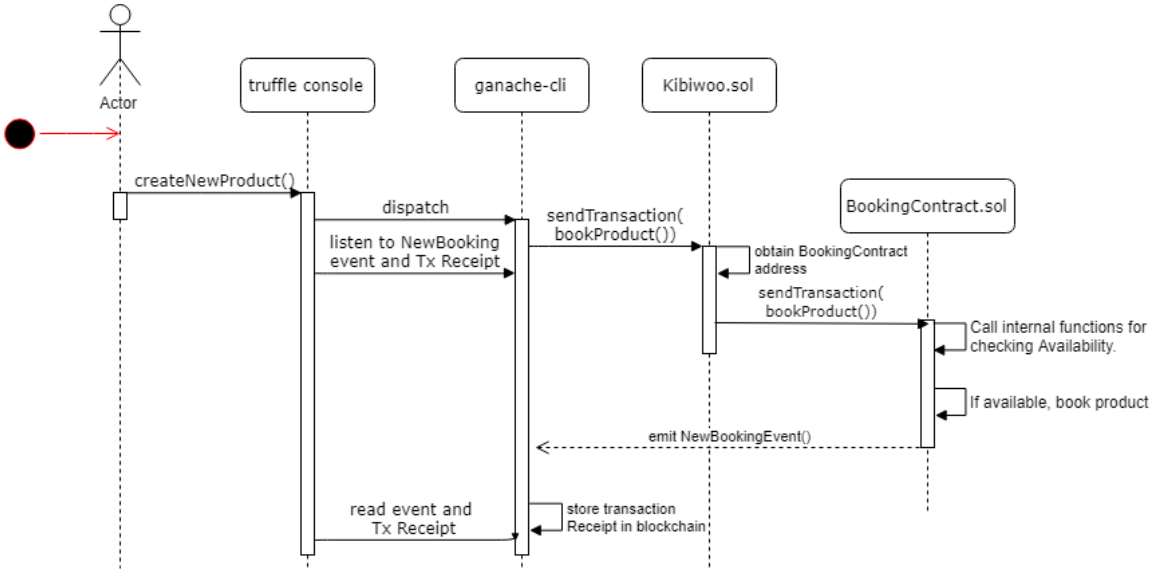


Figure 5.3: Book Product Use Case.

The logic for cancelling a booking is really similar to the one for booking it.

5.4 BookingContract Smart Contract

We have designed this contract with the idea of making it as simple as possible so that the core logic relies on Kibiwoo's smart contract. This contracts include only the strictly necessary functions and code o handle each products scheduling of reservations. As we have already explained, it inherits the ERC721 Enumerable interface, which provides the token reservations with exchangeable and other capabilities. In addition, the Enumerable part makes it easy to discover already booked tokens and time slots as well as methods for associating those token reservations to addresses. We present in Listing 5.5 the code for these contracts.

Listing 5.5: "BookingContract.sol"

```
1 pragma solidity ^0.5.0;
2
3 import 'openzeppelin-solidity/contracts/math/SafeMath.sol';
```

```
4  import 'openzeppelin-solidity/contracts/drafts/Counters.sol';
5  import 'openzeppelin-solidity/contracts/token/ERC721/ERC721Enumerable
       .sol';
6  import 'solidity-treemap/contracts/TreeMap.sol';
7
8  /// @author Alvaro Gericke
9  /// @title A contract for managing products  bookings.
10 contract BookingContract is ERC721Enumerable {
11
12   using SafeMath for uint256;
13   using Counters for Counters.Counter;
14   using TreeMap for TreeMap.Map;
15
16   // Constant that representss the Maximum duration allowed.
17   // This is implemented for security reasons
18   uint256 constant public RESERVATION_DURATION_LIMIT = 60 days;
19   /// tokenId this contract refers to in the Products Management
         contract.
20   uint256 private _tokenIdReference;
21   /// Constant that represents the amount of time each block
         represents.
22   uint256 private _blockTime;
23   // TreeMap that stores all reservations.
24   TreeMap.Map public timeBlocksMap;
25   /// Mapping from reservationId to startTimeStamp
26   mapping(uint256 => uint256) public startTimestamps;
27   /// Mapping from reservationId to endTimeStamp
28   mapping(uint256 => uint256) public stopTimestamps;
29   /// Variable to store the id for the next token
30   uint256 nextTokenId;
31
32   event NewBooking(
33     address indexed booker,
34     uint256 indexed bookingId,
35     uint256 startTimeBlock,
36     uint256 stopTimeBlock
37   );
38   event CancelBooking(
39     address indexed booker,
40     uint256 indexed bookingId
41   );
```

```
42
43  constructor (uint256 tokenIdRef, uint256 min_rent_time) public {
44    _tokenIdReference = tokenIdRef;
45    _blockTime = min_rent_time;
46  }
47
48  ///@notice special function for allowing the Samrt contract receive
        ether in case any of the
49  ///         existing functions is called
50  function() external payable {
51
52  }
53
54  /// @notice Reserve the period between time `_start` to time `_stop
        `
55  /// @dev A successful booking must ensure each time slot in the
       range _start to _stop
56  ///  is not previously booked.
57  /// @param _booker The Ethereum address of the person that wants to
        book.
58  /// @param _start startTimeBlock.
59  /// @param _stop stopTimeBlock
60  /// @return The token Id associated to this reservation.
61  function book(address _booker, uint256 _start, uint256 _stop)
62  public
63  returns(uint256)
64  {
65    // First ensure all timeblocks between `_start`and `_stop` are
        available.
66    if (!checkAvailability(_start, _stop)) {
67      revert("BookingContract: Time blocks are unavailable.");
68    }
69
70    // Obtain the Id that will be associated to this booking.
71    uint256 tokenId = nextTokenId;
72    nextTokenId = nextTokenId.add(1);
73
74    // Create the token of the reservation.
75    _mint(_booker, tokenId);
76
77    // Store start timeblock and stop timeblock for this booking
```

```
        tokenId.
78      startTimestamps[tokenId] = _start;
79      stopTimestamps[tokenId] = _stop;
80
81      timeBlocksMap.put(_start, tokenId);
82
83      emit NewBooking(_booker, tokenId, _start, _stop);
84
85      return tokenId;
86    }
87
88    /// @notice Cancel an existing booking. Only the owner can do it.
89    /// @param _bookingId Booking Identifier inside this contract.
90    /// @param origin equals to the msg.sender value of the origin
          transaction.
91    function cancelBooking(address origin, uint256 _bookingId)
92    public
93    {
94      require(_exists(_bookingId), "BookingContract: Booking does not
          exist");
95
96      uint256 startTime = startTimestamps[_bookingId];
97
98      _burn(origin, _bookingId);
99
100     delete startTimestamps[_bookingId];
101     delete stopTimestamps[_bookingId];
102     timeBlocksMap.remove(startTime);
103
104     emit CancelBooking(origin, _bookingId);
105   }
106
107   /// @notice Gets the tokenId it refers to in the product management
          contract.
108   /// @return uint256 representing product's Id this token refers to.
109   function getTokenIdRef() public view returns(uint256) {
110   return _tokenIdReference;
111   }
112
113   /// @notice Gets the number of seconds each block of time
          represents.
```

```
114   /// @return uint256 that represents the amount of time in seconds
          of each block time.
115   function getBlockTime() public view returns(uint256) {
116     return _blockTime;
117   }
118
119   /// @notice Get the reservationId for a specific startTime
120   /// @param _startTimeBlock The start timeblock save as a key in the
          TreeMap
121   /// @return uint256 representating the reservationId associated to
          that startTime.
122   function getReservationIdFromStartTimeBlock(uint256 _startTimeBlock
          )
123   public
124   view
125   returns(uint256)
126     {
127     bool found;
128     uint256 reservationId;
129
130     (found, reservationId) = timeBlocksMap.get(_startTimeBlock);
131
132     if (!found) {
133       revert("BookingContract: No Reservation Id for that
              startTimeBlock.");
134     }
135
136     return reservationId;
137   }
138
139   /// @notice Check if timeslots between startTimeBlock and
          stopTimeBlock are available.
140   /// @param _startTimeBlock The start time block for the reservation
          .
141   /// @param _stopTimeBlock The end time block for the reservation.
142   /// @return bool indicating if the token Id is available or not.
143   function checkAvailability(uint256 _startTimeBlock, uint256
          _stopTimeBlock)
144   public
145   view
146   returns(bool)
```

```
147   {
148
149     require(_stopTimeBlock > _startTimeBlock, "BookingContract:
          StopTimeBlock must end after startTimeBlock.");
150     require((_stopTimeBlock - _startTimeBlock) <= uint256(
          RESERVATION_DURATION_LIMIT), "BookingContract: Reservation
          duration must not exceed limit");
151
152     bool found;
153     uint256 reservationId;
154     uint256 startTime;
155
156     // find closest event that started after _start
157     (found, startTime, reservationId) = timeBlocksMap.ceilingEntry(
          _startTimeBlock);
158     if (found && _stopTimeBlock > startTime) {
159       return false;
160     }
161
162     // find closest event that started before _start
163     (found, startTime, reservationId) = timeBlocksMap.floorEntry(
          _startTimeBlock);
164     if (found) {
165       if (stopTimestamps[reservationId] > _startTimeBlock) {
166         return false;
167       }
168     }
169     return true;
170   }
171 }
```

We outline the use of TreeMap. This approach has been done in order to be more efficient in terms of required storage and cost. We selected to store only the startTimeStamps and soptTimeStamps for each booking in a treemap. This is a Red-Black tree based Navigable Order Static TreeMap in Solidity [1]. The treemap enables storage of (uint -¿ uint) mapping that further provides a total ordering by keys and navigation functions returning closest matches for any given keys.

The initial design purpose was to tokenize every time block, and for every booking

[1] https://github.com/saurfang/solidity-treemap

mint a reservation token for each of the time blocks that contained the booking. This approach was expensive although practical. By implementing a treemap, we considerably reduce the required tokens to be minted. Moreover, with the initial approach, although a token was minted for each time block, we did not relate them as a booking as a whole, as those tokens could be traded independently.

By using treemaps, we store only the start bock and the end time block for a specific reservation. As each BookingContract is unique for each product, there exists no possibility of having two tokens that refer to the same time block. This way we store the start time block, as the keys in the treemap and the values for that keys are the reservation ids. We see then how we can pack and link each booking with its start date and end date in the same token. This is because in addition to link start time blocks to reservation ids in the treemap, we also have two mappings, one for start time stamps and the other for stop time stamps, in which the keys are the reservation ids and the values the pertinent start or stop time.

Further, it is a much more efficient approach to check also availability of time blocks, as reads to the treemap have a complexity of *O(log n)*. We present in Listing 5.6 all the cases that have been checked to test the functionality of the checkAvailability function and the results of running them can be seen in Fig. 5.4 to show how we have extensively test all of our functions, specially those that are critical such as the checkAvailability function.

Listing 5.6: "Test cases for checkAvailability function"

```
describe('#Test CheckAvailability and Book function', function () {
  it("Reverts if stopTime is less than startTime.", async function()
      {
    await expectRevert(
      this.bookingContractInstance.checkAvailability(10000, 5000),
    ' BookingContract: StopTimeBlock must end after startTimeBlock.'
    );
  });

  it("Reverts if total Time exceeds Reservation Maximum Limit time.",
      async function() {
    let start = 1000;
    let stop = 1000+61*24*3600;
```

```
    await expectRevert(
      this.bookingContractInstance.checkAvailability(start, stop),
      'BookingContract: Reservation duration must not exceed limit'
    );
  });

  context('No revert cases', function () {
    it("Returns true if no reservations already made", async function
        () {
      let start = 1000;
      let stop = 1000+11*24*3600;
      expect(await this.bookingContractInstance.checkAvailability(
          start, stop)).to.be.true;
    });

    context('Cases where should return false due to overlap on
        intended bookings', function() {
      beforeEach('Create an initial booking', async function() {
        this.start1 = 1000+5*24*3600;
        this.stop1 = 1000+20*24*3600;
        this.start2 = 1000+30*24*3600;
        this.stop2 = 1000+35*24*3600;
        this.bookingId1 = await
        this.bookingContractInstance.book(customer1, this.start1,
            this.stop1, {from: customer1});
        this.bookingId2 = await
        this.bookingContractInstance.book(customer1, this.start2,
            this.stop2, {from: customer1});
      });

      // block time range >> reservation 1 and 2
      //  |--------------------|
      //     |--1--|    |--2--|
      it("Returns false if already reserved. Case 1.", async function
          () {
        let startTime = 1000+2*24*3600;
        let stopTime = 1000+36*24*3600;
        let available = await
        this.bookingContractInstance.checkAvailability(startTime,
            stopTime);
        expect(available).to.be.false;
```

```
        });

        //      // block time range >> reservation 1 but not 2
        //  |----------|
        //      |--1--|    |--2--|
        it("Returns false if already reserved. Case 2.", async function
            () {
          let startTime = 1000+2*24*3600;
          let stopTime = 1000+15*24*3600;
          let available = await
          this.bookingContractInstance.checkAvailability(startTime,
              stopTime);
          expect(available).to.be.false;
        });

        // block time range >> reservation 2 but not 1
        //               |----------|
        //      |--1--|    |--2--|
        it("Returns false if already reserved. Case 3.", async function
            () {
          let startTime = 1000+28*24*3600;
          let stopTime = 1000+36*24*3600;
          let available = await
          this.bookingContractInstance.checkAvailability(startTime,
              stopTime);
          expect(available).to.be.false;
        });

        // block time range starts before reservation 1 and ends during
            1
        //  |----|
        //      |--1--|    |--2--|
        it("Returns false if already reserved. Case 4.", async function
            () {
          let startTime = 1000+2*24*3600;
          let stopTime = 1000+10*24*3600;
          let available = await
          this.bookingContractInstance.checkAvailability(startTime,
              stopTime);
          expect(available).to.be.false;
        });
```

```
    // block time range starts within reservation 1
    //       |--|
    //     |--1--|    |--2--|
    it("Returns false if already reserved. Case 5.", async function
        () {
      let startTime = 1000+7*24*3600;
      let stopTime = 1000+10*24*3600;
      let available = await
      this.bookingContractInstance.checkAvailability(startTime,
          stopTime);
      expect(available).to.be.false;
    });

    // block time range starts during reservation 1 and ends before
        2
    //       |----|
    //     |--1--|    |--2--|
    it("Returns false if already reserved. Case 6.", async function
        () {
      let startTime = 1000+7*24*3600;
      let stopTime = 1000+18*24*3600;
      let available = await
      this.bookingContractInstance.checkAvailability(startTime,
          stopTime);
      expect(available).to.be.false;
    });

    // block time range starts after reservation 1 and ends during
        2
    //             |---|
    //     |--1--|    |--2--|
    it("Returns false if already reserved. Case 7.", async function
        () {
      let startTime = 1000+22*24*3600;
      let stopTime = 1000+32*24*3600;
      let available = await
      this.bookingContractInstance.checkAvailability(startTime,
          stopTime);
      expect(available).to.be.false;
    });
```

```
    // block time range within reservation 2
    //                  |--|
    //     |--1--|    |--2--|
    it("Returns false if already reserved. Case 8.", async function
        () {
      let startTime = 1000+32*24*3600;
      let stopTime = 1000+34*24*3600;
      let available = await
      this.bookingContractInstance.checkAvailability(startTime,
          stopTime);
      expect(available).to.be.false;
    });


    // block time range starts during reservation 2
    //                     |---|
    //     |--1--|    |--2--|
    it("Returns false if already reserved. Case 9.", async function
        () {
      let startTime = 1000+32*24*3600;
      let stopTime = 1000+37*24*3600;
      let available = await
      this.bookingContractInstance.checkAvailability(startTime,
          stopTime);
      expect(available).to.be.false;
    });


    // available time range before reservation 1
    // |--|
    //     |--1--|    |--2--|
    it("Returns false if already reserved. Case 10.", async
        function() {
      let startTime = 1000+1*24*3600;
      let stopTime = 1000+3*24*3600;
      let available = await
      this.bookingContractInstance.checkAvailability(startTime,
          stopTime);
      expect(available).to.be.true;
    });
```

```
      // available time range between reservation 1 and 2
      //             |--|
      //     |--1--|     |--2--|
      it("Returns false if already reserved. Case 11.", async
          function() {
        let startTime = 1000+22*24*3600;
        let stopTime = 1000+28*24*3600;
        let available = await
        this.bookingContractInstance.checkAvailability(startTime,
            stopTime);
        expect(available).to.be.true;
      });

      // available time range after 2
      //                          |--|
      //     |--1--|     |--2--|
      it("Returns false if already reserved. Case 12.", async
          function() {
        let startTime = 1000+37*24*3600;
        let stopTime = 1000+45*24*3600;
        let available = await
        this.bookingContractInstance.checkAvailability(startTime,
            stopTime);
        expect(available).to.be.true;
      });
    });
  });
});
```

```
Contract: BookingContract
  #Initial checks regarding initial set-up and general Smart contract configuration
    ✓ Check tokenIdRef is correct. (42ms)
    ✓ Check block time is correct. (77ms)
    ✓ Check correct value of Maximum Reservation constant (48ms)
    ✓ Check Smart Contract can receive ether. (79ms)
  #Test CheckAvailability and Book function
    ✓ Reverts if stopTime is less than startTime. (55ms)
    ✓ Reverts if total Time exceeds Reservation Maximum Limit time. (39ms)
    No revert cases
      ✓ Returns true if no reservations already made (64ms)
      Cases where should return false due to overlap on intended bookings
        ✓ Returns false if already reserved. Case 1. (43ms)
        ✓ Returns false if already reserved. Case 2. (42ms)
        ✓ Returns false if already reserved. Case 3. (41ms)
        ✓ Returns false if already reserved. Case 4.
        ✓ Returns false if already reserved. Case 5. (103ms)
        ✓ Returns false if already reserved. Case 6.
        ✓ Returns false if already reserved. Case 7.
        ✓ Returns false if already reserved. Case 8. (42ms)
        ✓ Returns false if already reserved. Case 9.
        ✓ Returns false if already reserved. Case 10.
        ✓ Returns false if already reserved. Case 11. (58ms)
        ✓ Returns false if already reserved. Case 12.
  #Test Book function
    ✓ Reverts if stopTime is less than startTime. (65ms)
    ✓ Reverts if total Time exceeds Reservation Maximum Limit time. (59ms)
    With correct values
      ✓ Check correct owner of bookings. (39ms)
      ✓ Check correct values for startTimeStamp and stopTimestamp (147ms)
      ✓ Booking a product emits NewBooking event with correct values.
      ✓ Reverts if trying to book an already booked timeslot (87ms)
  #Test CancellBooking function
    ✓ Reverts if trying to cancel a non-existent booking. (52ms)
    With correct values
      ✓ Reverts if trying to cancel when not owning that booking. (57ms)
      ✓ Makes cancelled time period available again. Check events values. (269ms)
  #Test getReservationIdFromStartTimeBlock function
    ✓ Reverts if querying a non-existent starttimeBlock.
    ✓ Returns correct reservationId value. (51ms)
```

Figure 5.4: checkAvailability function test results.

5.5 Environment Configuration

We finally introduce in this chapter the configured and set-up of the development environment for being able to code, compile, migrate, test and interact with the smart contracts. We can see in Fig. 5.5 an overview of the architecture.

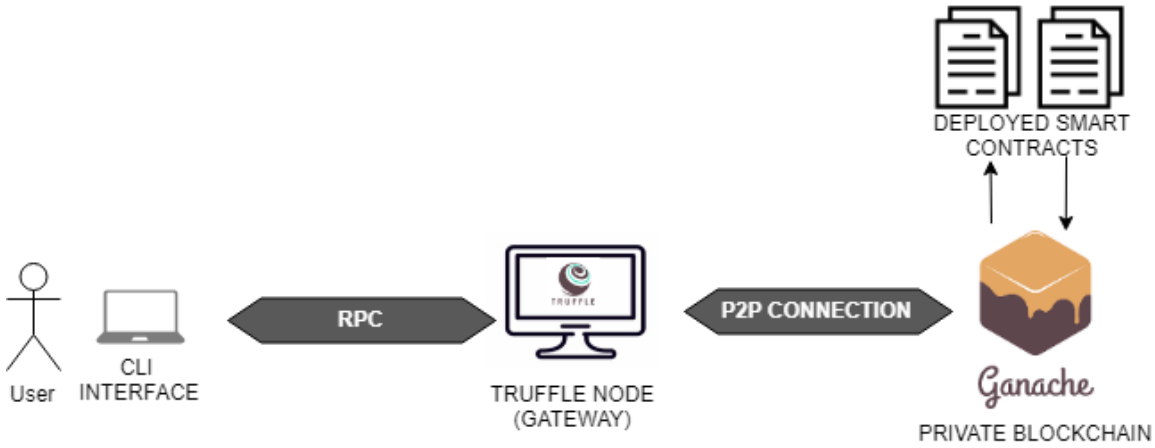


Figure 5.5: Development Environment Architecture.

Truffle console acts as a node to the private blockchain deployed by ganache-cli. Truffle provides us with tools for compiling, deploying and testing our smart contracts, while ganache-cli or Ganache UI sets up a private blockchain to deploy our contracts to.

CHAPTER 6

Decentralized Application

In this chapter, we will cover the design and implementation of the decentralized application. We will fisrtly explain the environment set up and architecture and finally dive into the explanation of all the components of the decentralized application.

6.1 Introduction

This decentralized application is being developed as a proof of our smart contracts development concept. We will try to create an application that squeezes all the characteristics of our blockchain contracts. We will not focus in implementing a single use case, such as a booking system, as outlined through this project. Instead, we will try to implement a solution that shows the general characteristics of our smart contracts to fit several different sectors or type of products.

For doing so, we included in our smart contracts some structures and code to give support to this decentralized application. We will design a system that registers several types of products, each of them with different booking characteristics. We will proof this way how our system can manage different products with different time

slots each one.

This complex decentralized application will be deployed only to the private blockchain,as still we need to refactor our code to improve it before final deployment to a real blockchain. Nevertheless, we deployed also a simpler version of our smart contracts and our decentralized application to the Ropsten Ethereum Testnet.

This decentralized application will have some characteristics of a marketplace that works in the blockchain, simulating the real cases of platforms such as Booking or Airbnb, in order to provide a initial point for migrating these platforms to the blockchain by using our smart contracts.

6.2 Overview and Architecture

For this section, we have configured some additional tools. We have set up the MetaMask web browser extension to access our private blockchain or the proper Ethereum mainnet. We have also configured and set up the web3.js provider inside our decentralized application. In addition, we have used Drizzle for maintaining our React Dapp updated with the contract state and variables values, and finally we have used some already existent React components for creating some layouts in our Dapp such as the calendars.

We can see an overview of this general architecture in Fig. 6.1.

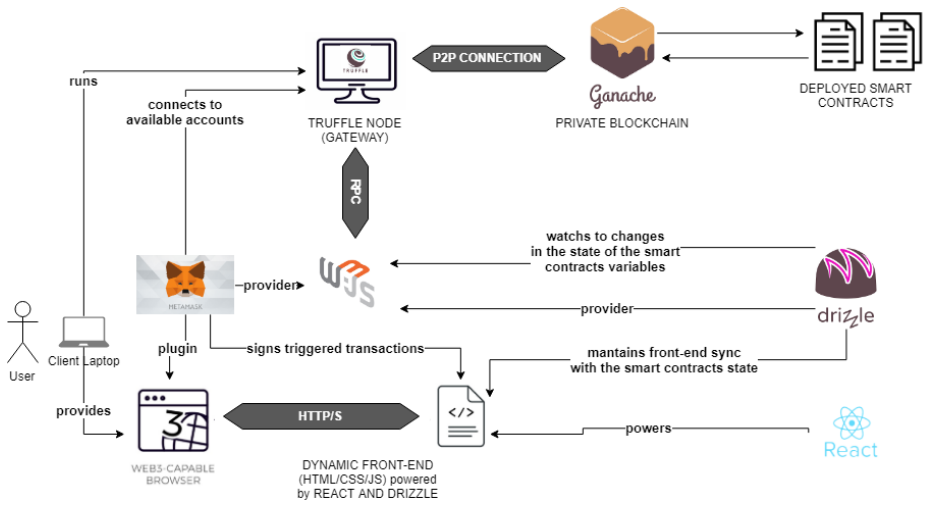


Figure 6.1: Decentralized Application Environment Architecture.

We see that our dynamic application front-end will be rendered through a web browser. That web browser will have installed the Metamask plugin. The Metamask plugin will connect to the running node and will make available its accounts through the browser, it will act as a really simple wallet. The front-end will be powered by React and Drizzle. Drizzle and Metamask will provide the web3.js object through which we will communicate with the node. Web3 exposes a human readable API for sending transactions and interacting with smart contracts. Web3.js automatically translates those human readable actions sent by the user into JSON-RPC, which is the language that the node understands and implements. These transactions are then sent to the blockchain through the node and reach the smart contract. Then the network processes and validates the transactions which cause the smart contract functions to be triggered, and the result is posted through the blockchain as transaction receipts. Then, Drizzle, which through its web3 object is listening and watching the blockchain for changes in the smart contracts, receives those transactions receipts and updates the front-end accordingly through the props and state of React components. In addition, drizzle maintains a state of specific contract variables for exposing them to the user interface.

We can see in Fig. 6.2 examples of how Metamask works. We can see that it picks the available account or accounts and exposes it/them through the browser plugin. We also can see how when a transaction is sent, it requires us to sign the transaction.

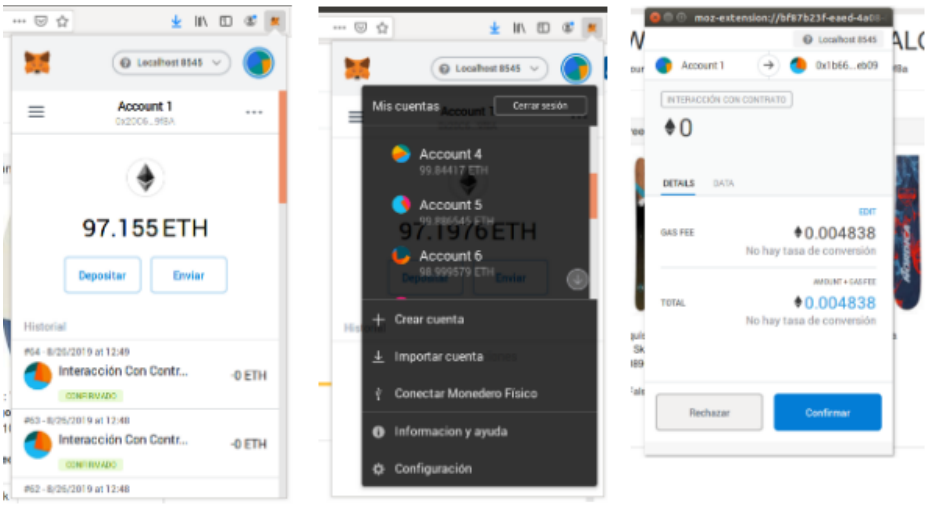


Figure 6.2: Metamask Functionality Examples.

We illustrate also in Fig. 6.3 the initialization of ganache-cli private blockchain and

the pre-funded accounts that it provides as well as the functions calls it receives from the web3 object and how transactions are mined, processed, validated and introduces into the private blockchain.

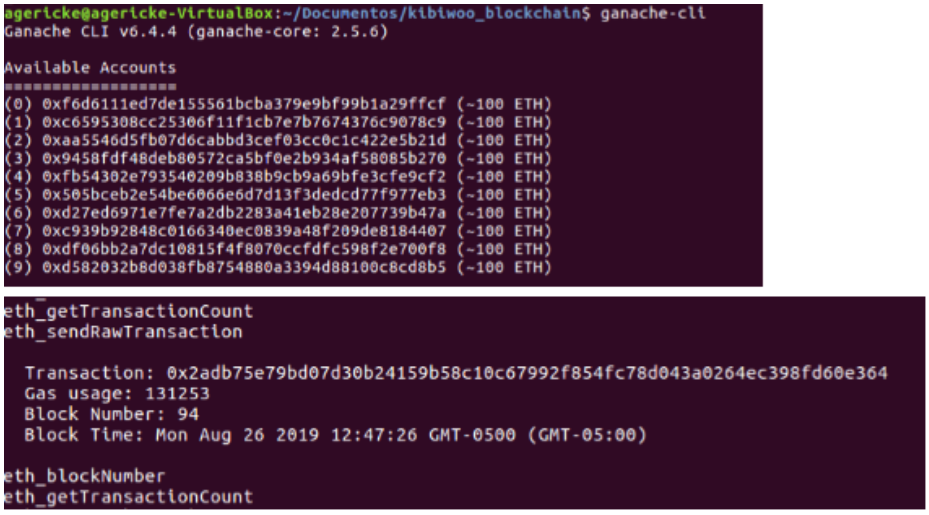


Figure 6.3: Ganache-Cli Functionality Examples.

6.3 Decentralized Applications Implementations

6.3.1 Simple Dapp Implementation

Firtly, developed a simple decentralized application for testing through a user interface the core of our Kibiwoo smart contract. Basically, with this first version we wanted to have all the environment set up, check that everything was integrated correctly and that every module provide its functionality.

This first version, consists of a single web page decentralized application which uses a JSON file as its database for storing the products. This solution is focused in testing the managing of products and checking that they were correctly registered in the blockchain. It allow us also to program a first version of a front-end that correctly synchronizes with the smart contract state variables and updates every time a transaction was sent.

Additionally, a minimum functionality for booking those registered products was included. With this basic implementation we could test that when a product was

booked, the Front-end disabled that product from being booked again. We can see in the following figures, the layout and logic of this decentralized application.

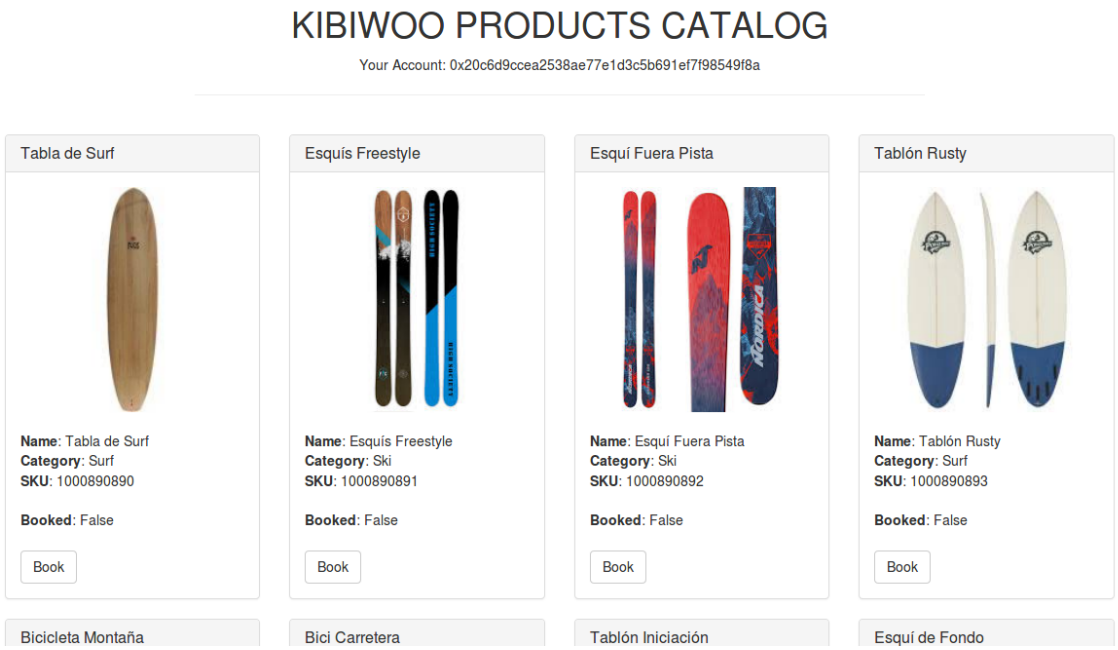


Figure 6.4: Simple Dapp Webpage.

We can see in Fig. 6.4 how we obtain the current active account from the web3 object and print it at the top of the page. We also can see how the dapp retrieves each product from the database, and as we can see in Fig. 6.5, the dapp checks by asking the smart contract in the blockchain if that product exists inside the contract, and in case it does not exist, it sends a transaction to the smart contract for creating the product. We also can appreciate how the information for each product is retrieved from the product struct we saw in Listing 5.4. Finally we show in Fig. 6.6 how when we click on book button of a product, it sends a transaction for booking it and then disables the booking of the product.

Although this may seem a simple application, we have put in practice many ideas and use cases of a dapp. We have tested corrected the connection to MetaMask and the sign of transactions through it. Retrieve and send data to a smart contract in the blockchain, update our front-end accordingly to the contracts state, and also we have been able to test a simple implementation of a booking system. So we can say that it is a simple but really complete Dapp. This was what was delivered as part of one of the sprints in order to provide a minimum viable product that works correctly

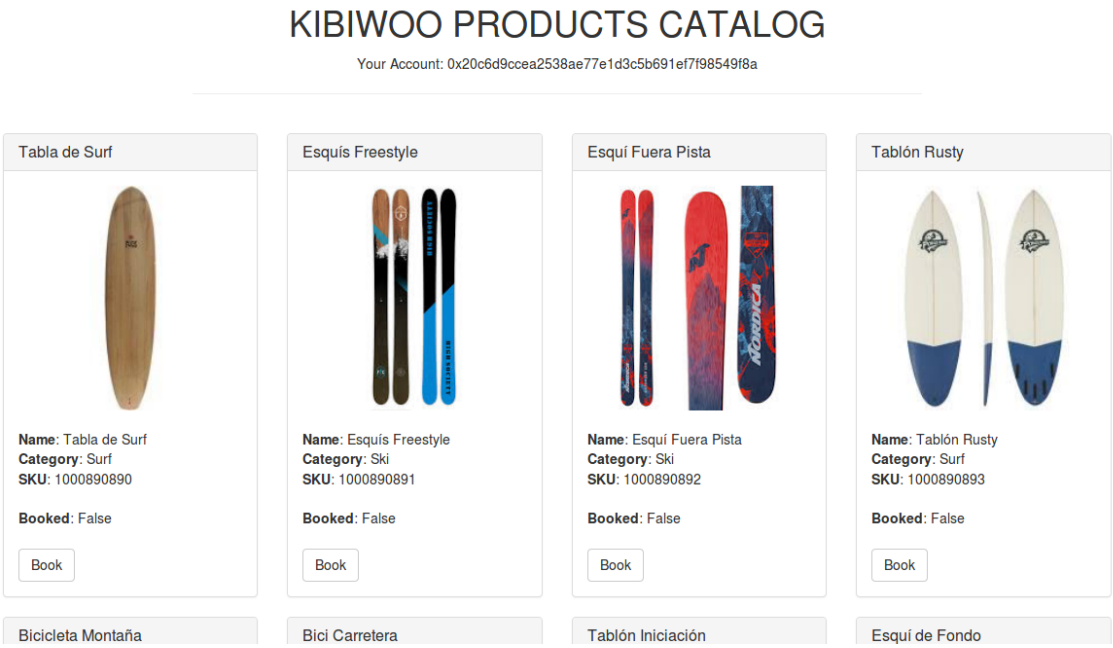


Figure 6.5: Create a New Product Transaction.

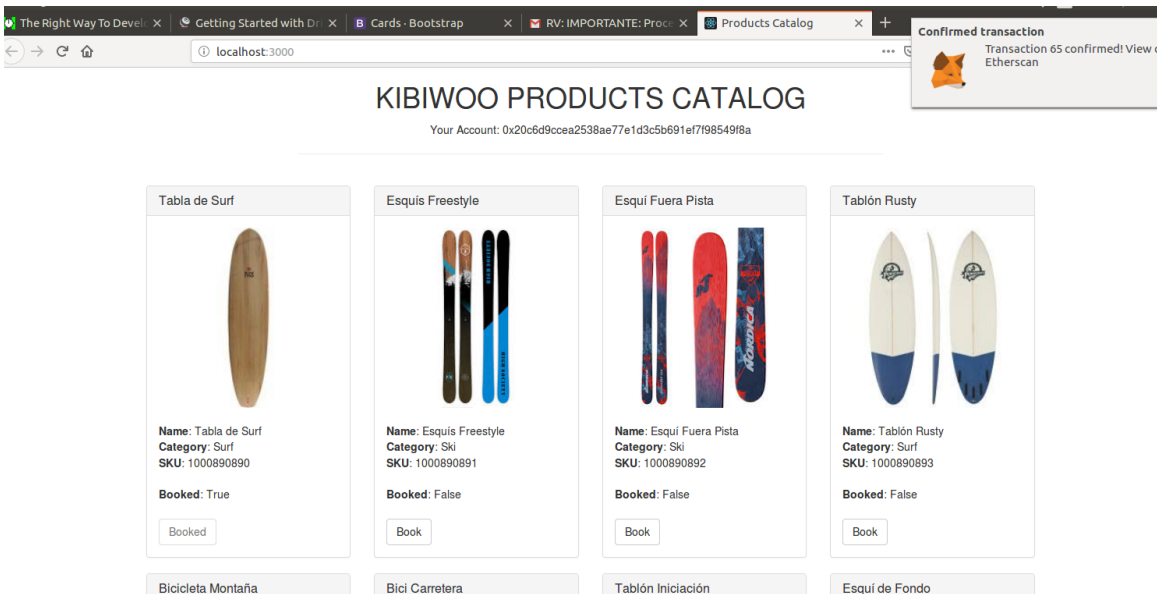


Figure 6.6: After Booking Layout.

and included all functionalities.

6.4 Full Dapp Implementation

This implementation is more complex and it includes more functionality, such as a multipage web application, drizzle implementation, and also connects with the full smart contract system. This implementation is still work in progress so it will not be deployed to the Ropsnet Testnet still.

Although it may seem more complex, it really has the same logic as the simple dapp, but the most interesting innovation is the introduction of drizzle and react for synchronizing the front-end with the smart contracts variables and data.

As we have distinct Bookingcontracts for each product, each of them will be handle by drizzle so that we can maintain a local state of each of them locally for being accessible by the front-end, and at the same time drizzle handles the updates of those states so that they are in sync with the smart contracts. We will have a landing page similar to the one shown for the simple dapp, but additionally we will have additional web pages.

One additional web page is a form for registering products in the Blockchain. An then, every product will have its own webpage where it will display the available time slots for being booked and also we will be able to see the existing booking for that calendar. For rendering nice calendars and user interfaces we have ued third-party javascript libraries.

Regarding React, we have structured the Front-End in React components for having a modular design. We have the header module, the product card module, the calendar module and the registerForm module. We can see an example of a product React Component in Listing 6.1.

Listing 6.1: "Test cases for checkAvailability function"

```
import 'bootstrap/dist/css/bootstrap.min.css';
import React from "react";

class ProductHolder extends React.Component {

  componentDidMount() {
    const { drizzle, drizzleState } = this.props;
```

```
  }

  render() {

    return (
      <div id="productTemplate">
        <div className="col-sm-6 col-md-4 col-lg-3">
          <div className="card panel-product">
            <div className="card-header">
              <h3 className="card-title">Scrappy-{this.props.name}</
                  h3>
            </div>
            <img
              className="card-img-top"
              alt="140x140"
              data-src="holder.js/140x140"
              style={{width: '100%'}}
              src="https://animalso.com/wp-content/uploads/2017/01/
                  Golden-Retriever_6.jpg"
              data-holder-rendered="true"
            />
            <div className="card-body">
              <br/><br/>
              <strong>Name</strong>: <span className="product-name">
                  PRODUCT NAME</span><br/>
              <strong>Category</strong>: <span className="product-
                  category">CAT.</span><br/>
              <strong>SKU</strong>: <span className="product-sku">SKU
                  </span><br/><br/>
              <strong>Booked</strong>: <span className="product-
                  booked">False</span><br/><br/>
              <button className="btn btn-default btn-book" type="
                  button" data-id="0">Book</button>
            </div>
          </div>
        </div>
      </div>
    );
  }
}
```

```
export default ProductHolder;
```

We show in Listing 6.2 an example of the App Component to show how each component is called and a drizzle object and state is passed, so that this object is available through the props of a React Component. Drizzle as already explained, acts as a Redux store for maintaining the state of each smart contract that is part of our system.

Listing 6.2: "Test cases for checkAvailability function"

```
import React, { Component } from 'react'
import logo from './logo.svg';
import './App.css';
//import ReadSum from './ReadSum'
import ReadNumberProducts from './ReadNumberProducts';
import ProductHolder from './Product-holder';
import ProductRegisterForm from './ProductRegisterForm';

class App extends Component {

  state = { loading: true, drizzleState: null };

  componentDidMount() {
    const { drizzle } = this.props;

    // subscribe to changes in the store
    this.unsubscribe = drizzle.store.subscribe(() => {

      // every time the store updates, grab the state from drizzle
      const drizzleState = drizzle.store.getState();

      // check to see if it's ready, if so, update local component
         state
      if (drizzleState.drizzleStatus.initialized) {
        this.setState({ loading: false, drizzleState });
      }
    });
  }
```

```
  compomentWillUnmount() {
    this.unsubscribe();
  }

  render() {
    if (this.state.loading) return "Loading Drizzle...";
    var rows = [];
    for (var i = 0; i < 10; i++) {
      // note: we add a key prop here to allow react to uniquely
          identify each
      // element in this array. see: https://reactjs.org/docs/lists-
          and-keys.html
      rows.push(
      <ProductHolder
        drizzle={this.props.drizzle}
        drizzleState={this.state.drizzleState}
        name={i}
        key={i}
      />
    );
    }
    return (
      <div className="App">
      <ReadNumberProducts
        drizzle={this.props.drizzle}
        drizzleState={this.state.drizzleState}
      />
      <ProductRegisterForm
        drizzle={this.props.drizzle}
        drizzleState={this.state.drizzleState}
      />
      <div className="products-list">
        {rows}
      </div>
      </div>
    );
  }
}

export default App;
```

Finally, we show some images of the Dapp front-end. We can see in Fig. 6.7 how the react app asks for access and connecting to the MetaMask browser extension. We can see also how while the web3 and accounts are not available, the drizzle object is not ready still so it shows *Loading Drizzle...*'. Whenever is ready, it renders the actual webpage as we can see in Fig. 6.8. We can see the Productholder react components there. Finally we show in Fig. 6.9 how the calendar for a specific products shows.

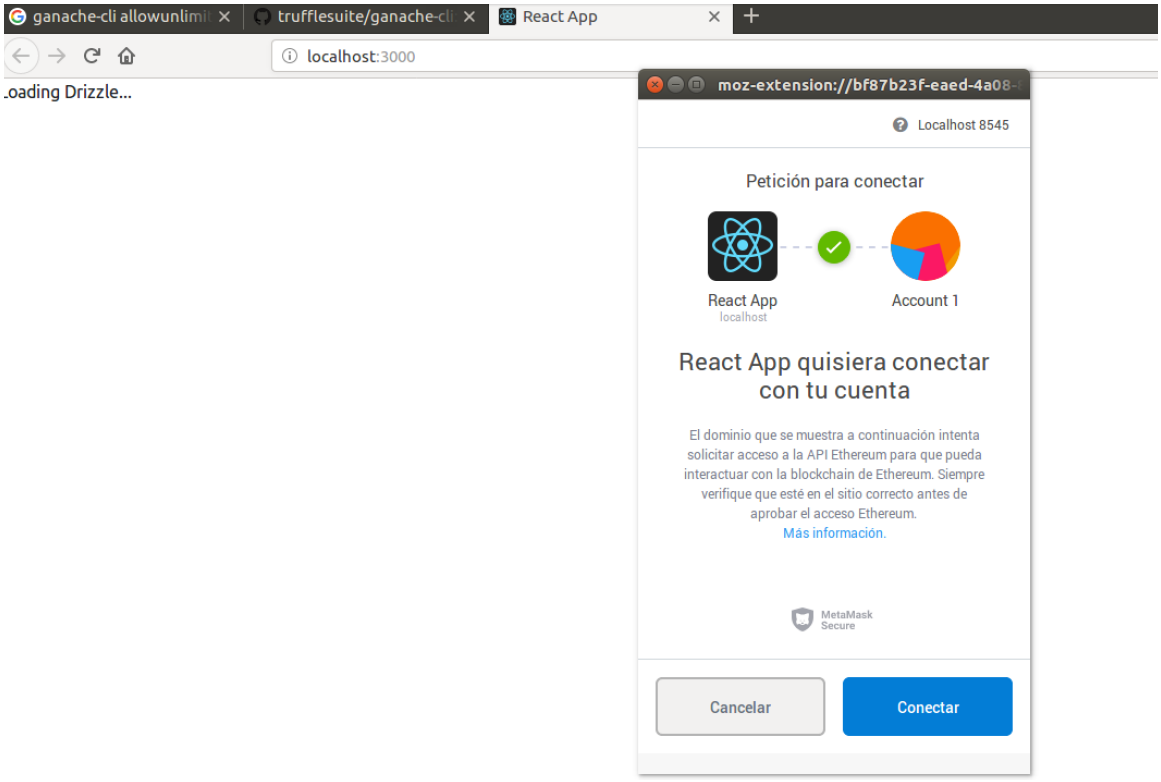


Figure 6.7: React App Initialization Process.

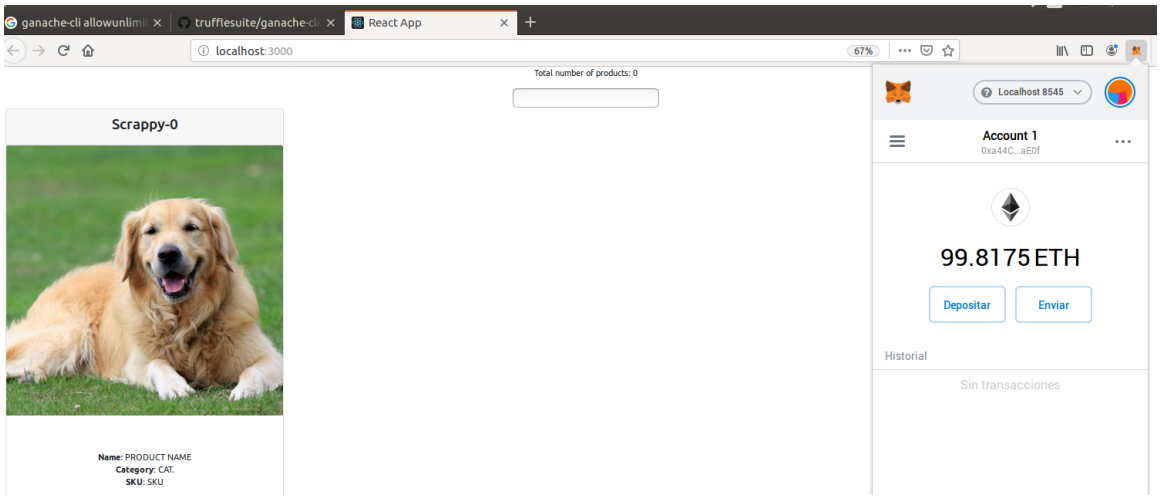


Figure 6.8: React App Landing Page.

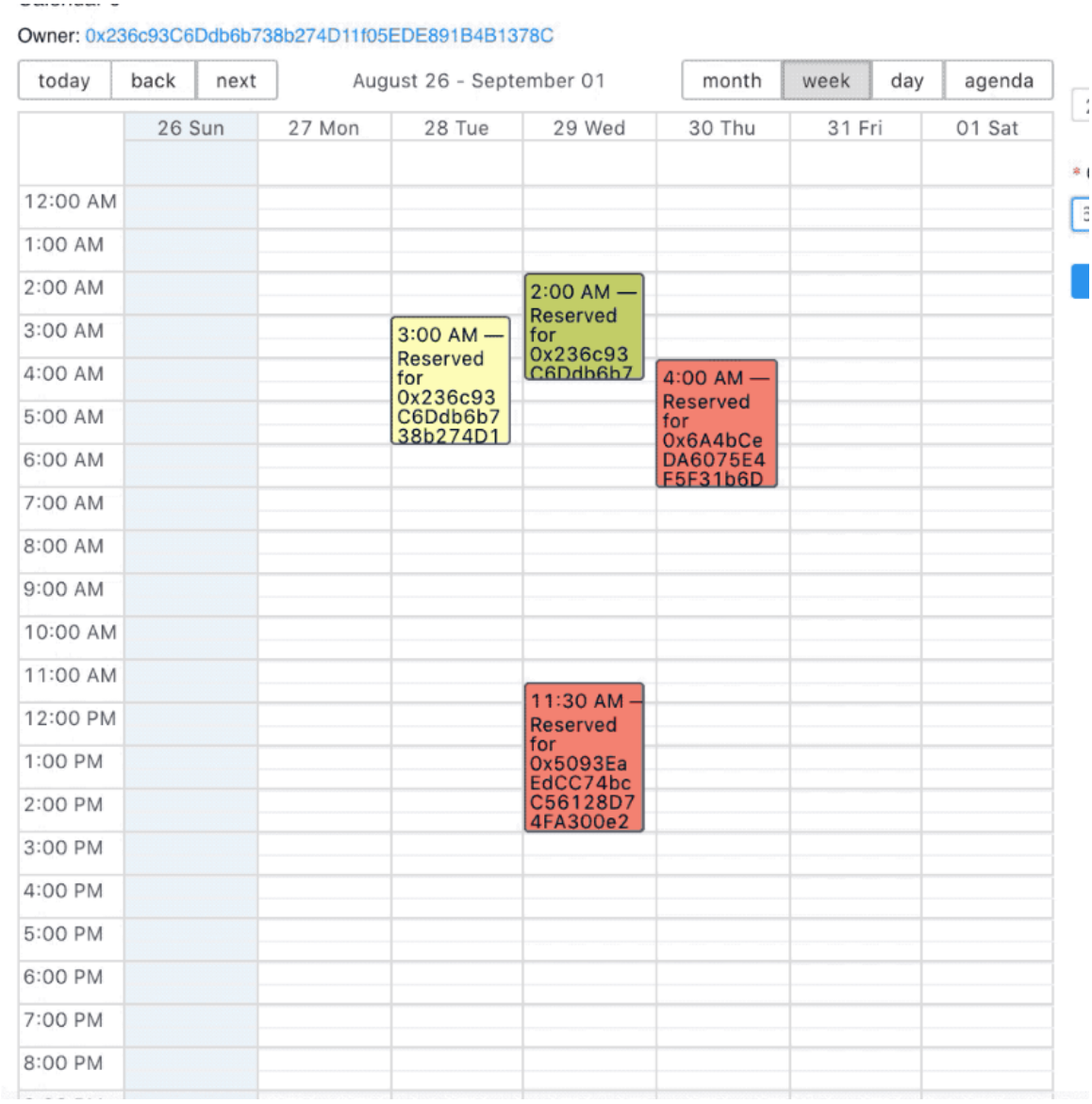


Figure 6.9: React App Calendar Example.

CHAPTER 7

Results Analysis

In this chapter, we make a critical analysis of the develop system as well as outline the main results obtained. We will determine if the concepts and ideas we had at the beginning have been supported by the system or in the other hand our ideas were wrong. We will outlined the most important ideas and knowledge acquired through the devlopment of the system.

7.1 Results Analysis and Final Thoughts

A booking, configurable system has been developed and deployed to a blockchain infrastructure. This is a major achievement, as no other booking system that worked in a decentralized environment was previously developed with the same characteristics that our implementation provides.

With this project, we have demonstrate that is possible to develop a booking system which is deployed to a blockchain, without the necessity of implementing any private tokens to enable the correct functionality and interaction between the different actors. Neither there is a need to have a private blockchain with different

configurations to the public blockchains. Although we were not able to deployed the final dapp to the Ethereum Network, it was not due to the inability of the system to work in public Blockchain but because of additional tests and refactoring we decided to do before migrating it to the mainnet.

We have also proof that it is possible to re use several existing and secure libraries as well as implement Ethereum standards in order to provide our system with lost of valuable functionality. The ERC721 Ethereum standard fits perfect for our case.

Additionally, several ideas of what should be handled on-chain and what should be processed off-chain have been provided. Nevertheless, a more profound study needs to be done in order to set the barrier between the requried on-chain logic and off-chain processes. We have slightly mention through the project what parts we think should be better remain in off-chain. What is clear, is that still it is better not to have a completely decentralized booking system that runs in the blockchain. This is not because of technology or design patterns barriers, but because it will be very costly and difficult to have all the shops migrate to the blockchain. Specially, when this technology is still being developed and everyday we have new software.

CHAPTER 8

Conclusions And Future Work

This chapter covers the conclusions reached from the work done. An analysis of the acquired knowledge is done as well as whether we have fulfilled the initial established objectives or not. Finally, future work is suggested for completing and extending the scope of this project.

8.1 Conclusions

Through the development of this project, it has been necessary to learn and put in practice several new technologies. It has been covered not only what it will be the same to a back-end developer of a Blockchain, which will be in charge of developing, maintaining and setting up an environment that runs a node on the blockchain and deploys or interact with smart contracts under the hood, but also we have acquired Front-end Blockchain developer experience by developing a Dapp that provides a user interface for interacting with the smart contracts ina friendly way.

We can differentiate between the technical part of a Blockchain, which consists of understanding and working with the protocols, consensus mechanisms, cryptographic

algorithms, clients set-up and mining processes among other things, and the practical part of the blockchain, that consists in the development of smart contracts and Decentralized applications that run in a blockchain. With this project we have covered both parts. First the technical part for understanding in depth the Ethereum structure, and then the practical part for developing a booking system that runs in a decentralized infrastructure. We have also used several tools, standards, libraries and plugins which have been covered extensively.

It can be said that our initial thoughts of being able to deploy a booking solution that really empowers the potential of blockchain without introducing concepts such as private tokens that limit this potential have been successfully proved. Also, it can be concluded that implementing smart contracts standards, and designing the system so that it fits and implements these standard interfaces is one of the best decisions as it automatically transform the system in a free totally interoperable solution.

Regarding the new versions and software that constantly is being introduced in the blockchain field, it has created lots of difficulties as several times this new modifications introduced breaking changes which brought down the already functioning solutions. This is why it is really important to be really thorough in configuring correctly the files that determine the required software of our application for being able to run as well as the version for each package. In addition, it is really necessary to track the work through control version tools such as Git, specially due to the fact commented before.

It has been proved that a proof of concept system that is general and not bounded to a specific type of bookable products can be deployed and manage by the same smart contract. We need to consider from now on ways of reducing the cost of transactions in order to increase the value of our system while reducing the cost of making use of it for its users. This way, this project may rise worries among existing marketplaces that will need to re-invent or introduce innovations in their business models in order to maintain their privileged and leading position.

8.2 Future Work

Although all the objectives have not been achieved, most of them have been fulfilled. Taking this project as a starting point, future studies should be focused in introducing blockchain standards in all the booking system logics so that it is possible to reach an scenario in which any platform can migrate from a marketplace or decentralized solution to another without losing its data, reviews, etc. Also, further work should be done regarding the storage of the bookings in the blockchain. Our approach has been to implement a TreeMap but other solutions should be covered and compared against the proposed one though this project.

Further tests must be done to the full dapp, and a migration to Alastria Blockchain is a good continuation of this project, as it has been one of the non reached objectives.

Finally, it is really important to test the migration of the systems that real shops use daily in their offices. Starting from a decentralized booking system such as the one developed in this project, plugins or similar light software should be developed in order to enable the migration of universally used tools such as Wordpress.

Bibliography

[1] Blockchain technology - blockchaintechnologies.com. https://www.blockchaintechnologies.com/blockchain-technology.

[2] Ethereum wiki - ethereum clients (github.com/wiki).

[3] https://git-scm.com.

[4] Cb insights - banking is only the beginning: 55 big industries blockchain could transform. https://www.cbinsights.com/research/industries-disrupted-blockchain, June 2019.

[5] Solidity documentation. https://solidity.readthedocs.io/en/latest/, 2019.

[6] Andreas M Antonopoulos and Gavin Wood. *Mastering ethereum: building smart contracts and dapps*. O'Reilly Media, 2018.

[7] Vitalik Buterin et al. Ethereum white paper. *GitHub repository*, 2013.

[8] Vidal Chriqui and Hervé Hababou. Booking token unit (btu) protocol whitepaper. https://www.btu-protocol.com/pdf/whitepaper.pdf, February 2018.

[9] Ethereum Community. Ethereum improvement proposals (eips) official site, 2019.

[10] Wei Dai. B-money. *Consulted*, 1:2012, 1998.

[11] Deloitte. Deloitte 2019 global blockchain survey, 2019.

[12] William Entriken, Dieter Shirley, Jacob Evans, and Nastassia Sachs. Eip 721: Erc-721 non-fungible token standard. https://eips.ethereum.org/EIPS/eip-721, January 2018.

[13] David Flanagan. *JavaScript: the definitive guide*. O'Reilly Media, Inc, 2006.

[14] Anton Gasol. Blockchain, the internet of value - the new barcelona post. https://www.thenewbarcelonapost.com/en/blockchain-the-internet-of-value/, April 2018.

[15] Satoshi Nakamoto. Bitcoin white paper, 2008.

[16] David Siegel. Understanding the dao attack. *Retrieved June*, 13:2018, 2016.

[17] Don Tapscott and Alex Tapscott. *Blockchain revolution: how the technology behind bitcoin is changing money, business, and the world*. Penguin, 2016.

[18] Wattana Viriyasitavat and Danupol Hoonsopon. Blockchain characteristics and consensus in modern business processes. *Journal of Industrial Information Integration*, 13:32–39, 2019.

[19] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[20] Zibin Zheng, Shaoan Xie, Hongning Dai, Xiangping Chen, and Huaimin Wang. An overview of blockchain technology: Architecture, consensus, and future trends. In *2017 IEEE International Congress on Big Data (BigData Congress)*, pages 557–564. IEEE, 2017.

APPENDIX A

ANEXO A

Anexo A

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA
I.C.A.I.

PROYECTOS FIN DE MÁSTER
CURSO:

Ficha de proyecto fin de máster
(RELLENAR CON LETRAS DE IMPRENTA EN ORDENADOR)

Titulación y optatividad: Máster Universitario en Ingeniería de Telecomunicación y MBA (MIT-MBA)

Alumno 1º Apellido: Gericke
2º Apellido: Parga
Nombre: Álvaro Guillermo

Teléfono de contacto: +1 312-610-2114 / +34 689-077-659
e-mail: alvgericke@gmail.com

Título del Proyecto Fin de Máster:
Diseño e implementación de una solución basada en tecnología Blockchain para lograr la descentralización de un sistema de alquileres aplicando Smart Contracts conformes a los estándares de Ethereum. Aplicación de dicha solución en industria real.

Director (nombre y dos apellidos): Carlos Pastor Matut
Teléfono de contacto: +34 629-479-312
e-mail: cpastor@grupobme.es

Breve descripción del proyecto (5 o 6 líneas)

Se desarrollarán una serie de Smart Contracts en Ethereum que permitan trasladar todo el sistema de alquileres en el que participan dos o tres tipos de agentes (tiendas que poseen los productos, usuarios que alquilan y posibilidad de intermediarios) a una solución descentralizada y automatizada en la Blockchain de Ethereum. Dichos Smart Contracts deberán respetar e implementar el estándar ERC-721 de Ethereum para facilitar su integración con distintos cryptoexchanges así como pagos con distintos tipos de divisas o cryptotokens. Los Smart Contracts se programarán en Solidity, y se investigará la forma de programar las funciones del estándar de tal manera que las transacciones sean lo más baratas posibles. Así mismo, se estudiarán las distintas posibilidades de persistencia de los datos que podemos utilizar en Ethereum (variables en Smart contracts, eventos o IPFS) con la finalidad de desplegar una solución lo más eficiente desde el punto de vista económico. Se desplegará una solución en la testnet de Ethereum (Ropsten) en la que se implementará una solución para un caso de uso real. Adicionalmente, se desplegará dicha solución en Alastria.

Se realizarán los siguientes desarrollos y se harán uso de las siguientes tecnologías principalmente:

- Sincronización y configuración de un Full-Node de Etheruem utilizando el cliente Parity.
- Desarrollo de Smart contracts en Ethereum usando Solidity.
- Estudio e implementación del estándar ERC-721 de Ethereum.
- Estudio de las distintas posibilidades de persistencia para reducir lo máximo posible el coste de almacenamiento de datos y variables en el Smart Contract.
- Despliegue del Smart contract en Ropsten, la Testnet de Ethereum.
- Implementación de un Front-End que se comunique con la Blockchain de Ethereum y permite a un usuario interactuar con el Smart Contract. (Desarrollo off-chain).
- Implementación de un caso de uso real.
- Adaptación y despliegue en la Blockchain Alastria.

El documento final del proyecto será subido al Repositorio Institucional de Comillas con acceso público. El alumno podrá solicitar un nivel restringido de acceso (incluido el "cerrado" o "confidencial") que podrá concederse, excepcionalmente, si está plenamente justificado.

The final report of the Project will be uploaded to the Comillas Institutional Repository with public access. The student will be able to ask for a restricted access (even "closed" or "confidential") which will be exceptionally accepted if it is fully justified.

Aceptación del Director (firma y fecha)

30-4-2019

Carlos Pastor Matat

DNI 52579685F

DATOS RELATIVOS AL PROYECTO FIN DE GRADO

Título del Proyecto Fin de Grado:
Diseño e Implementación de un Marketplace para una Startup basado en un Sistema de Gestión de Contenidos. Aplicación de un Sistema de Análisis de Sentimientos y Emociones para Conocer y Mejorar la Experiencia de Usuario.

Director/es del Proyecto Fin de Grado:
Carlos A. Iglesias Fernández

Curso Académico en el que se realizó:
2016 - 2017

Universidad (indicarla si no es Comillas):
UNIVERSIDAD POLITÉCNICA DE MADRID
ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN

En el caso de realización en Comillas, indicar especialidad en el Grado: