



**COMILLAS**

UNIVERSIDAD PONTIFICIA

ICAI

# GRADO EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

**SMART CONTRACTS: Different approaches to  
introduce external (off-chain) data**

Autor: Gonzalo Tamariz-Martel Sánchez

Director: Seth James Nielson

Madrid



I declare, under my responsibility, that the Project presented with the title  
SMART CONTRACTS: Different approaches to introduce external (off-chain) data  
at the ETS of Engineering - ICAI of the Universidad Pontificia Comillas in the  
academic year 2021/22 is of my authorship, original and unpublished and has not been  
previously submitted for other purposes.

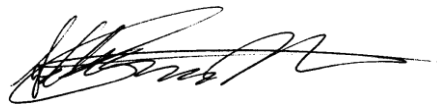
The Project is not plagiarized from another, neither totally nor partially, and the  
information that has been taken from other documents is duly referenced.



Signed: Gonzalo Tamariz-Martel Sánchez Date: 23/ 06/ 2022

Authorized the presentation of the project

THE SUPERVISING INSTRUCTOR



Signed: Seth James Nielson Date: 13/07/2022





**COMILLAS**

UNIVERSIDAD PONTIFICIA

ICAI

# GRADO EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

**SMART CONTRACTS: Different approaches to  
introduce external (off-chain) data**

Autor: Gonzalo Tamariz-Martel Sánchez

Director: Seth James Nielson

Madrid

# Agradecimientos

Me gustaría agradecer a mi familia y amigos por el apoyo que siempre me han dado confiando en mí. También a todos los profesores que me han ido guiando a lo largo de mi trayectoria académica.

Gracias.



# **SMART CONTRACTS: DIFFERENT APPROACHES TO INTRODUCE EXTERNAL (OFF-CHAIN) DATA**

**Autor: Tamariz-Martel Sánchez, Gonzalo**

Director: Nielson, Seth James

Entidad Colaboradora: ICAI – Universidad Pontificia Comillas

## **RESUMEN DEL PROYECTO**

Los Smart Contracts son programas que se almacenan en una cadena de bloques o blockchain y pueden utilizarse para ejecutar un acuerdo cuando se cumplen unas condiciones. Un problema que surge con los Smart Contracts es la dificultad de introducir datos externos ya que son programas que se ejecutan simultáneamente en distintos nodos y todos deben obtener un mismo resultado. En este proyecto se exploran distintos métodos para introducir datos externos (off-chain) a los Smart Contract que se encuentran en un blockchain.

**Palabras clave:** Blockchain, Smart Contract, Oracles, Ethereum, Chainlink, Dapp

### **1. Introducción**

El Blockchain se ha convertido en una tecnología en boga y que abre las puertas a una infinidad de posibilidades para cambiar el mundo tal y como lo conocemos hoy en día. Algunas de las aplicaciones o sectores donde se está empleando Blockchain en la actualidad son las criptomonedas, banca y finanzas, registro de propiedades o Smart Contracts.

Blockchain hace referencia a una base de datos distribuida que se comparte a través de distintos nodos de una red informática. Estas bases de datos en lugar de almacenar la información en tablas como las bases de datos tradicionales almacenan la información en bloques. La información es almacenada en bloques hasta que su capacidad está completa, y cuando esto ocurre, se cierra el bloque y se vincula mediante criptografía al bloque anterior formando una cadena de bloques tal y como su nombre indica. Esta forma de almacenar los datos permite garantizar la fidelidad y seguridad de un registro de datos generando un sistema confiable sin la necesidad de un tercero.

Una de las aplicaciones del Blockchain y en la que se va a centrar este proyecto son los Smart Contracts. Podemos entender los Smart Contracts como programas que se almacenan en una cadena de bloques y que se pueden emplear para facilitar, validar o verificar un acuerdo contractual. Es decir, son programas que se ejecutan automáticamente cuando se cumplen unas condiciones previamente acordadas por los usuarios y escritas en código.

### **2. Definición del proyecto**

Uno de los principales problemas que presentan los Smart Contracts es que son aplicaciones que se ejecutan en una red Blockchain, que es una red distribuida. Esto significa que cada nodo debe ejecutar el Smart Contract de forma independiente y obtener los mismos resultados para una entrada determinada. Esto supone un reto a la hora de introducir los datos desde fuera de la cadena (off-chain), ya que si, por ejemplo, todos los nodos realizasen una llamada a una API, los resultados podrían



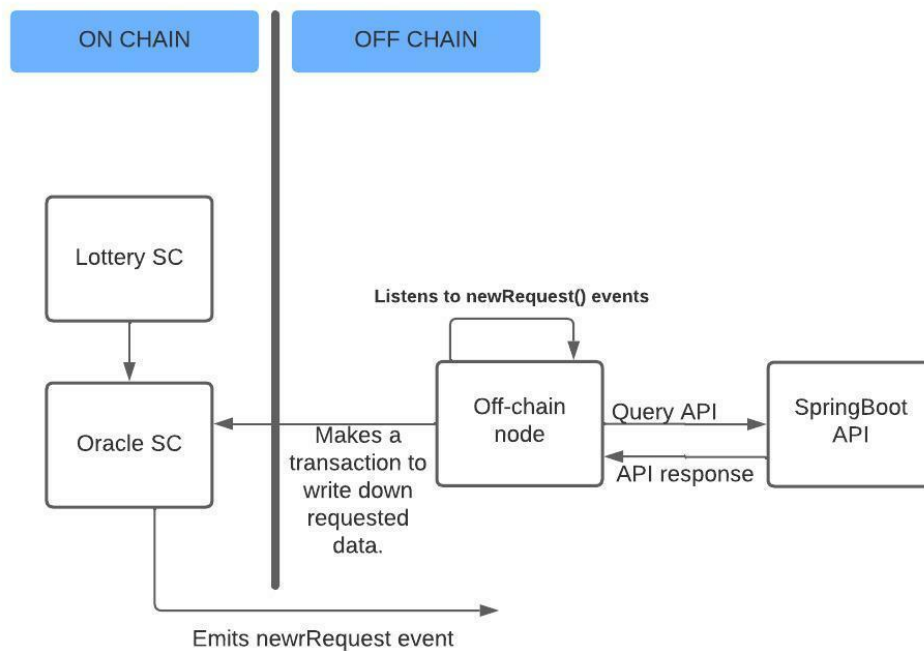
variar si las llamadas se producen en diferentes instantes temporales o la API es hackeada.

En este proyecto se busca implementar distintos métodos para introducir datos desde fuera de la cadena. Para ello se han planteado tres alternativas diferentes: un modelo de oráculo centralizado, un oráculo descentralizado y el uso de una red oracular descentralizada como Chainlink.

### 3. Descripción del modelo/sistema/herramienta

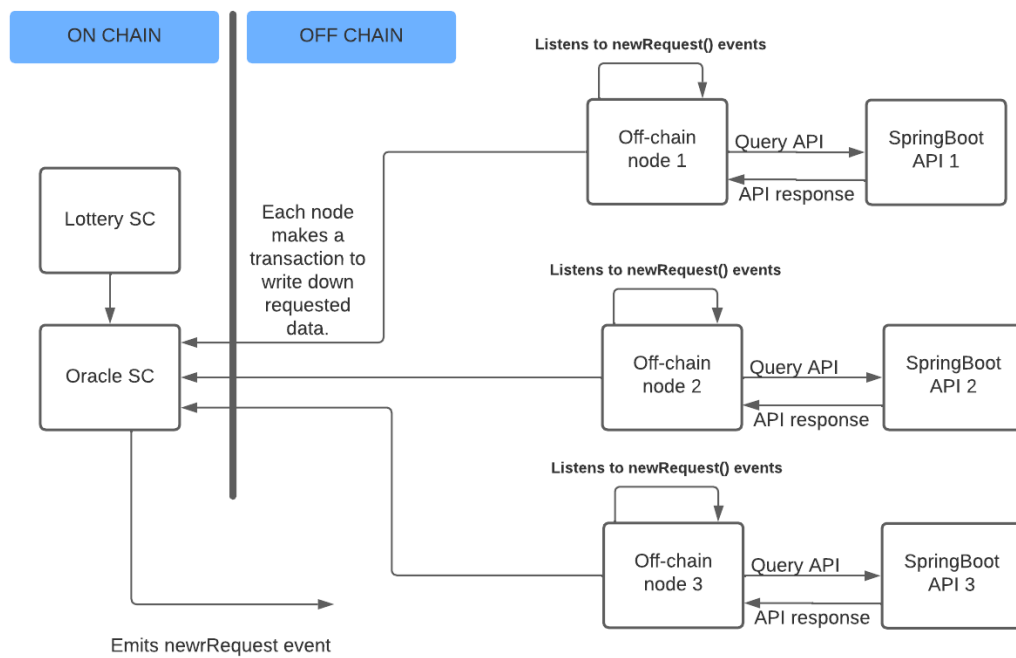
Para este proyecto se ha decidido realizar una sencilla aplicación de loterías. Como los Smart Contract no pueden generar números aleatorios, ya que es un programa que se ejecuta en diferentes nodos de forma simultánea y todos deben obtener el mismo resultado, este será el dato que introduzcamos desde el mundo exterior. En base a este ejemplo sencillo, realizaremos pruebas de los distintos modelos implementando cada uno de ellos.

Para el primer modelo, es decir, el oráculo centralizado se ha diseñado la siguiente arquitectura para introducir datos desde el exterior de la cadena a esta misma (Ilustración 1).



*Ilustración 1- Arquitectura oráculo centralizado*

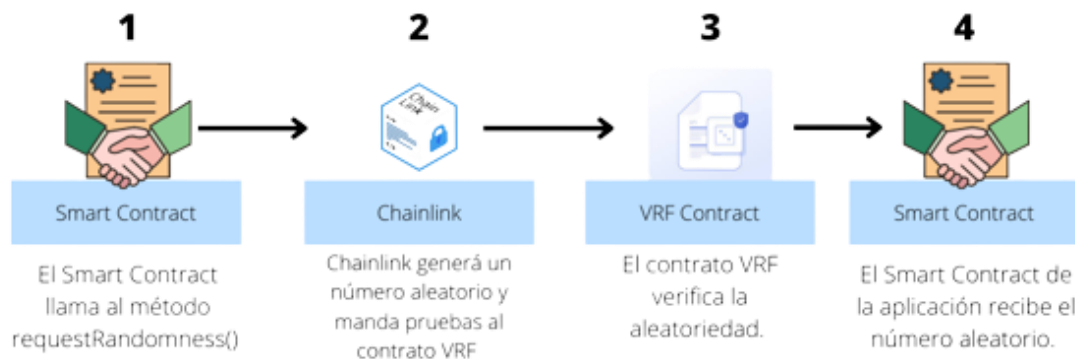
El segundo modelo se trata de un oráculo descentralizado. En este caso, se ha empleado la siguiente arquitectura (Ilustración 2).



*Ilustración 2 - Arquitectura oráculo descentralizado*

En tercer lugar, se ha empleado la plataforma de Chainlink y el servicio que ofrece para la generación de números aleatorios Verifiable Random Function (VRF).

A continuación, se muestra un esquema del proceso para introducir un número aleatorio empleando Chainlink VRF (Ilustración 3).



*Ilustración 3- Arquitectura Chainlink VRF*

Para el desarrollo de los Smart Contract necesarios y los scripts de Python se ha empleado el framework Brownie.

## 4. Resultados

Tras programar e implementar los tres modelos para introducir un dato externo a la aplicación se han podido apreciar los siguientes resultados.

En el caso del modelo de oráculo centralizado, se ha podido ver que supone un riesgo para la seguridad de nuestro Smart Contract. Al existir un único nodo fuera de la cadena que actúa de forma centralizada, dicho nodo tiene el control de nuestro Smart Contract. Aunque a priori se trata de un nodo confiable, debemos tener en cuenta que dicho nodo puede ser atacado o actuar de forma maliciosa.

Por otra parte, se ha visto como esa amenaza de seguridad se resuelve al emplear un modelo de oráculo descentralizado. En el ejemplo que se ha empleado para este proyecto, es decir, una aplicación de loterías, el modelo descentralizado es bastante seguro ya que el dato que buscamos introducir es un número aleatorio y con que alguno de los nodos externos proporcione como dato un número aleatorio, al agregar todos los datos la respuesta final será un número aleatorio.

En el caso de Chainlink VRF se ha conseguido implementar siguiendo la documentación oficial y se ha comprobado que funciona correctamente.

## 5. Conclusiones

Finalmente, se ha realizado una comparación entre los distintos modelos.

El modelo de oráculo centralizado es el modelo más sencillo y que al realizar una única transacción para introducir el dato desde el exterior al Smart Contract que se encuentra en la blockchain también tiene un coste mínimo. Debemos recordar que, en Ethereum, todas las transacciones que realizan modificaciones tienen un coste por la tasa de gas. En su contra, se trata de un modelo inseguro que destruye la descentralización que el blockchain y los Smart Contract ofrecen.

Por otra parte, el modelo descentralizado consigue resolver casi totalmente el problema de seguridad del modelo anterior, pero a cambio se trata de un modelo más complejo que requiere un mayor número de recursos. Además, su coste es variable en función del número de nodos externos que se empleen, ya que cada uno de ellos escribirá un dato en la blockchain realizando una transacción por la cual se deberán pagar las tasas de gas asociadas. A mayor número de nodos, mayor será el coste, pero estaremos generando un sistema más seguro contra los ataques.

Por último, Chainlink VRF es una opción excelente y muy sencilla de implementar. Al tratarse de un servicio de Chainlink nos facilita mucho la programación ya que solo debemos seguir sus instrucciones para implementarlo y usarlo. Por otra parte, la red descentralizada de Chainlink es considerada uno de los oráculos más seguros hoy en día. Eso sí, para emplearlo, debemos pagar unos incentivos en forma de su token ERC-20 llamado LINK por cada transacción que deseemos realizar en su red.

A continuación, se muestra una tabla resumen comparativa entre los distintos modelos:

Modelo	Dificultad implementación	Coste	Seguridad
Oráculo centralizado	Sencillo	Poco	Inseguro
Oráculo descentralizado	Complejo	Depende del número de nodos externos	Seguro
Chainlink VRF	Muy sencillo	Fijo - LINK	Muy seguro

Tabla 1 - Comparativa distintos modelos implementados

## 6. Referencias

- [1] *Blockchain Explained*. (2022, 20 enero). Investopedia. <https://www.investopedia.com/terms/b/blockchain.asp>
- [2] C. (2021, 8 diciembre). *Hybrid Smart Contracts Explained*. Chainlink Blog. <https://blog.chain.link/hybrid-smart-contracts-explained/>
- [3] Collins, P. (2022, 4 febrero). *What is a blockchain oracle? What is the oracle problem? Why can't blockchains make API calls? This is everything you need to know about off-chain data | Better Programming*. Medium. <https://betterprogramming.pub/what-is-a-blockchain-oracle-f5ccab8dbd72>
- [4] colaboradores de Wikipedia. (2022, 11 febrero). *Ethereum*. Wikipedia, la enciclopedia libre. <https://es.wikipedia.org/wiki/Ethereum>
- [5] Costa, P. (2021, 10 diciembre). *Implementing a Blockchain Oracle on Ethereum - Pedro Costa*. Medium. <https://medium.com/@pedrodc/implementing-a-blockchain-oracle-on-ethereum-cedc7e26b49e>
- [6] Hovsepyan, N. (2018, 20 junio). *Lottery Smart Contract: Can we generate random numbers In Solidity?* Medium. <https://medium.com/@promentol/lottery-smart-contract-can-we-generate-random-numbers-in-solidity-4f586a152b27>



# **SMART CONTRACTS: DIFFERENT APPROACHES TO INTRODUCE EXTERNAL (OFF-CHAIN) DATA**

**Author: Tamariz-Martel Sánchez, Gonzalo**

Supervisor: Nielson, Seth James

Collaborating Entity: ICAI – Universidad Pontificia Comillas

## **ABSTRACT**

Smart Contracts are programs that are stored in a blockchain and can be used to execute an agreement when certain conditions are met. A problem that arises with Smart Contracts is the difficulty of introducing external data since they are programs that are executed simultaneously in different nodes and all must obtain the same result. In this project we explore different methods to introduce external data (off-chain) to Smart Contracts that are in a blockchain.

**Keywords:** Blockchain, Smart Contract, Oracles, Ethereum, Chainlink, Dapp, Decentralized Oracle

## **1. Introduction**

Blockchain has become a technology in vogue and opens the doors to an infinite number of possibilities to change the world as we know it today. Some of the applications or sectors where Blockchain is currently being used are cryptocurrencies, banking and finance, property registration or Smart Contracts.

Blockchain refers to a distributed database that is shared through different nodes of a computer network. These databases instead of storing information in tables like traditional databases store information in blocks. The information is stored in blocks until its capacity is full, and when this happens, the block is closed and linked by cryptography to the previous block forming a block chain as its name indicates. This way of storing data allows to guarantee the fidelity and security of a data record generating a reliable system without the need of a third party.

One of the applications of the Blockchain and on which this project is going to focus is Smart Contracts. We can understand Smart Contracts as programs that are stored in a blockchain and can be used to facilitate, validate or verify a contractual agreement. In other words, they are programs that are automatically executed when conditions previously agreed upon by users and written in code are met.

## **2. Project definition**

One of the main problems with Smart Contracts is that they are applications running on a Blockchain network, which is a distributed network. This means that each node must run the Smart Contract independently and obtain the same results for a given input. This poses a challenge when entering data from outside the chain (off-chain), since if, for example, all nodes were to make a call to an API, the results could vary if the calls occur at different time instants or the API is hacked.

This project seeks to implement different methods to introduce data from outside the chain. For this purpose, three different alternatives have been proposed: a centralized oracle model, a decentralized oracle and the use of a decentralized oracle network such as Chainlink.

### 3. Description of the model/system/tool

For this project it has been decided to realize a simple lottery application. As Smart Contracts cannot generate random numbers, since it is a program that runs on different nodes simultaneously and all of them must obtain the same result, this will be the data that we introduce from the outside world. Based on this simple example, we will test the different models by implementing each of them.

For the first model, i.e. the centralized oracle, the following architecture has been designed to introduce data from outside the chain (Figure 1).

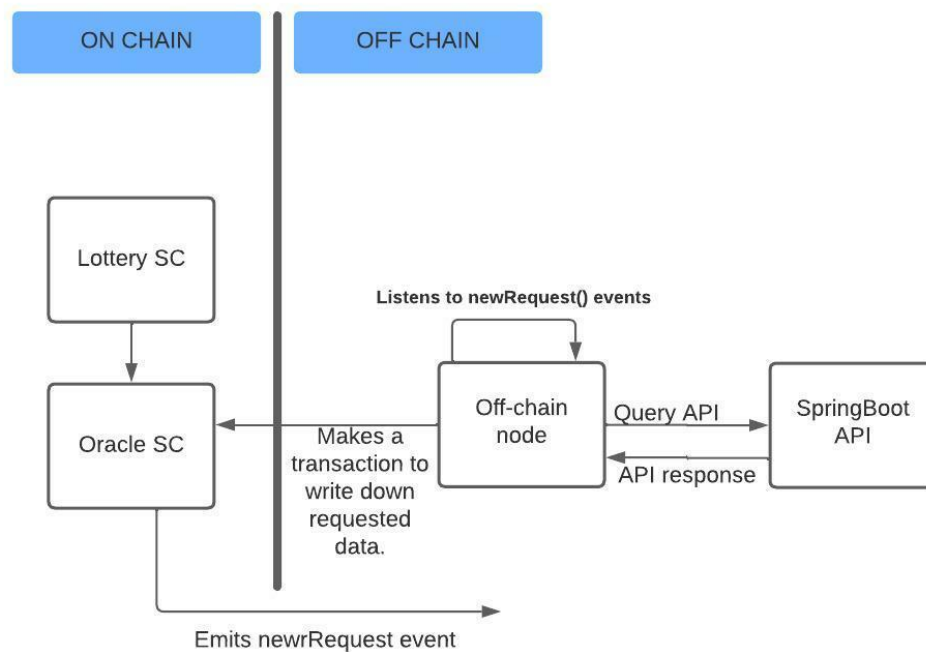


Figure 4- Arquitectura oráculo centralizado

The second model is a decentralized oracle. In this case, the following architecture has been used (Figure 2).

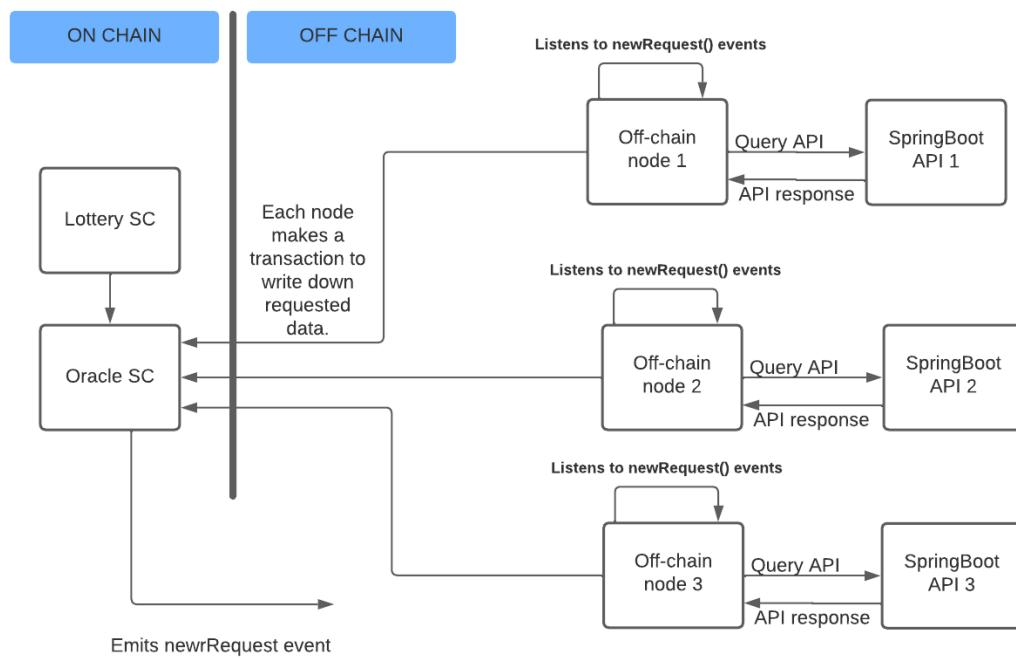


Figure 5 - Arquitectura oráculo descentralizado

Thirdly, the Chainlink platform and the service it offers for the generation of Verifiable Random Function (VRF) random numbers have been used.

A schematic of the process for entering a random number using Chainlink VRF is shown below (Figure 3).

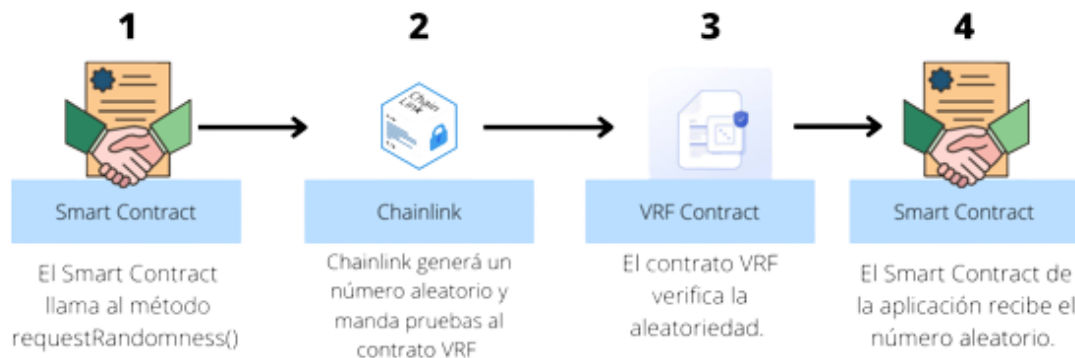


Figure 6- Arquitectura Chainlink VRF

The Brownie framework was used to develop the necessary Smart Contracts and Python scripts.



## **4. Results**

After programming and implementing the three models to introduce an external data to the application, the following results were obtained.

In the case of the centralized oracle model, it has been seen that it poses a risk to the security of our Smart Contract. As there is only one node outside the chain that acts in a centralized way, this node has control of our Smart Contract. Although a priori it is a trusted node, we must take into account that this node can be attacked or act maliciously.

On the other hand, it has been seen how this security threat is solved by employing a decentralized oracle model. In the example used for this project, i.e., a lottery application, the decentralized model is quite secure since the data we seek to enter is a random number and if any of the external nodes provides a random number as data, when all the data is aggregated, the final answer will be a random number.

In the case of Chainlink VRF it has been implemented following the official documentation and it has been proved to work correctly.

## **5. Conclusions**

Finally, a comparison has been made between the different models.

The centralized oracle model is the simplest model and that by performing a single transaction to introduce the data from outside to the Smart Contract that is in the blockchain also has a minimum cost. We must remember that, in Ethereum, all transactions that perform modifications have a cost for the gas fee. Against it, this is an insecure model that destroys the decentralization that the blockchain and Smart Contracts offer.

On the other hand, the decentralized model manages to almost completely solve the security problem of the previous model, but in exchange it is a more complex model that requires a greater number of resources. In addition, its cost is variable depending on the number of external nodes used, since each of them will write a piece of data in the blockchain performing a transaction for which the associated gas fees must be paid. The greater the number of nodes, the higher the cost, but we will be generating a more secure system against attacks.

Finally, Chainlink VRF is an excellent option and very simple to implement. As it is a Chainlink service, it makes programming much easier since we only have to follow their instructions to implement and use it. Moreover, Chainlink's decentralized network is considered one of the most secure oracles today. However, to use it, we must pay incentives in the form of its ERC-20 token called LINK for each transaction we wish to perform in its network.

A comparative summary table between the different models is shown below:

Model	Difficulty of implementation	Cost	Security
Centralized Oracle	Simple	Low	Insecure
Decentralized Oracle	Complex	Depends on the number of external nodes	Secure
Chainlink VRF	Very simple	Fixed - LINK	Highly secure

Tabla 2 - Comparativa distintos modelos implementados

## 6. References

- [1] *Blockchain Explained*. (2022, 20 enero). Investopedia. <https://www.investopedia.com/terms/b/blockchain.asp>
- [2] C. (2021, 8 diciembre). *Hybrid Smart Contracts Explained*. Chainlink Blog. <https://blog.chain.link/hybrid-smart-contracts-explained/>
- [3] Collins, P. (2022, 4 febrero). *What is a blockchain oracle? What is the oracle problem? Why can't blockchains make API calls? This is everything you need to know about off-chain data | Better Programming*. Medium. <https://betterprogramming.pub/what-is-a-blockchain-oracle-f5ccab8dbd72>
- [4] colaboradores de Wikipedia. (2022, 11 febrero). *Ethereum*. Wikipedia, la enciclopedia libre. <https://es.wikipedia.org/wiki/Ethereum>
- [5] Costa, P. (2021, 10 diciembre). *Implementing a Blockchain Oracle on Ethereum - Pedro Costa*. Medium. <https://medium.com/@pedrodc/implementing-a-blockchain-oracle-on-ethereum-cedc7e26b49e>
- [6] Hovsepyan, N. (2018, 20 junio). *Lottery Smart Contract: Can we generate random numbers In Solidity?* Medium. <https://medium.com/@promentol/lottery-smart-contract-can-we-generate-random-numbers-in-solidity-4f586a152b27>



## *Índice de la memoria*

<b>Capítulo 1. Introducción</b> .....	<b>9</b>
1.1 Blockchain .....	9
1.2 Smart Contract .....	11
<b>Capítulo 2. Descripción de las Tecnologías</b> .....	<b>13</b>
2.1 Ethereum.....	13
2.2 Testnet (Rinkeby).....	14
2.3 Metamask.....	15
2.4 Visual Studio Code.....	15
2.5 Brownie.....	16
2.6 WEB3.PY .....	17
2.7 Etherscan.....	17
2.8 Solidity.....	19
2.9 Ganache .....	19
2.10 Node.js.....	21
2.11 Infura .....	22
2.12 Spring Boot .....	23
<b>Capítulo 3. Estado de la Cuestión</b> .....	<b>24</b>
<b>Capítulo 4. Definición del Trabajo</b> .....	<b>27</b>
4.1 Justificación .....	27
4.2 Objetivos.....	29
4.3 Metodología.....	30
4.4 Planificación y Estimación Económica .....	32
4.4.1 Planificación temporal.....	32
4.4.2 Estimación económica del proyecto.....	34
<b>Capítulo 5. Blockchain Key Concepts</b> .....	<b>40</b>
5.1 Información general.....	40
5.2 Tipos de blockchain.....	42
5.3 ¿Cómo funciona una cadena blockchain?.....	43

5.3.1 HASH .....	43
5.3.2 Algoritmos de consenso.....	44
5.3.3 Bloques.....	46
5.4 Transacciones .....	50
5.5 Gas.....	52
<b>Capítulo 6. Sistema/Modelo Desarrollado.....</b>	<b>53</b>
6.1 Arquitectura .....	53
6.2 Smart contract: Aplicación de lotería .....	56
6.3 Nodo off-chain .....	60
6.4 API EN SPRINGBOOT.....	64
<b>Capítulo 7. Oráculo centralizado .....</b>	<b>67</b>
7.1 Ejemplo en rinkeby.....	70
7.2 Ejemplo de inseguridad del modelo centralizado .....	73
<b>Capítulo 8. Oráculo descentralizado .....</b>	<b>77</b>
8.1 Ejemplo Rinkeby.....	80
8.2 Seguridad del modelo descentralizado .....	85
<b>Capítulo 9. Chainlink VRF.....</b>	<b>92</b>
9.1 Ejemplo Rinkeby.....	95
<b>Capítulo 10. Conclusiones y Trabajos Futuros .....</b>	<b>100</b>
<b>Capítulo 11. Bibliografía.....</b>	<b>104</b>
<b>ANEXO I: ALINEACIÓN DEL PROYECTO CON LOS ODS.....</b>	<b>108</b>
<b>ANEXO II 110</b>	
11.1 Manual de instalación .....	110
11.1.1 Metamask.....	110
11.1.2 Visual studio code.....	112
11.1.3 Instalación de Python y paquetes necesarios .....	114
11.1.4 Brownie .....	115
11.2 Código fuente.....	116
11.2.1 node_offchain.py.....	116

<i>11.2.2 deploy_lottery.py</i> .....	<i>120</i>
<i>11.2.3 helpful_scripts.py</i> .....	<i>123</i>
<i>11.2.4 lottery.sol</i> .....	<i>126</i>
<i>11.2.5 CentralizedApproach/oracle.sol</i> .....	<i>128</i>
<i>11.2.6 DecentralizedApproach/oracle.sol</i> .....	<i>130</i>
<i>11.2.7 Chainlink/lottery.sol</i> .....	<i>132</i>
<i>11.2.8 brownie-config.yaml</i> .....	<i>134</i>

## *Índice de Ilustraciones*

Ilustración 1- Arquitectura oráculo centralizado .....	9
Ilustración 2 - Arquitectura oráculo descentralizado .....	10
Ilustración 3- Arquitectura Chainlink VRF.....	10
Figure 4- Arquitectura oráculo centralizado .....	15
Figure 5 - Arquitectura oráculo descentralizado .....	16
Figure 6- Arquitectura Chainlink VRF .....	16
Ilustración 7 - Logo de Ethereum .....	13
Ilustración 8- Estadísticas de Rinkeby .....	14
Ilustración 9 – Logo de Metamask .....	15
Ilustración 10 - Icono de Visual Studio Code .....	15
Ilustración 11 - Logo de Brownie.....	16
Ilustración 12 - Logo de Etherscan .....	17
Ilustración 13 - Página principal de EtherScan (Rinkeby).....	18
Ilustración 14 - Icono de Solidity .....	19
Ilustración 15 - Logo de Ganache.....	19
Ilustración 16- Ejemplo de ganache-cli.....	20
Ilustración 17 - Ejemplo de Ganache UI.....	21
Ilustración 18 - Logo de Node.js .....	21
Ilustración 19 - Logo de Infura.....	22
Ilustración 20- Logo de Spring Boot .....	23
Ilustración 21 - Logo de Chainlink.....	26
Ilustración 22 - Diagrama de Gantt del proyecto.....	32
Ilustración 23 - Panel Kanban del proyecto .....	33
Ilustración 24 - Logo de Bitcoin.....	40
Ilustración 25 - Estructura de bloques de una blockchain.....	41
Ilustración 26 - Clasificación de los tipos de blockchain.....	42

Ilustración 27 - Ejemplo 1 de SHA256.....	43
Ilustración 28 - Ejemplo 2 de SHA256.....	43
Ilustración 29 - Bloque antes de ser minado .....	47
Ilustración 30 - Bloque después de ser minado .....	48
Ilustración 31 - Ejemplo blockchain (I).....	49
Ilustración 32- Ejemplo blockchain (II).....	49
Ilustración 33 - Ejemplo blockchain (III) .....	49
Ilustración 34 - Public/ Private Key Pair.....	50
Ilustración 35 - Ejemplo de firma de una transacción .....	51
Ilustración 36 - Ejemplo verificación de transacción correcta .....	51
Ilustración 37 - Ejemplo verificación de transaccion incorrecta .....	51
Ilustración 38 - Arquitectura oráculo centralizado .....	53
Ilustración 39 - Arquitectura oráculo descentralizado .....	55
Ilustración 40 - Ejemplo Rinkeby centralizado: transacciones (I).....	70
Ilustración 41 - Ejemplo Rinkeby centralizado: nodo offchain (II).....	71
Ilustración 42 - Ejemplo Rinkeby centralizado: resultados (III) .....	72
Ilustración 43 - Ejemplo Rinkeby centralizado: Etherscan (IV) .....	72
Ilustración 44 - Ejemplo Rinkeby centralizado: Etherscan transacciones internas (V) .....	73
Ilustración 45- Ejemplo malicioso Rinkeby Centralizado: Desplegar el contrato (I).....	74
Ilustración 46 - Ejemplo malicioso Rinkeby centralizado: Inicializar nodo off chain (II) ..	74
Ilustración 47 - Ejemplo malicioso Rinkeby centralizado: transacciones (III).....	75
Ilustración 48 - Ejemplo malicioso Rinkeby centralizado: resultados (V) .....	75
Ilustración 49 - Ejemplo malicioso Rinkeby centralizado: nodo off-chain (IV).....	75
Ilustración 50 - Ejemplo Rinkeby descentralizado: despliegue del contrato (I) .....	80
Ilustración 51- Ejemplo Rinkeby descentralizado: nodo off-chain 1 (II) .....	80
Ilustración 52 - Ejemplo Rinkeby descentralizado: nodo off-chain 2 (III) .....	81
Ilustración 53 - Ejemplo Rinkeby descentralizado: nodo off-chain 3 (IV).....	81
Ilustración 54 - Ejemplo Rinkeby descentralizado: transacciones (V).....	82
Ilustración 55 - Ejemplo Rinkeby descentralizado: resultados (VI).....	84
Ilustración 56 - Ejemplo Rinkeby descentralizado: Etherscan (VII).....	84



Ilustración 57 - Ejemplo Rinkeby descentralizado: transacciones internas Etherscan (VIII)	85
Ilustración 58 - Ejemplo Rinkeby descentralizado seguro: desplegar contrato	86
Ilustración 59 - Ejemplo Rinkeby descentralizado seguro: iniciar lotería y compra de tickets	86
Ilustración 60 - Ejemplo Rinkeby descentralizado seguro: acabar lotería	87
Ilustración 61- Ejemplo Rinkeby descentralizado: nodo externo 1	87
Ilustración 62 - Ejemplo Rinkeby descentralizado: nodo externo 2	87
Ilustración 63- Ejemplo Rinkeby descentralizado: nodo externo 3	88
Ilustración 64 - Ejemplo Rinkeby descentralizado inseguro: despliegue de contrato	89
Ilustración 65 - - Ejemplo Rinkeby descentralizado inseguro: inicialización lotería y compra de tickets	89
Ilustración 66- Ejemplo Rinkeby descentralizado inseguro: finalizar lotería	90
Ilustración 67- - Ejemplo Rinkeby descentralizado inseguro:nodos externos	90
Ilustración 68 - - Ejemplo Rinkeby descentralizado inseguro: Resultados lotería	91
Ilustración 69 - Funcionamiento Chainlink VRF	93
Ilustración 70 - Ejemplo Rinkeby Chainlink VRF: desplegar contrato	95
Ilustración 71 - Ejemplo Rinkeby Chainlink VRF: nueva loterías y entradas de jugadores	96
Ilustración 72 - Ejemplo Rinkeby Chainlink VRF: Financiar contrato con LINK	97
Ilustración 73 - Ejemplo Rinkeby Chainlink VRF: finalizar lotería	97
Ilustración 74 - Ejemplo Rinkeby Chainlink VRF: Etherscan (I)	98
Ilustración 75 - Ejemplo Rinkeby Chainlink VRF: Etherscan (II)	98
Ilustración 76 - Ejemplo Rinkeby Chainlink VRF: Etherscan (III)	99
Ilustración 77 - ODS	108
Ilustración 78 - Botón para descargar extensión de Metamask para Google Chrome	110
Ilustración 79 - Añadir extensión de Metamask desde Chrome Web Store	110
Ilustración 80 - Primera pantalla de la extensión de Metamask	111
Ilustración 81 - Metamask: importar o crear cartera	111
Ilustración 82- Metamask: crear contraseña	111
Ilustración 83 - Metamask: frase secreta de recuperación	112

Ilustración 84 - VSC opciones de instalación.....	113
Ilustración 85 - VSC: pantalla de inicio.....	113
Ilustración 86 - Python: Descarga para la instalación.....	114

## *Índice de tablas*

Tabla 1 - Comparativa distintos modelos implementados .....	12
Tabla 2 - Comparativa distintos modelos implementados .....	18
Tabla 3 – Salario anual según convenio.....	34
Tabla 4 - Cálculo del coste directo para la empresa por perfil.....	35
Tabla 5 - Coste real por empleado para la empresa.....	36
Tabla 6 - Coste total para la empresa por perfil .....	36
Tabla 7 - Beneficios estimados para la empresa por perfil .....	37
Tabla 8 - Precio por hora por perfil .....	37
Tabla 9 - Número de horas por perfil y tarea .....	38
Tabla 10 - Coste total del proyecto por perfil.....	39
Tabla 11 - Nodos off-chain y números aleatorios generados .....	83
Tabla 12 - Comparativa distintos modelos implementados .....	103

# Capítulo 1. INTRODUCCIÓN

## *1.1 BLOCKCHAIN*

En primer lugar, para entender este proyecto, debemos saber qué es el Blockchain. El término Blockchain, cuya traducción sería “cadena de bloques”, hace referencia a un tipo de base de datos distribuida que es compartida entre los nodos de una red informática.

La principal diferencia con una base de datos tradicional es la forma en la que se almacena la información. En una base de datos tradicional, la información generalmente se almacena en tablas. En Blockchain, los datos son almacenados en bloques que están vinculados mediante criptografía. Los bloques tienen una capacidad de almacenamiento y una vez esta capacidad se llena, se cierra el bloque y se vincula al bloque anterior formando una cadena.

Una de las mayores utilidades y ventajas del Blockchain es la capacidad de mantener una red segura y descentralizada para almacenar transacciones. Esto se debe a que garantiza la fidelidad y seguridad de los datos almacenados sin necesidad de una entidad ajena de confianza que lo garantice. Algunos de los ejemplos más famosos de Blockchain son Bitcoin y Ethereum.

Además, otra de las ventajas del Blockchain es que al ser un sistema descentralizado se dificulta la manipulación maliciosa de los datos. Esto se debe a que, al existir redundancia al tener los datos almacenados en varios nodos, en el caso de que alguien quisiera manipular maliciosamente el registro de transacciones de un nodo, el resto de los nodos no se verían afectados y podrían señalar fácilmente al nodo que contiene la información incorrecta. Este sistema ayuda a generar un orden de los eventos exacto y transparente, evitando que un nodo individual dentro de la red pueda modificar la información contenida dentro de un bloque.

Para que Blockchain sea seguro, los bloques se añaden al final de la cadena de forma lineal y cronológica. Cada bloque genera su propio hash mediante funciones matemáticas (como SHA256 o Keccak256) que transforman la información contenida en el bloque a una cadena de números y letras, el hash del bloque anterior y una marca de tiempo. De este modo, si la información contenida en un bloque es cambiada, el código hash cambia también. En el caso de querer alterar información en la cadena, debe haber un consenso en la red habiendo al menos un 51% de los nodos que acepten el cambio.

Algunos mecanismos de consenso empleados en Blockchain son Proof of Work (PoW) y Proof of Stake (PoS). Estos mecanismos aseguran que haya un consenso, aunque no exista un único nodo a cargo.

Podemos resumir las principales ventajas de Blockchain en los siguientes puntos:

- ❖ **Descentralización:** No existe un único punto de autoridad.
- ❖ **Seguridad:** Comprobando el último hash de la cadena y mediante una votación se puede detectar y aceptar/descartar cambios en la cadena.
- ❖ **Resiliencia:** Mientras haya un único nodo activo, la red Blockchain sigue activa.
- ❖ **Transparencia y eficiencia** a la hora de realizar transacciones.

Por otra parte, existen algunos inconvenientes asociados al Blockchain como pueden ser el significativo uso de energía a la hora de minar los bloques, la escasa y confusa regulación que varía según la jurisdicción o el bajo ratio de transacciones por segundo. Aun así, Blockchain es una tecnología por la que cada vez se apuesta más y que está en continuo desarrollo para minimizar dichos inconvenientes.

## ***1.2 SMART CONTRACT***

Una vez conocemos el concepto de Blockchain, debemos conocer qué es un Smart Contract. Un Smart Contract o contrato inteligente es el código de un programa que puede ser implementado en un Blockchain para facilitar, validar o verificar un acuerdo contractual.

Los Smart Contract se ejecutan cuando se cumplen una serie de condiciones previamente acordadas por los usuarios y escritas en código. Es decir, los Smart Contract pueden ser utilizados para automatizar la ejecución de un acuerdo cuando se cumplan unas determinadas condiciones.

La mayor ventaja de los contratos inteligentes es que no se necesita una tercera persona o mediador para el cumplimiento de un acuerdo. El propio Smart Contract se encarga de ejecutar el acuerdo en el caso de que se cumplan las condiciones acordadas por los usuarios.

Las ventajas de emplear Smart Contracts son:

- ❖ Rapidez, eficiencia y precisión: Cuando se cumplen las condiciones el contrato es ejecutado de forma automática.
- ❖ Transparencia y confianza: Todas las transacciones son compartidas entre los participantes y no existe la necesidad de una tercera persona involucrada.
- ❖ Seguridad: Las transacciones realizadas en Blockchain están encriptadas y conectadas a los bloques anteriores. Esto dificulta mucho los posibles ataques.
- ❖ Ahorro económico: Al no necesitar de intermediarios o terceras personas involucradas para el cumplimiento de un contrato, esto supone el ahorro de tasas o retrasos temporales asociados con estos.

El principal problema de los Smart Contracts es que al ser ejecutados en una red Blockchain, que es una red distribuida, cada nodo de la red tiene que ser capaz de ejecutar el contrato y obtener los mismos resultados para unos determinados datos de entrada. Esto es un problema a la hora de introducir datos de fuera de la cadena (off-chain), ya que si, por ejemplo, todos los nodos de una cadena realizasen una llamada a una determinada API, el resultado podría variar si las llamadas se producen en diferentes instantes, la API es hackeada u otros motivos.

Existen contratos inteligentes que combinan código ejecutándose en un Blockchain (on-chain) con datos y computaciones que se producen en el exterior de la cadena (off-chain). Este tipo de contratos inteligentes son los Hybrid Smart Contract.

Los Hybrid Smart Contract emplean Oráculos. Un Oráculo es cualquier entidad o dispositivo que conecta un Blockchain cuyos resultados tienen que ser determinísticos con datos de fuera de la cadena. Los Oráculos se encargan de introducir los datos del mundo exterior mediante una transacción externa asegurando que todo el Blockchain tiene la misma información para verificar una transacción.

En este proyecto se pretende programar y comparar diferentes formas de introducir datos externos (off-chain) a los SmartContracts de un Blockchain.

## Capítulo 2. DESCRIPCIÓN DE LAS TECNOLOGÍAS

En este capítulo se van a describir las tecnologías, y herramientas específicas que han sido empleadas para el proyecto.

### 2.1 *ETHEREUM*



*Ilustración 7 - Logo de Ethereum*

Ethereum es una red descentralizada que nace en 2015 y una de las más populares hoy en día. Esta red está basada en tecnología Blockchain y surge con la finalidad de crear aplicaciones descentralizadas, es decir, aplicaciones que no dependen de un organismo regulador o gobierno.

La red de Ethereum permite validar las transacciones que se realizan en la cadena mediante los SmartContract de una forma ágil, descentralizada y segura. Los Smart Contract son programas que se almacenan en la cadena de bloques y que se ejecutan automáticamente cuando se cumplen unas determinadas condiciones. Los Smart Contract o contratos inteligentes fue algo pionero y uno de los proyectos más ambiciosos del mundo cripto y que nace gracias a Ethereum.

El Ether o ETH es su criptomoneda nativa y puede ser usada para pagar por las transacciones que se realizan dentro de la red. Esto último se conoce como gas, y es que, cada transacción tiene un coste de gas asociado.



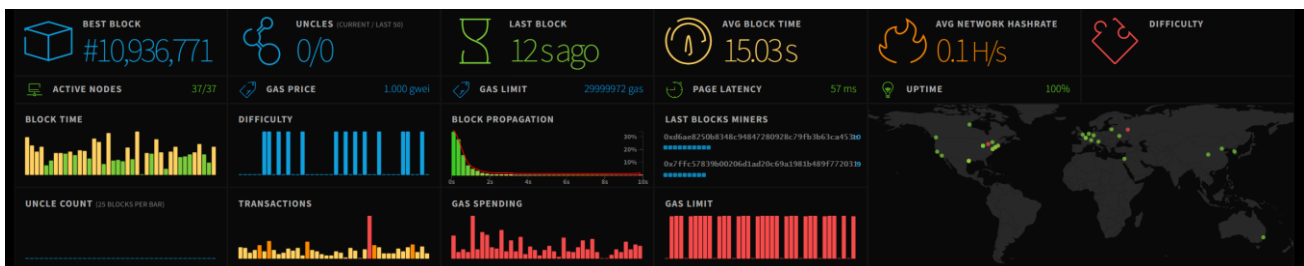
En este proyecto se emplea la red de Ethereum para el desarrollo de Smart Contracts. Por otra parte, se emplearán las redes de pruebas disponibles para el testeo del proyecto como se detalla más adelante.

## 2.2 TESTNET (RINKEBY)

Las Testnets son colecciones de nodos que se utilizan para probar el protocolo de Ethereum. Podemos utilizar testnets para desplegar y probar Smart Contracts sin tener que escribirlos y desplegarlos en la Mainnet, que es la red principal de Ethereum, ya que esto es un proceso caro y que emplea dinero real. Las Testnets proporcionan criptomonedas gratuitas sin valor que se puede utilizar para pagar las tasas de gas necesarias en las transacciones.

Rinkeby es una testnet de Ethereum. Es una bifurcación de la Mainnet de Ethereum y fue construida por Fundación Ethereum (EF) en abril de 2017. Actualmente, la red de pruebas Rinkeby es mantenida por el equipo de desarrolladores de Geth. Es la testnet que utilizaremos en este proyecto para realizar pruebas y comprobar cómo funcionaría nuestro código en la red real de Ethereum o Mainnet.

A continuación, se muestra una imagen (Ilustración 4) del estado actual de la red de pruebas Rinkeby con sus estadísticas. Esta información se puede consultar en el siguiente enlace <https://www.rinkeby.io/#stats>.



*Ilustración 8- Estadísticas de Rinkeby*

## 2.3 *METAMASK*



*Ilustración 9 – Logo de Metamask*

Metamask es una wallet de Ethereum que tiene un plugin o extensión para navegadores web. Esto nos permitirá interactuar fácilmente con los SmartContracts mediante Web3, permitiéndonos realizar transacciones a cualquier dirección de Ethereum. También nos permite interactuar con direcciones de las redes de pruebas de Ethereum o Testnets.

## 2.4 *VISUAL STUDIO CODE*



*Ilustración 10 - Icono de Visual Studio Code*

Para este proyecto, principalmente se empleará Visual Studio Code. Se trata de un editor de código fuente desarrollado por Microsoft y que ofrece soporte para Windows, Linux, macOS y Web. Además, permite instalar paquetes para facilitar la edición del código resaltando la sintaxis. Por otra parte, tiene soporte para la depuración de código. Por último, al permitir el control de versiones de los archivos mediante Git, podremos alojar nuestros repositorios en GitHub.

## 2.5 *BROWNIE*



*Ilustración 11 - Logo de Brownie*

Brownie es un framework para desarrollar Smart Contract de Ethereum y que permite poder testear y comprobar su correcto funcionamiento. Brownie está basado en Python y es una plataforma para desarrollar y testear los contratos centrada en Ethereum Virtual Machine (EVM), que es el entorno donde todos los Smart Contract y cuentas de Ethereum existen. La EVM define las reglas y normas para los estados validos de la cadena de bloques de Ethereum.

Brownie ayuda a manejar fácilmente el despliegue, la interacción la depuración y las pruebas de los contratos inteligentes. Además, brownie nos permite emplear redes privadas locales como Ganache o redes de pruebas como Rinkeby.

Brownie es el framework que se emplea en este proyecto para facilitar el desarrollo de los Smart Contract y realizar distintas pruebas para comprobar el correcto funcionamiento de estos.

## 2.6 *WEB3.PY*

Web3.py es una biblioteca de Python para interactuar con los contratos inteligentes. Algunas de las funciones que nos ofrece Web3.py es poder enviar transacciones, interactuar con contratos inteligentes o leer datos de bloques, además de otra variedad de casos de uso.

En este proyecto, se emplea Web3.py para poder crear scripts de Python que interactúen con los Smart Contract pudiendo de este modo realizar transacciones y testear el correcto funcionamiento de los Smart Contract.

## 2.7 *ETHERSCAN*

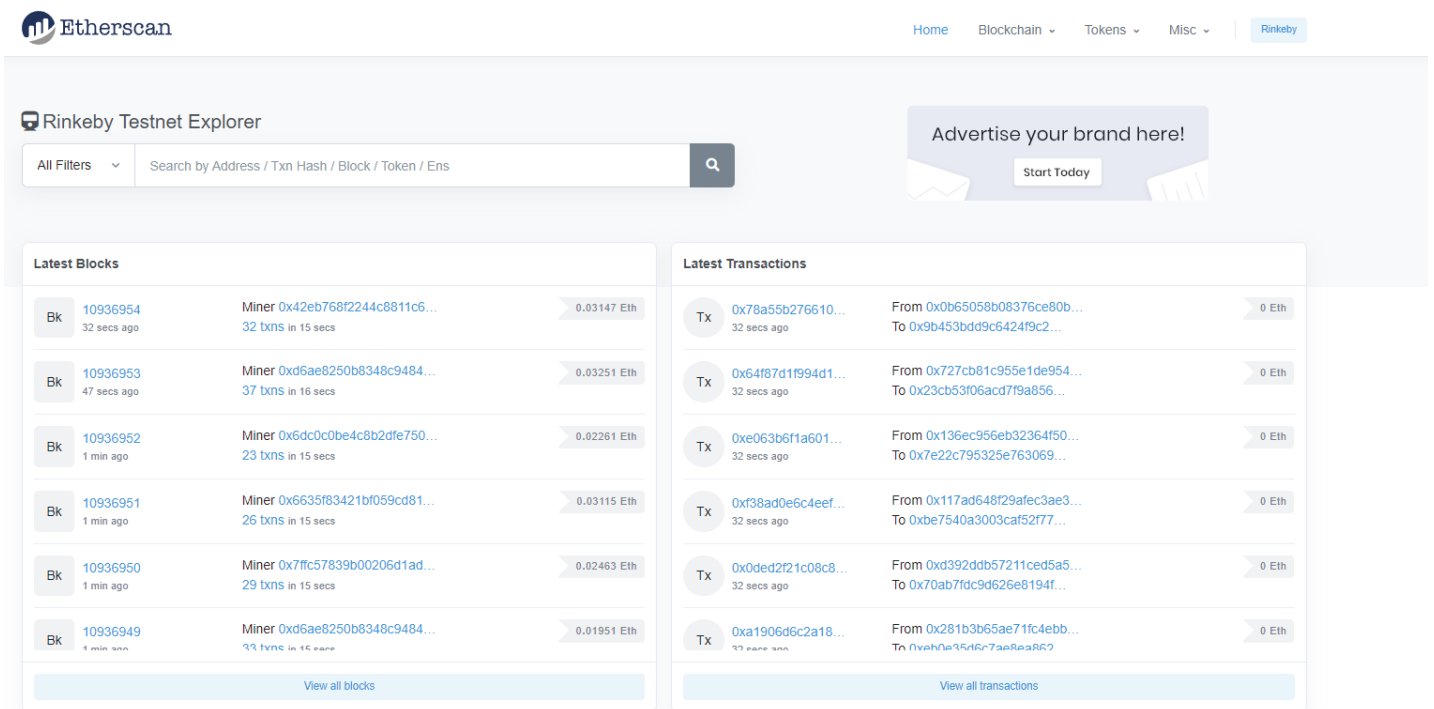


*Ilustración 12 - Logo de Etherscan*

Etherscan es un explorador de bloques que nos ayuda a acceder a los detalles de cualquier transacción realizada en cualquier blockchain de Ethereum. Muestra transacciones pendientes o confirmadas. Este sitio web nos permite buscar transacciones, bloques, direcciones de cuentas o SmartContract, y otros datos de la cadena de bloques.

En nuestro proyecto usaremos Etherscan para comprobar el resultado de las transacciones realizadas y poder testear el correcto funcionamiento de nuestros Smart Contract.

A continuación, se muestra la página principal de Etherscan para la red de pruebas de Rinkeby (Ilustración 13) donde aparece una barra donde se puede buscar por dirección, hash de la transacción, número de bloque u otros filtros. Además, nos muestra los últimos bloques minados y las últimas transacciones realizadas en esta red.



**Etherscan** Home Blockchain Tokens Misc Rinkeby

Rinkeby Testnet Explorer

All Filters Search by Address / Txn Hash / Block / Token / Ens

Advertise your brand here! Start Today

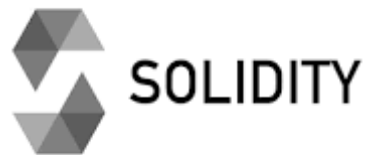
Latest Blocks			
Bk	10936954 32 secs ago	Miner 0x42eb768f2244c8811c6... 32 txns in 15 secs	0.03147 Eth
Bk	10936953 47 secs ago	Miner 0xd6ae8250b8348c9484... 37 txns in 16 secs	0.03251 Eth
Bk	10936952 1 min ago	Miner 0x6dc0c0be4c8b2dfe750... 23 txns in 15 secs	0.02281 Eth
Bk	10936951 1 min ago	Miner 0x6635f83421b0f59cd81... 26 txns in 15 secs	0.03115 Eth
Bk	10936950 1 min ago	Miner 0x7ffc57839b00206d1ad... 29 txns in 15 secs	0.02463 Eth
Bk	10936949 1 min ago	Miner 0xd6ae8250b8348c9484... 33 txns in 15 secs	0.01951 Eth

Latest Transactions			
Tx	0x78a55b276610... 32 secs ago	From 0x0b65058b08376ce80b... To 0x9b453bdd9c6424f9c2...	0 Eth
Tx	0x64f87d1f994d1... 32 secs ago	From 0x727cb81c955e1de954... To 0x23cb53f06acd7f9a856...	0 Eth
Tx	0xe063b6f1a601... 32 secs ago	From 0x136ec956eb32364f50... To 0x7e22c795325e763069...	0 Eth
Tx	0xf38ad0e6c4eef... 32 secs ago	From 0x117ad648f29afec3ae3... To 0xbe7540a3003caf52f77...	0 Eth
Tx	0x0ded2f21c08c8... 32 secs ago	From 0xd392db57211ced5a5... To 0x70ab7fcd9d626e8194f...	0 Eth
Tx	0xa1906d6c2a18... 32 secs ago	From 0x281b3b65ae71fc4ebb... To 0x6b0e3546c7ae8a86f...	0 Eth

View all blocks View all transactions

*Ilustración 13 - Página principal de EtherScan (Rinkeby)*

## 2.8 *SOLIDITY*



*Ilustración 14 - Icono de Solidity*

Solidity es un lenguaje de alto nivel orientado a la programación de contratos inteligentes o Smart Contract. Es un lenguaje de programación orientado a objetos y se emplea para implementar Smart Contract en varias plataformas blockchain entre las que destaca Ethereum.

## 2.9 *GANACHE*



*Ilustración 15 - Logo de Ganache*

Ganache es un blockchain local y personal para Ethereum que se ejecuta en tu escritorio. Nos permite desarrollar, desplegar y probar en un entorno seguro y determinista.

El problema de probar directamente en una Testnet o red de pruebas es que tarda en procesar las transacciones, ya que se trata de una simulación de la red

principal. Utilizando Ganache, podemos probar rápidamente nuestros contratos inteligentes ya que nos permite desplegar una red local de Ethereum donde realizar nuestras pruebas.

Por otra parte, Ganache nos ofrece dos versiones diferentes. La primera, Ganache-cli se trata de una versión en línea de comandos y que Brownie despliega automáticamente. A continuación, se muestra un ejemplo (Ilustración 16). Como podemos observar, se crean 10 cuentas para esta red local con un saldo de 100 ETH y sus respectivas claves privadas. Por otra parte, se establece el precio del gas y el límite de gas dispuesto a gastar en las transacciones.

```
C:\Users\Gonzalo>ganache-cli
Ganache CLI v6.12.2 (ganache-core: 2.13.2)

Available Accounts
=====
(0) 0xbe6DF06B603bb3aA38529afa415231B878744034 (100 ETH)
(1) 0x7F35951B5DE94C23A263E996D77b3A0770F8dd66 (100 ETH)
(2) 0x64d408bC114Edb2A30034a7355a74c4c857bCacA (100 ETH)
(3) 0x1c75F1e8aDEBA1a8Dfa86E531A13626B8FFA8359 (100 ETH)
(4) 0xB8f700505E4F0644F02B3dd3D6787AD916a5D0Bb (100 ETH)
(5) 0x42A9d104d011948a892295b7aA29F8E3b2b7E0Cf (100 ETH)
(6) 0x4A391B3aCDae8a7524851c8F06b3120894B05AE0 (100 ETH)
(7) 0x81ef631F4261ad7467be6D06019891B0FA150A7E (100 ETH)
(8) 0x635baD5651C19da0abe8ec4d7831D61a86A2373a (100 ETH)
(9) 0x9aaB299FCb1110D93E108E6B89c3e8c1aB3968Bd (100 ETH)

Private Keys
=====
(0) 0x92c5c627c3d50475119aa5cc68e23958812f0fa50c826627cc21280eb822228d
(1) 0xa55dc617d73b0cd24b6996a53598f92e259e56910e6c6b77ea3d2432dab21f45
(2) 0xe8f28784df76d388e22393863a8cbaf28fbcd5e663219768a111516ccafdd65a
(3) 0xfbc1f9929f733782708105b536ec687e9f0aefde53c0d49a3adbbb41ee74c42
(4) 0xc01e6054f8cf71604992003649a35d0ba2e5e344dca8c35c18451e037ea09c4c
(5) 0x012f48546281254708ae2cd5ee9d2a1d76e6acf459613bea9351ca90fedaf25d
(6) 0xc6e199d52f3ff365ee7a7881474467595d3ca2c6ccab6d74130483a1b92e83bd
(7) 0x7f750cff1f64a390e27ebeae60ad26d22639ce6509857d371305587f269d9703
(8) 0xafdf5d25d3b0b97dff7b0108fa14b093de3dd2c70e911a65bc754044be641be
(9) 0xfb53e1aec4e66e1de1764f7f9a7509b8d689597c11b9e7fd14e5411c7ab77c22

HD Wallet
=====
Mnemonic:      limit change ignore lesson cherry toast mansion combine open drink dutch wealth
Base HD Path:  m/44'/60'/0'/0/{account_index}

Gas Price
=====
20000000000

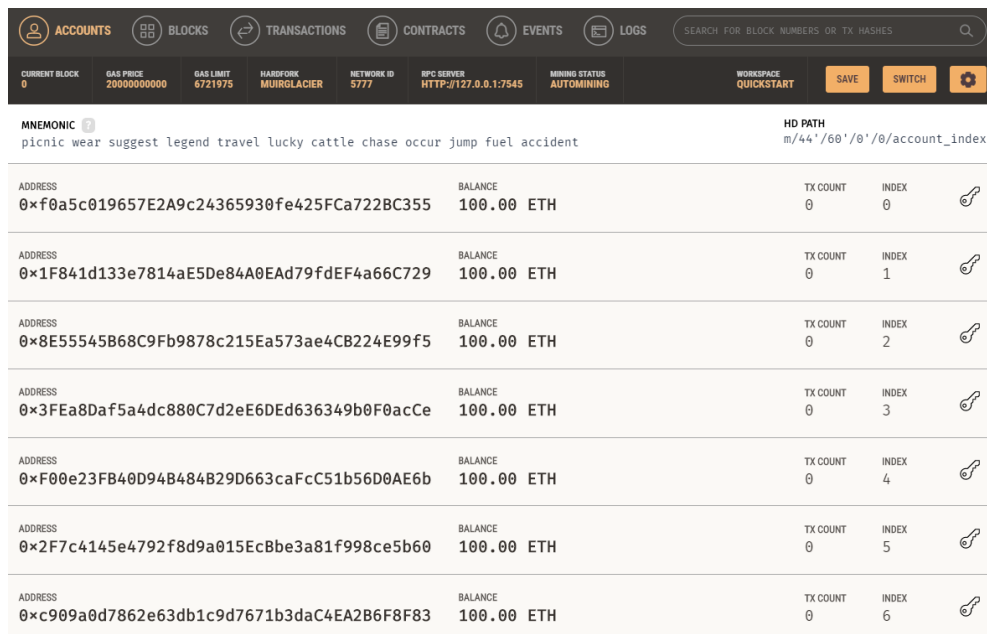
Gas Limit
=====
6721975

Call Gas Limit
=====
9007199254740991

Listening on 127.0.0.1:8545
```

*Ilustración 16- Ejemplo de ganache-cli*

Por otra parte, la segunda versión es Ganache UI. Esta versión cuenta con una interfaz gráfica y es otra forma de crear una red local mediante Ganache. A continuación, se muestra un ejemplo (Ilustración 17).



ACCOUNTS	BLOCKS	TRANSACTIONS	CONTRACTS	EVENTS	LOGS					
CURRENT BLOCK 0	GAS PRICE 20000000000	GAS LIMIT 6721973	HARDFORK MUIRGLACIER	NETWORK ID 5777	RPC SERVER HTTP://127.0.0.1:7545	MINING STATUS AUTOMINING	WORKSPACE QUICKSTART	SAVE	SWITCH	⚙️
MNEMONIC picnic wear suggest legend travel lucky cattle chase occur jump fuel accident					HD PATH m/44'/60'/0'/0/account_index					
ADDRESS 0xf0a5c019657E2A9c24365930fe425FCa722BC355	BALANCE 100.00 ETH	TX COUNT 0	INDEX 0							
ADDRESS 0x1F841d133e7814aE5De84A0EAd79fdEF4a66C729	BALANCE 100.00 ETH	TX COUNT 0	INDEX 1							
ADDRESS 0x8E55545B68C9Fb9878c215Ea573ae4CB224E99f5	BALANCE 100.00 ETH	TX COUNT 0	INDEX 2							
ADDRESS 0x3FEa8Daf5a4dc880C7d2eE6DEd636349b0F0acCe	BALANCE 100.00 ETH	TX COUNT 0	INDEX 3							
ADDRESS 0xF00e23FB40D94B484B29D663caFcC51b56D0AE6b	BALANCE 100.00 ETH	TX COUNT 0	INDEX 4							
ADDRESS 0x2F7c4145e4792f8d9a015EcBbe3a81f998ce5b60	BALANCE 100.00 ETH	TX COUNT 0	INDEX 5							
ADDRESS 0xc909a0d7862e63db1c9d7671b3daC4EA2B6F8F83	BALANCE 100.00 ETH	TX COUNT 0	INDEX 6							

*Ilustración 17 - Ejemplo de Ganache UI*

## 2.10 NODE.JS



*Ilustración 18 - Logo de Node.js*

Se trata de un entorno open source multiplataforma que permite ejecutar el código JavaScript fuera del navegador. Este entorno se orienta a eventos asíncronos y permite



construir aplicaciones red que puedan realizar muchas conexiones de manera simultánea sin tener que ejecutar el código línea a línea ni abrir múltiples procesos.

Ya que Ganache está escrito en JavaScript y está distribuido como un paquete de Node.js via npm, será necesario tener instalada una versión superior a v8 de Node.js.

## ***2.11 INFURA***



*Ilustración 19 - Logo de Infura*

Infura es un conjunto de herramientas para que una aplicación pueda conectarse a una red Ethereum de forma instantánea. Infura evita que una aplicación necesite ejecutar una complicada infraestructura para conectarse a una red Ethereum.

Los Smart Contract o las aplicaciones que se almacenan en una blockchain necesitan conexiones a través de redes peer-to-peer que pueden ser lentas y requieren de tiempos de inicialización elevados. Por otra parte, almacenar toda la cadena de bloques de Ethereum puede ser caro y requiere de una infraestructura que necesita ser escalada a lo largo del tiempo.

Infura resuelve estos problemas y proporciona la infraestructura y herramientas necesarias para que conectar nuestro Smart Contract o aplicación a una red Ethereum sea algo rápido y rentable.

## ***2.12 SPRING BOOT***



*Ilustración 20- Logo de Spring Boot*

Spring Boot es una infraestructura ligera que nos permite crear aplicaciones basadas en Spring de forma sencilla y rápida que sean fáciles de configurar. Spring Boot nos ayuda a centrarnos en el desarrollo y olvidarnos de la arquitectura.

En este proyecto se ha usado Spring Boot para crear una aplicación muy sencilla con varios endpoints que nos servirán para generar datos o para almacenar y consultar las direcciones de los últimos contratos que han sido desplegados en la cadena de bloques.

## Capítulo 3. ESTADO DE LA CUESTIÓN

Cuando analizamos el estado del arte, primero necesitamos entender que es un oráculo y la importancia que tienen estos a la hora de introducir datos desde el mundo real a un blockchain.

Un oráculo es una entidad o dispositivo que conecta una blockchain que tiene un carácter determinístico con datos que se encuentran fuera de la cadena.

Dado que una blockchain tiene una naturaleza distribuida, cada nodo de la red tiene que ser capaz de obtener el mismo resultado para una entrada. En caso contrario, cuando un nodo tratase de validar una transacción realizada por otro nodo, este terminaría con otro resultado.

Los blockchain como Ethereum están pensadas para ser determinísticos, esto significa que, si se reproducen todas las transacciones, se terminaría en el mismo estado de forma correcta. Por ello, si se realizasen llamadas a una API o alguna otra fuente de datos no determinística desde la blockchain, existe una probabilidad elevada de que el valor del dato haya cambiado, la fuente haya sido hackeada o incluso haya dejado de funcionar, por lo que no podríamos validar la transacción.

Es necesario encontrar alguna manera de conectar una blockchain con el mundo exterior, ya que estas necesitan de datos externos para poder desarrollar aplicaciones tales como DeFi o Smart Contract que requieran datos externos.

Como solución surgen los llamados oráculos, que como se ha mencionado antes sirven para introducir datos que se encuentran fuera de la cadena a una cadena de bloques con carácter determinístico. Los oráculos introducen los datos desde fuera de la cadena realizando una transacción. De este modo, nos aseguramos de que la blockchain tenga toda la información necesaria para poder verificarse. Podríamos decir que un oráculo actúa como un puente entre una blockchain y el mundo exterior.

El problema de los oráculos es que una cadena de bloques es un sistema descentralizado. Si introducimos datos desde un oráculo que sea un sistema centralizado, estamos perdiendo todas las ventajas de la descentralización que ofrece blockchain. Por ejemplo, si únicamente existe un nodo fuera de la cadena introduciendo datos a un Smart Contract, este nodo tiene poder sobre el Smart Contract haciendo que este pierda su propósito y no sea mejor que un contrato convencional. Y aunque se trate de un nodo confiable o con la mejor de las intenciones, dicho nodo puede ser atacado o dejar de funcionar.

Es decir, el problema de los oráculos hacer referencia a que las blockchain necesitan alguna forma de obtener datos desde el exterior ya que no pueden hacerlo por sí mismas, pero si el mecanismo para introducir dichos datos es un sistema centralizado entonces se anula totalmente las ventajas que ofrecen los Smart Contract e incluso se generan grandes amenazas de seguridad.

Actualmente, la solución que se está ofreciendo se trata de oráculos descentralizados. Un oráculo descentralizado o una red de oráculos descentralizada se trata de un grupo de oráculos para blockchain independientes que se introducen datos a la blockchain. Cada nodo independiente u oráculo en una red de oráculos descentralizada obtiene los datos de forma independiente de la fuente y se encarga de introducirlos en la cadena. Los datos son agregados dentro de la cadena para obtener un único valor determinístico.

Chainlink es un proyecto de oráculos descentralizados que se ha convertido en líder para la interconexión del mundo real con Smart Contracts, Dapps y el ecosistema DeFi. Chainlink se ejecuta sobre la red Ethereum. Esta red de oráculos descentralizados permite alcanzar a los Smart Contract su verdadero potencial, ya que, si un nodo o fuente de datos fuese hackeada, dejase de dar servicio o diese un valor erróneo, la red de Chainlink es capaz de seguir adelante porque se trata de una red descentralizada.



*Ilustración 21 - Logo de Chainlink*

Además, todo el código de Chainlink es open source y está constantemente trabajando para integrarse con más blockchain, aparte de Ethereum, para que todas puedan tener una manera segura y fiable de obtener datos de fuera de la cadena.

Aunque Chainlink se ha convertido en la plataforma líder de oráculos, existen otras plataformas que cumplen la misma función. Entre ellas se encuentran Provable, Town Crier y Witnet.

Ahora que conocemos el estado del arte, este proyecto pretende explorar las distintas opciones que existen a la hora de introducir datos desde fuera de la cadena a un Smart Contract que se encuentra en la blockchain de Ethereum. Para ello, se quiere desarrollar e implementar distintos modelos de oráculo para poder comparar y contrastar las diferentes opciones.

## Capítulo 4. DEFINICIÓN DEL TRABAJO

### 4.1 JUSTIFICACIÓN

Los Smart Contract, como ya comentábamos previamente, son programas o protocolos que facilitan, verifican o ejecutan los términos de un contrato de forma automática cuando se cumplen unas cláusulas acordadas con anterioridad por las partes interesadas. Podemos decir que los Smart Contract tienen las siguientes características:

- Operan en una red o sistema descentralizado.
- Son independientes.
- Son inmutables e irrevocables.

Al operar en cadenas de bloques cuyos registros son inmutables y que son sistemas distribuidos, los Smart Contract tienen un gran atractivo para las empresas presentando una innovadora opción para estas por las ventajas que estos ofrecen.

Aunque también tienen algunos contras, por ejemplo, que, al ser independientes, una vez son desplegados en una blockchain, estos no pueden ser controlados ni por el creador, ni por ningún sistema legal ya que estos solo obedecen al código que tienen escrito. Otro problema asociado a estos es que al ser irrevocables solo pueden ser sustituidos por un contrato completamente nuevo. Por ello, los Smart Contract deben ser testeados exhaustivamente para comprobar todos los escenarios posibles antes de ser desplegados. Un ejemplo de lo que puede llegar a pasar si los contratos no son testeados correctamente fue DAO y las pérdidas millonarias que se produjeron por tener vulnerabilidades que fueron explotadas por un hacker.

*“The DAO was a decentralized autonomous organization (DAO) that was launched in 2016 on the Ethereum blockchain. After raising \$150 million USD worth of ether (ETH) through a token sale, The DAO was hacked due to vulnerabilities in its code base. The*

*Ethereum blockchain was eventually hard forked to restore the stolen funds, but not all parties agreed with this decision, which resulted in the network splitting into two distinct blockchains: Ethereum and Ethereum Classic.” [12]*

A pesar de los riesgos, los Smart Contract son el futuro y se trata de una tecnología innovadora que abre la puerta a una infinitud de posibilidades. Por ejemplo, grandes compañías como Gartner ya predijeron que serán empleados cada vez más por las organizaciones globales y las grandes empresas.

*“Despite the risks, Gartner estimates that by 2022, ratified unbundled (that is, defined impact) smart contracts will be in use by more than 25% of global organizations.” [10]*

Debido a su atractivo para grandes empresas y organizaciones, los Smart Contract están desarrollándose y mejorando continuamente de forma muy rápida.

Otro problema asociado a los Smart Contract, como se ha comentado en el capítulo anterior es el problema de los oráculos, es decir, la manera de introducir datos desde el exterior de una blockchain a un Smart Contract que está desplegado en la cadena. Por ello, este proyecto pretende implementar y comparar distintos modelos para introducir un dato externo a un contrato que se encuentra en la cadena.

## **4.2 OBJETIVOS**

La idea principal de este proyecto es implementar y comparar distintos métodos para introducir datos externos (off-chain) a los Smart Contract que se encuentran en un blockchain. Para ello, se han establecido los siguientes objetivos:

### **I. Documentarse e investigar sobre BlockChain y SmartContracts**

Tanto el Blockchain como los Smart Contract son tecnologías innovadoras y emergentes que han tenido un gran auge en los últimos años. Para el correcto desarrollo del proyecto, el primer objetivo debe ser asentar unas bases teóricas y adquirir los conocimientos necesarios para poder desarrollar una aplicación que funcione correctamente.

### **II. Programar SmartContract empleando un ejemplo concreto:**

En este proyecto se va a seleccionar un tema particular para programar un SmartContract. El tema concreto en este caso será una aplicación de loterías. Los SmartContracts al ser determinísticos no pueden generar de manera segura un número aleatorio, por lo tanto, se necesita que dicho dato sea aportado desde el exterior de la cadena.

### **III. Realizar un Oráculo centralizado**

El primer método y el más sencillo para obtener un dato de fuera de la cadena. Se pretende programar un Oráculo de un único nodo para obtener un número aleatorio que será introducido al SmartContract. Además, se busca demostrar las amenazas de seguridad que provoca un oráculo centralizado.

### **IV. Realizar un Oráculo Descentralizado**

El segundo método que se realizará es un oráculo descentralizado que cuente con varios nodos fuera de la cadena para obtener un número aleatorio que será introducido al SmartContract.

### **V. Emplear ChainLink VRF (Verifiable Random Function)**

Emplear Chainlink y el servicio que ofrece destinado a la generación de números aleatorios para poder introducir dicho dato al Smart Contract.



### **4.3 METODOLOGÍA**

Para poder realizar este trabajo en el plazo de tiempo establecido y asegurar que se cumplen los objetivos propuestos, se ha empleado la siguiente metodología:

En primer lugar, se ha dividido el proyecto en dos fases diferenciadas: fase de investigación y fase de desarrollo/pruebas.

La primera fase, se trata de una etapa de búsqueda de información, investigación y pruebas de distintas herramientas. Al tratarse de un proyecto basado en Blockchain y Smart Contracts, que son tecnologías con la que no estaba familiarizado antes del proyecto, es una etapa fundamental. Durante esta etapa, se ha estudiado en un primer momento el funcionamiento de la tecnología Blockchain en profundidad para asentar unas bases teóricas. Una vez se ha comprendido esa parte, se ha continuado por entender y familiarizarse con los Smart Contract a nivel práctico en la red Ethereum, realizando varias pruebas de programación para poder asentar conocimientos y empezar a hacer uso de Solidity, que es el lenguaje de programación que emplean los Smart Contract.

La segunda fase ha sido la parte de desarrollo del proyecto como tal. Podemos dividir esta fase en dos partes diferenciadas. La primera parte, ha sido elegir y programar, a modo de ejemplo y para poder realizar pruebas, un Smart Contract que tenga la necesidad de ser alimentado con datos externos. Finalmente, se ha decidido hacer una aplicación de loterías que necesita un número aleatorio generado fuera de la cadena, ya que los Smart Contract no pueden generar números aleatorios al ser programas que se ejecutan en varios nodos de forma simultánea y todos deben obtener el mismo resultado.

Por otra parte, la segunda parte de la fase de desarrollo se ha basado en implementar los distintos modelos de oráculo que se habían propuesto como objetivos del proyecto. Los modelos propuestos en los objetivos son:

- Oráculo Centralizado
- Oráculo Descentralizado

- Chainlink VRF

Para asegurar que los objetivos se cumplieran correctamente, se han realizado sesiones de seguimiento con el director cada dos o tres semanas para mostrar los avances y realizar correcciones en caso de ser necesario.

Por último, al ser un proyecto personal se han establecido “milestones” o hitos semanales para realizar un avance de forma constante y progresiva del proyecto. De este modo, el proyecto ha ido avanzando semana a semana, teniendo siempre algo de tiempo extra por si aparecían dificultades o bloqueos.

## 4.4 PLANIFICACIÓN Y ESTIMACIÓN ECONÓMICA

### 4.4.1 PLANIFICACIÓN TEMPORAL

Como se comentaba al final del apartado anterior, para asegurar que el proyecto se desarrollaba correctamente en los plazos establecidos se diseñó una planificación temporal al principio.

La planificación temporal del proyecto se ha recogido en un diagrama de Gantt, que es una herramienta cuyo objetivo es la visualización del tiempo previsto para las diferentes tareas y objetivos que abarca un proyecto, y ha servido como guía para establecer hitos y fechas clave asegurando que el proyecto se desarrollaba dentro de los plazos establecidos. A continuación, se muestra el diagrama de Gantt empleado (Ilustración 22).



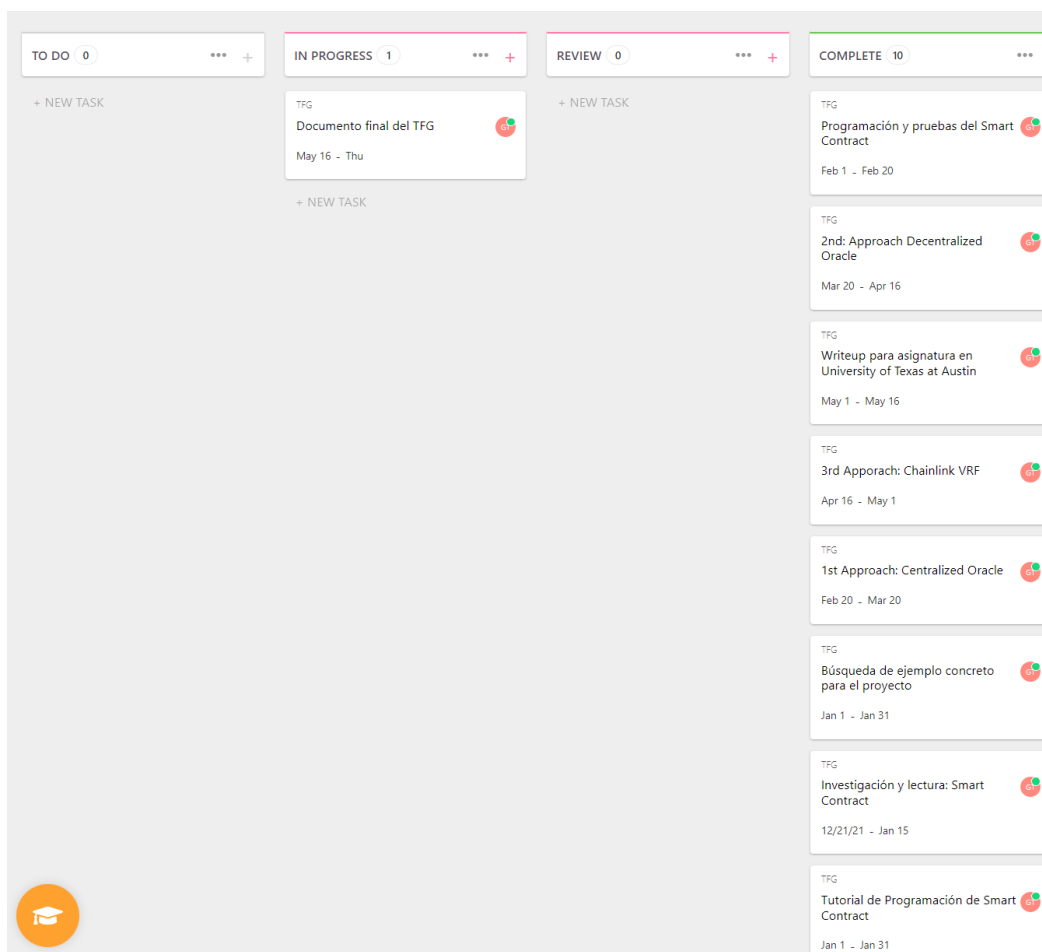
Ilustración 22 - Diagrama de Gantt del proyecto

Además, se ha contado con un panel Kanban donde se podía visualizar en todo momento las diferentes actividades para saber en que estado se encontraban. Se han establecido 4 posibles estados:

- **TO DO:** Al principio están todas las tareas del proyecto, según va avanzando el proyecto las tareas cambian al siguiente estado.
- **IN PROGRESS:** Este estado indica que es una tarea que actualmente se está desarrollando.

- **REVIEW:** Indica que una tarea ha sido finalizada y que debe ser revisada por el director del proyecto
- **COMPLETE:** Las tareas que ya han sido finalizadas y revisadas por el director del proyecto.

A continuación, se muestra una imagen del panel Kanban del proyecto (Ilustración 23):



*Ilustración 23 - Panel Kanban del proyecto*

#### 4.4.2 ESTIMACIÓN ECONÓMICA DEL PROYECTO

Para finalizar este capítulo se ha realizado una estimación económica del proyecto.

Para desarrollar este proyecto son necesarios dos personas con diferentes rangos salariales.

- Jefe de Proyectos: en este caso el tutor del proyecto.
- Consultor Junior y programador: el alumno.

En primer lugar, se va a calcular una estimación de cuanto le costaría a una empresa cada hora realmente productiva de estos trabajadores.

Para calcular sus precios-hora nos basamos en el salario bruto del XVII Convenio colectivo estatal de empresas de consultoría y estudios de mercado de la opinión pública publicado en el BOE de 6 de marzo de 2018. Este convenio es la referencia de cuanto tiene que cobrar como mínimo, de salario bruto, cada uno de los perfiles indicados.

De este convenio extraemos la siguiente tabla.

Perfil solicitado	Actividad/Grupo/Nivel (del convenio)	Salario convenio (actualizado a 2022)
Jefe de Proyectos	Consultoría, Desarrollo y Sistemas/A/I	26.567,92 €
Consultor Junior/Programador	Consultoría, Desarrollo y Sistemas/C/I	15.860,56 €

*Tabla 3 – Salario anual según convenio*

El primer paso que es importante entender es que estas personas, según convenio cobran tanto sus salarios como su antigüedad, pero la empresa además tiene que pagar por ellos a la seguridad social un 31,40% de sus sueldos, dando lugar a la siguiente tabla:

Tabla

Concepto	Jefe de Proyecto/	Consultor Junior/ Programador
Salario bruto (convenio)	26.790,31 €	15.860,56 €
Antigüedad y subida salarial	6%	1%
Importe antigüedad	1.607,42 €	158,61 €
Salario bruto más antigüedad	28.397,73 €	16.019,17 €
Cotización empresarial (31,40%)	8.916,89 €	5.030,02 €
Coste directo para la empresa por perfil	37.314,62 €	21.049,18 €

*Cálculo del coste directo para la empresa por perfil*

En esta tabla anterior, se puede ver lo que tiene que pagar la empresa al año por cada uno de estos trabajadores incluyendo su sueldo, antigüedad y lo que tiene que pagar la empresa de seguridad social por ellos.

Pero adicionalmente, tenemos que contar que no todas las 1800 horas laborables que se le pagan a un trabajador al año son productivas. Se ha considerado que el 80 % del tiempo del personal es facturable. Ya que hay otro tiempo que tienen que dedicar a formarse, labores comerciales que no se facturan directamente al cliente, descansos, etc.

Por tanto, el coste real para la empresa de 1 año de trabajo de estas personas será esta cifra dividida entre 0,8

Concepto	Jefe de Proyecto/	Consultor Junior/ Programador
Gasto de personal por perfil	46.643,27 €	26.311,48 €

*Tabla 5 - Coste real por empleado para la empresa*

Pero además hay que tener en cuenta costes otros costes como el alquiler y mantenimiento de oficinas, equipos informáticos necesarios, etc.

Para incluir estos costes, se ha procedido a utilizar los ratios sectoriales de la Central de Balances del Banco de España del año 2019 (última versión disponible) para sociedades no financieras en el sector de actividad “J62 Programación, consultoría y otras actividades relacionadas con la informática”.

Este documento, parte de una variable estadística que nos indica la relación entre lo que tiene que facturar la empresa y sus gastos de personal. A este ratio le denomina R02

$$R02 = \text{Gastos de personal} / \text{Costes totales} = 64,47 \%$$

Despejando la cifra de negocios (cifra que tiene que facturar la empresa por persona), obtenemos:

Concepto	Jefe de Proyecto/	Consultor Junior/ Programador
Costes totales	72.348,80 €	40.811,97 €

*Tabla 6 - Coste total para la empresa por perfil*

Y la empresa tiene que obtener beneficios. Esto nos lo da el ratio R03 del mismo documento

$$R03 = \text{Beneficio} / \text{Costes totales} = 17,46 \%$$

Concepto	Jefe de Proyecto/ Consultor Senior	Consultor Junior/ Programador
Beneficio	12.632,10 €	7.125,77 €

*Tabla 7 - Beneficios estimados para la empresa por perfil*

Sumamos los costes totales y el beneficio obtenemos lo que tiene que facturar la empresa por cada una de estas personas al año para obtener beneficio.

Dividiendo esta cifra entre 1800 horas laborables que según convenio tiene el año, obtenemos el precio MÍNIMO al que deberá vender la empresa estos perfiles para obtener beneficio.

Concepto	Jefe de Proyecto/ Consultor Senior	Consultor Junior/ Programador
Precio de venta anual	84.980,90 €	47.937,74 €
Precio unitario por hora	47,21 €	26,63 €

*Tabla 8 - Precio por hora por perfil*

A partir de aquí, es importante entender que el proyecto no es necesario venderlo por su precio “mínimo” sino que en función de cómo sea el mercado (si hay mucha o poca competencia de empresas dispuestas a hacer el mismo trabajo) podría venderse a un precio-hora superior (nunca inferior porque estaríamos no ganando dinero).

Utilizaremos pues estos precios-hora en lo que sigue de nuestros cálculos.



Ahora, vamos a realizar una estimación del número de horas necesarias para la realización de este proyecto:

	Consultor Programador	Junior/ Jefe de Proyecto/ Consultor Senior
Investigación y formación en Blockchain y Smart Contract	60	5
Programar Smart Contract con ejemplo concreto	50	10
Modelo de oráculo centralizado	60	10
Modelo de oráculo descentralizado	60	10
Implementación usando Chainlink VRF	50	5
Realización del documento final	100	10
<b>Horas totales</b>	<b>380</b>	<b>50</b>

*Tabla 9 - Número de horas por perfil y tarea*

La estimación de horas ha sido obtenida a partir de la planificación temporal realizada para este proyecto.

Finalmente, si multiplicamos por el precio unitario por hora mínimo de cada uno de los perfiles, podemos obtener la estimación del coste total de este proyecto.

Concepto	Jefe de Proyecto/ Consultor Senior	Consultor Junior/ Programador
Numero de horas	50	380
Precio unitario por hora	47,21 €	26,63 €
Coste total del proyecto por perfil	2.360,5 €	10.119,4

*Tabla 10 - Coste total del proyecto por perfil*

Sumando ambas cantidades, obtenemos que el coste mínimo de este proyecto es de 12.479,9€

## Capítulo 5. BLOCKCHAIN KEY CONCEPTS

### 5.1 INFORMACIÓN GENERAL

Aunque ya se ha hablado un poco en la introducción de qué es Blockchain, en este capítulo se pretende profundizar algo más en qué es una cadena de bloques y cómo funciona.

Como ya hemos comentado previamente en la introducción, Blockchain es una tecnología emergente que en los últimos años ha crecido muy rápido y que tiene potencial para revolucionar de forma drástica muchos sectores.

El nacimiento de Blockchain surge en 2008 dentro del proyecto Bitcoin de Satoshi Nakamoto. El creador del proyecto Bitcoin, presentó una moneda digital que empleaba técnicas de criptografía para gestionar la propiedad y crear un sistema seguro. En su publicación presentó un nuevo sistema de dinero electrónico peer-to-peer llamado Bitcoin y que empleaba blockchain. En el sistema de blockchain empleado por Bitcoin, se empleó un mecanismo de consenso llamado Proof of Work (PoW) para validar las transacciones y asegurar la red contra transacciones duplicadas. Además, la red de Bitcoin ofrecía transparencia a través del uso de blockchain y mediante técnicas criptográficas aseguraba la propiedad e identidad, aunque permitía la existencia de un pseudoanonimato.



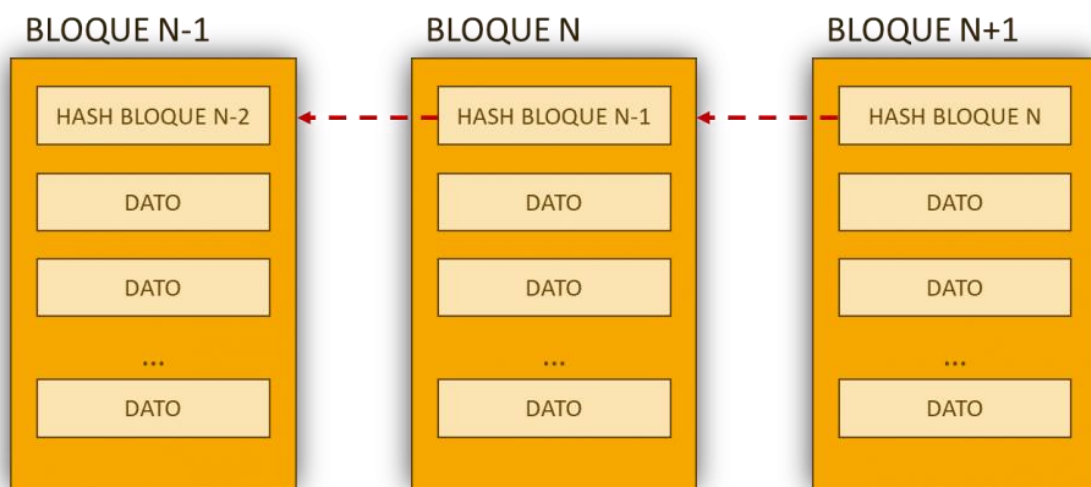
*Ilustración 24 - Logo de Bitcoin*

En muchas ocasiones se confunde el concepto de Blockchain con Bitcoin debido a que es la tecnología por debajo de este proyecto de criptomoneda. Blockchain es una base de datos en la que únicamente se pueden añadir cosas y que crece cronológicamente, empleando marcas de tiempo. Además, emplea medidas criptográficas para proteger las transacciones almacenadas para que estas no puedan ser manipuladas, es decir, los datos no pueden ser borrados o alterados.

Este tipo de base de datos recoge todas las transacciones validadas en unos elementos llamados bloques. Cuando estos bloques son validados por el mecanismo de consenso de la red, son añadidos a una cadena de bloques, en la que todos los bloques están enlazados criptográficamente mediante hashes para asegurar la integridad de los datos.

Si se cambia un único bit de uno de los bloques que pertenecen a la cadena, los enlaces criptográficos mediante hashes colapsan, haciendo que la cadena sea rechazada por la red.

A continuación, se muestra una imagen de la estructura de la blockchain de Bitcoin (Ilustración 25).



*Ilustración 25 - Estructura de bloques de una blockchain*

## 5.2 TIPOS DE BLOCKCHAIN

La siguiente ilustración (Ilustración 26), sacada del libro *Badr, B., Horrocks, R., & Wu, X. (2018). Blockchain By Example [6]* recoge los distintos tipos de blockchain que podemos encontrar.

La distinción entre un tipo u otro de blockchain se basa principalmente en dos criterios: el tipo de red y el modelo de control de acceso a dicha red. Generalmente las redes privadas de blockchain suelen ser empleadas para construir entornos donde realizar pruebas y evitar los costes asociados a las transacciones. Por otra parte, las redes del tipo Blockchain as a service (BaaS) suelen estar destinadas para facilitar desarrollar y desplegar aplicaciones o contratos y poder tener una gran escalabilidad.

Model	characteristics	Technologies	Strengths
Public blockchain	The general public can join the network and write (under consensus protocol) and read data. This model is a true representation of the original blockchain used in the cryptocurrencies.	Bitcoin, Zcash, Ethereum, Litecoin, NXT, etc..	Full-Decentralization High Security Censorship-resistant Low-trust Anonymity Transparency
Private or internal blockchain	A network under the governance of one organization, which defines the access rules to join its private network. Therefore, only authorized entities can read the transactions data. The organization defines nodes with high trust levels to accept the transaction.	Monax, Symbiont Assembly, Iroha, Kadena, Chain, Quorum, MultiChain.	Confidentiality Authenticated parties Privacy Faster Less expensive
Consortium or hybrid Blockchain	A partly private and permissioned blockchain network operated by known entities such as stakeholders of a given industry regrouped in a consortium or exploiting a shared platform. The network participants have control over who can join the network, and who can participate in the consensus process of the blockchain.	Hyperledger Fabric, Tendermint, Symbiont Assembly, R3 Corda, Iroha, Kadena, Chain, Quorum, MultiChain	Confidentiality Authenticated parties Privacy Faster Less expensive
Blockchain as a service (BaaS)	Cloud platform hosted by a service provider to deploy blockchain applications. The service provider manages the blockchain network while the customer defines the business logic.	Bluemix, Azure, Rubix, Stratis, AWS, SAP, Oracle.	Flexibility Scalability Complexity reduction

*Ilustración 26 - Clasificación de los tipos de blockchain*

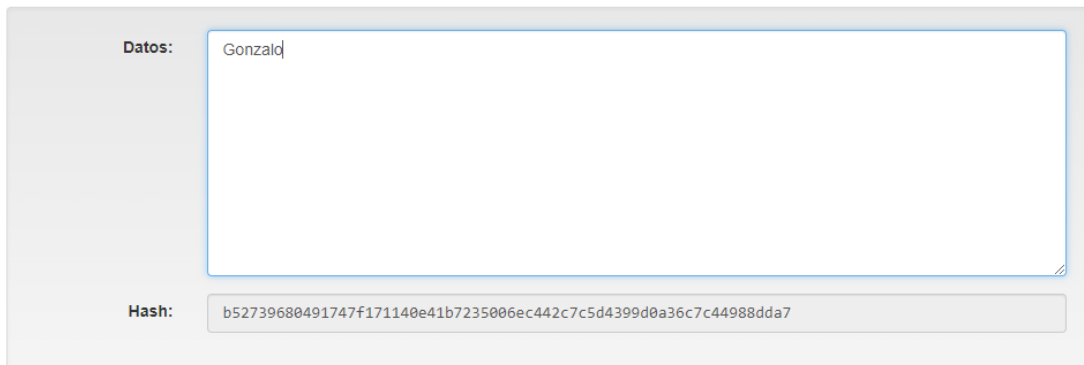
## 5.3 ¿CÓMO FUNCIONA UNA CADENA BLOCKCHAIN?

Para ilustrar cómo funciona una cadena de bloques, se va a hacer uso de una demo de blockchain creada por Anders Brownworth. [13]

### 5.3.1 HASH

En primer lugar, debemos entender cómo funcionan las funciones criptográficas de hash. Una función criptográfica de hash es un algoritmo matemático que convierte los elementos de entrada a la función a otro elemento de longitud fija. Para comprender mejor el funcionamiento de estas funciones, vamos a emplear como ejemplo el algoritmo SHA256.

#### SHA256 Hash




Datos: Gonzald

Hash: b52739680491747f171140e41b7235006ec442c7c5d4399d0a36c7c44988dda7

*Ilustración 27 - Ejemplo 1 de SHA256*

#### SHA256 Hash



Datos: Gonzalo Tamariz-Martel Sánchez

Hash: 87608569e1355f5f9c332b99e7e50766c08db30ef5d94752aa4d969662291d91

*Ilustración 28 - Ejemplo 2 de SHA256*

Como podemos ver en las dos imágenes mostradas (Ilustración 27 e Ilustración 28) al introducir una cadena de caracteres en el bloque de datos, obtenemos como salida un hash. Al variar los datos obtenemos un hash diferente, pero lo interesante de los algoritmos de hash es que independientemente de la longitud de la entrada, siempre obtenemos como salida un hash con la misma longitud. Además, cabe destacar que, para una misma entrada, siempre se obtiene el mismo hash de salida.

### **5.3.2 ALGORITMOS DE CONSENSO**

Los algoritmos de consenso sirven para que dentro de un sistema distribuido los distintos agentes puedan ponerse de acuerdo y coordinarse. Gracias a los algoritmos de consenso, se puede establecer una única verdad. En el caso de blockchain, estos algoritmos son necesarios para que todos los nodos contengan una copia idéntica de la base de datos. En caso de no existir dichos mecanismos de consenso, los nodos acabarían teniendo información contradictoria.

En blockchain, se usan algoritmos de consenso para validar todas las transacciones y los bloques que se añaden a la cadena. De este modo, se evita que alguien de forma maliciosa valide transacciones incorrectas, haga “double spending” de criptomonedas (gastar más de una vez sus monedas) y asegura que la cadena de bloques es muy difícil de atacar o manipular.

Los mecanismos de consenso más empleados para blockchain son:

- Proof of Work (PoW)
- Proof of Stake (PoS)

#### ***5.3.2.1 Proof of Work***

El algoritmo de consenso Proof of Work (PoW) fue implementado por primera vez en el proyecto Bitcoin. En PoW los validadores o mineros someten a funciones de hash los datos que se desean añadir hasta encontrar una solución concreta.

Un ejemplo sería obligar a que únicamente un hash que empiece por 0000 sea válido. Al proceso de buscar una solución que cumpla este requisito se llama minado. Los mineros buscan la solución mediante fuerza bruta probando distintas entradas y ajustando un parámetro llamado nonce.

El problema asociado a PoW es que requiere mucha energía y esto crece cuantos más mineros se unan a la red.

*"To maintain security and decentralization, Ethereum on proof-of-work consumes 73.2 TWh annually, the energy equivalent of a medium-sized country like Austria." [18]*

### **5.3.2.2 Proof of Stake**

Por otra parte, el algoritmo Proof of Stake (PoS) se propuso como alternativa a PoW. Este algoritmo elimina el concepto de mineros y, por lo tanto, resuelve el problema de consumo excesivo de energía. Además, PoS es un método que no tiene unos requisitos de hardware tan elevados como en el caso de PoW.

En PoS un validador bloquea y pone como colateral una cantidad determinada de la criptomoneda de la red. El protocolo elige un bloque y el validador de dicho bloque recibe una pequeña comisión por la transacción. En caso de que alguien intente validar transacciones no válidas perderá parte o la totalidad del colateral que había puesto e incluso puede llegar a ser expulsado de la red.

Actualmente, solo pequeñas blockchain están usando este sistema, pero Ethereum tiene un proyecto de implementar este algoritmo en Ethereum 2.0

Podríamos decir que en PoW, los mineros demuestran que están poniendo capital en riesgo debido al coste energético que tiene, mientras que en PoS los validadores realmente ponen capital en riesgo.



### **5.3.2.3 Conclusiones**

Finalmente, cabe destacar que técnicamente PoW o PoS no son por sí mismos protocolos de consenso. Ambos protocolos aseguran Sybil Resistance (resistencia ante ataques en los que un usuario o grupo pretende ser muchos usuarios) y conocer quién ha sido el autor del bloque. En el caso de PoW lo asegura obligando a los usuarios a gastar mucha energía y en PoS mediante la participación con un colateral de capital.

Por otra parte, se necesita un mecanismo para saber cuál es la cadena correcta. En el caso de Ethereum y Bitcoin se sigue la regla de la cadena más larga. En el caso de PoW la cadena más larga es aquella con mayor dificultad acumulada.

La combinación de un mecanismo que asegure Sybil Resistance y una regla de selección de cadena son realmente un protocolo de consenso. Cabe destacar, que en el caso de PoW y la regla de la cadena más larga este protocolo de consenso es conocido como “Nakamoto Consensus”

### **5.3.3 BLOQUES**

Ahora que entendemos cómo funcionan los algoritmos de hash, podemos entender cómo funcionan los bloques de un blockchain.

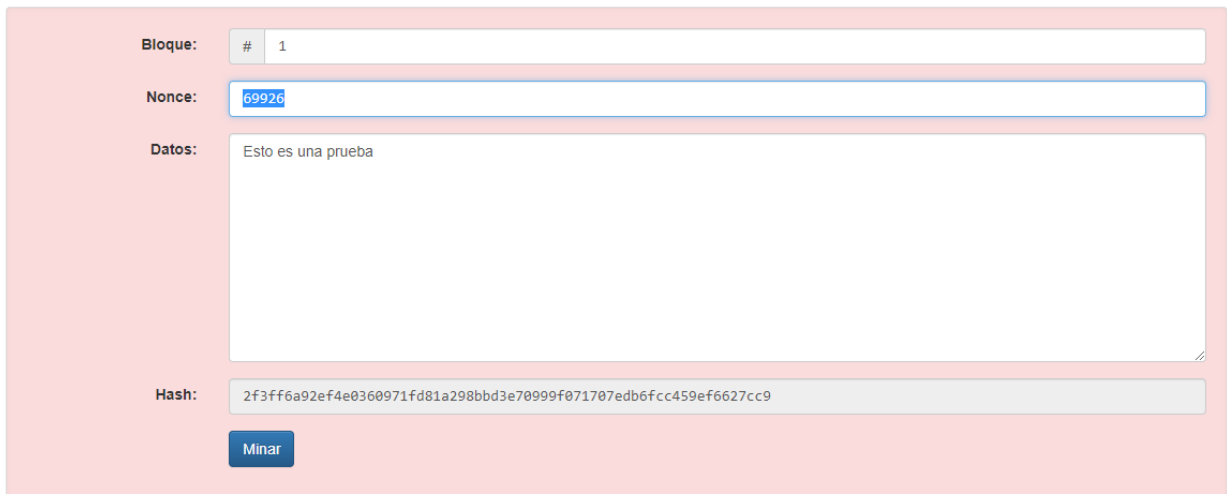
Un bloque es una unidad que contiene información sobre distintas transacciones confirmadas. Cada bloque almacenado en la cadena tiene un hash que funciona como identificador único, una cabecera y un cuerpo. El hash es la huella digital de los datos que contiene el bloque y es único para cada bloque. La cabecera está compuesta por información acerca de la creación del bloque (marca de tiempo, nonce, dificultad del bloque, etc) y tiene una referencia al bloque anterior en la cadena. Por otra parte, el cuerpo contiene la información sobre las transacciones confirmadas.

A continuación, se describen algunos de los elementos que contiene un bloque:

- Hash bloque previo: Sirve para vincular los bloques secuencialmente en la cadena.
- Marca de tiempo: Permite identificar en qué momento o instante fue creado el bloque
- Dificultad del bloque: Indica lo difícil o lento que ha sido el proceso de minado del bloque.
- Nonce: Valor encontrado por fuerza bruta durante el proceso de minado.

Para mostrar cómo funciona un bloque, se muestra en las siguientes imágenes un bloque con una serie de datos antes y después de ser minado (Ilustración 29 e Ilustración 30). En este caso, el proceso de minado consiste en calcular un nonce tal que el hash resultante empiece por 0000. Como se puede observar, después de ser minado, se establece un nonce y el bloque pasa a ser válido.

## Bloque



Bloque: # 1

Nonce: 69926

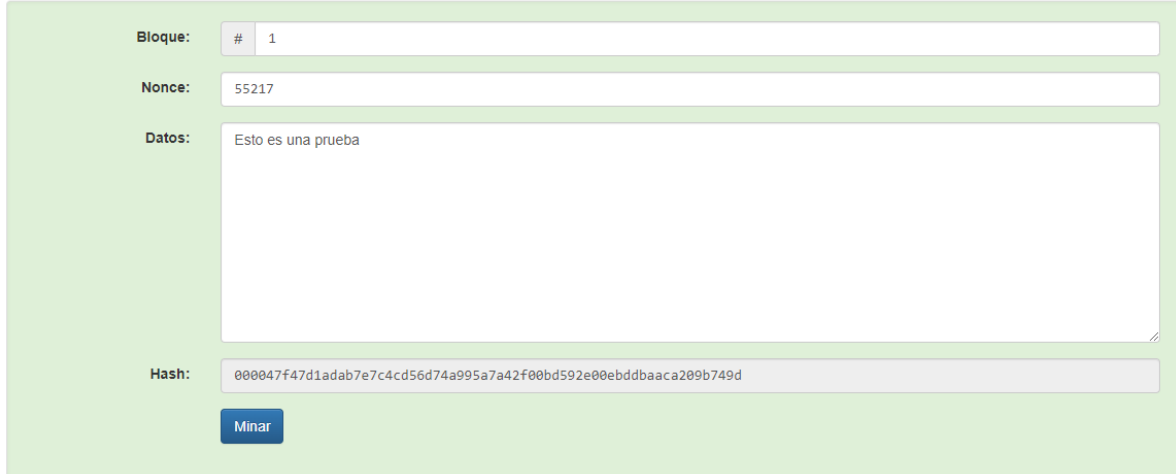
Datos: Esto es una prueba

Hash: 2f3ff6a92ef4e0360971fd81a298bbd3e70999f071707edb6fcc459ef6627cc9

Minar

*Ilustración 29 - Bloque antes de ser minado*

## Bloque



Bloque: # 1

Nonce: 55217

Datos: Esto es una prueba

Hash: 000047f47d1adab7e7c4cd56d74a995a7a42f00bd592e0ebddbaaca209b749d

Minar

*Ilustración 30 - Bloque después de ser minado*

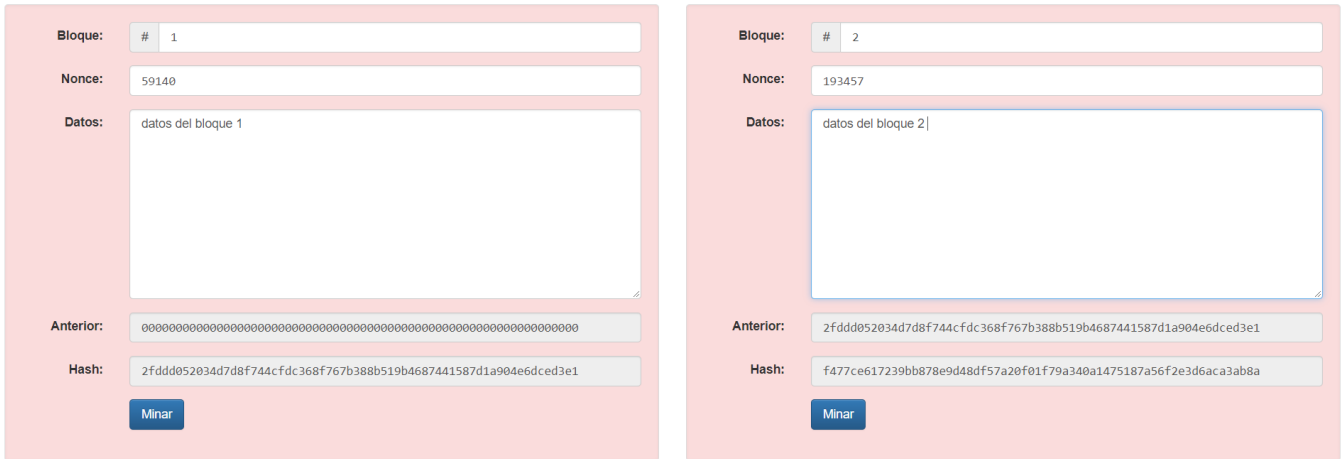
Por último, se muestra el ejemplo de una cadena de bloques (Ilustración 31, Ilustración 32 e Ilustración 33). Como se puede observar, al principio los dos bloques son inválidos. Al ser bloques pertenecientes a una cadena, ambos bloques contienen el hash del bloque previo, siendo todo ceros en el caso del primero.

Cuando se mina el primer bloque y se obtiene un hash para que el bloque sea válido, se actualiza el campo de hash del bloque previo del bloque 2, pero este sigue siendo inválido ya que debe ser minado.

Por último, cuando el bloque 2 es minado, se transforma en un bloque válido.

Cabe destacar que, si se modificase la información de alguno de los bloques, todos los bloques que han sido añadidos más adelante serían inválidos ya que el hash del bloque modificado habría cambiado. Esto hace que blockchain sea inmutable.

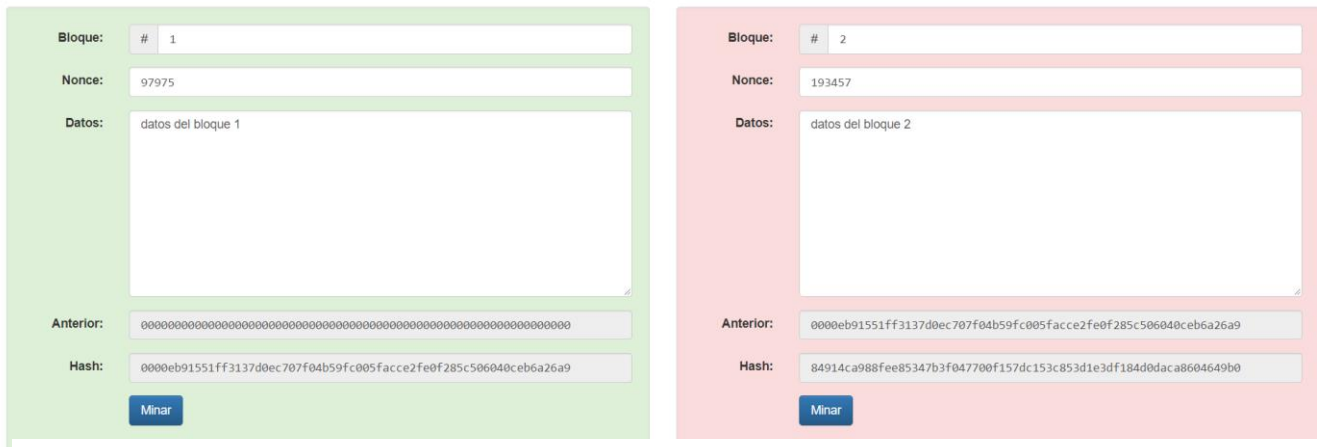
Blockchain



Bloque:	#	Nonce:	Datos:	Anterior:	Hash:
1	1	59140	datos del bloque 1	00	2fddd052034d7d8f744cfdc368f767b388b519b4687441587d1a904e6dced3e1
2	2	193457	datos del bloque 2	2fddd052034d7d8f744cfdc368f767b388b519b4687441587d1a904e6dced3e1	f477ce617239bb878e9d48df57a20f01f79a340a1475187a56f2e3d6aca3ab8a

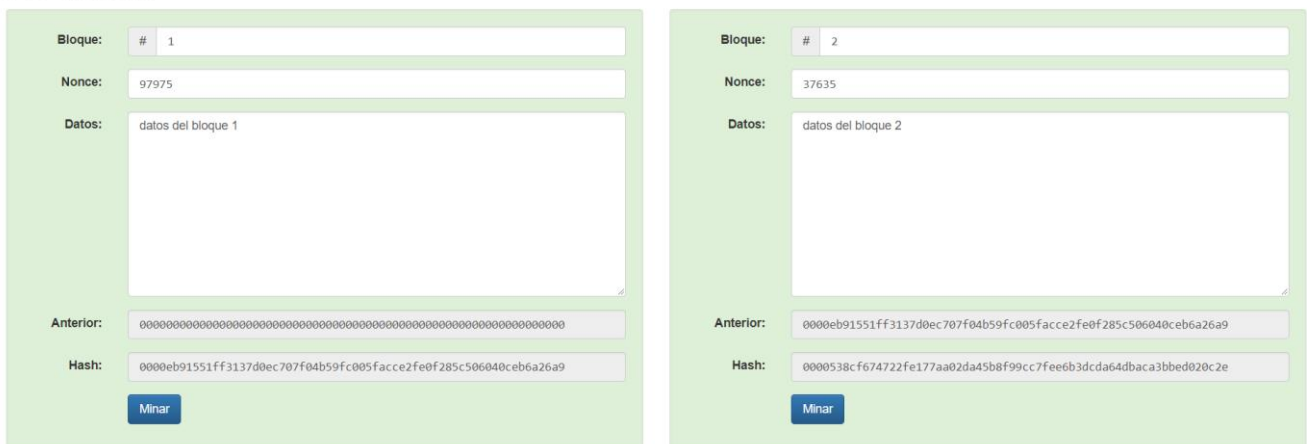
*Ilustración 31 - Ejemplo blockchain (I)*

Blockchain



Bloque:	#	Nonce:	Datos:	Anterior:	Hash:
1	1	97975	datos del bloque 1	00	0000eb91551ff3137d0ec707f04b59fc005facc2fe0f285c506040ceb6a26a9
2	2	193457	datos del bloque 2	0000eb91551ff3137d0ec707f04b59fc005facc2fe0f285c506040ceb6a26a9	84914ca988fee85347b3f047700f157dc153c853d1e3df184d0dac8604649b0

*Ilustración 32- Ejemplo blockchain (II)*



Bloque:	#	Nonce:	Datos:	Anterior:	Hash:
1	1	97975	datos del bloque 1	00	0000eb91551ff3137d0ec707f04b59fc005facc2fe0f285c506040ceb6a26a9
2	2	37635	datos del bloque 2	0000eb91551ff3137d0ec707f04b59fc005facc2fe0f285c506040ceb6a26a9	0000538cf674722fe177aa02da45b8f99cc7fee6b3dcd46dbaca3bb020c2e

*Ilustración 33 - Ejemplo blockchain (III)*

## 5.4 TRANSACCIONES

Las transacciones son otro elemento clave dentro de cualquier blockchain. Representan la transferencia de criptomonedas o valor entre dos direcciones dentro de la red blockchain.

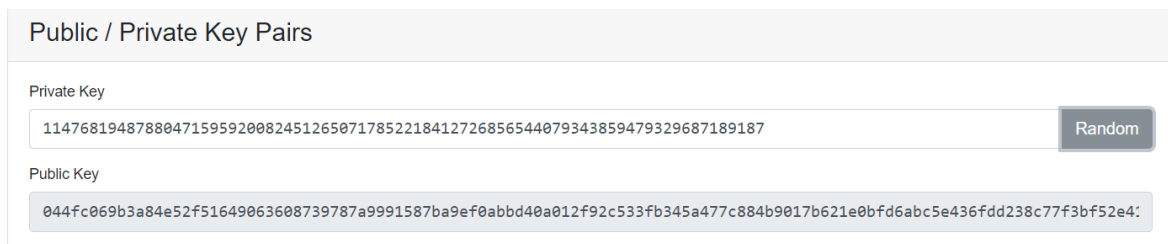
Para firmar y verificar las transacciones aparece el concepto de clave pública y clave privada.

- Clave privada: Solo es conocida por el propietario de la llave y se emplea para firmar transacciones. Esta clave jamás debe ser compartida.
- Clave pública: Deriva de la clave privada y se emplea para que otras personas puedan verificar las transacciones. La clave pública es la dirección de nuestra cartera.

El proceso de firma de una transacción es un proceso que solo puede realizarse en una dirección, es decir, es una función que genera una firma, pero a partir de una firma no se pueden conocer los datos de una transacción. La firma se genera sometiendo a hash a los datos de una transacción junto con la clave privada.

Si alguien quiere verificar una transacción, solo necesita conocer el mensaje, la firma y la clave pública.

A continuación, se muestra un ejemplo de un par de claves (Ilustración 34):



Public / Private Key Pairs

Private Key

114768194878804715959200824512650717852218412726856544079343859479329687189187 Random

Public Key

044fc069b3a84e52f51649063608739787a9991587ba9ef0abbd40a012f92c533fb345a477c884b9017b621e0bfd6abc5e436fdd238c77f3bf52e4:

*Ilustración 34 - Public/ Private Key Pair*

Supongamos que ahora firmamos una transacción con la clave privada y generamos una firma tal y como se ve en la siguiente imagen (Ilustración 35):

**Transaction**

Sign    Verify

Message

\$ 20.00    From: 044fc069b3a84e52f516490636087397:    ->    04cc955bf8e359cc7ebbb66f4c2dc616:

Private Key

114768194878804715959200824512650717852218412726856544079343859479329687189187

Sign

Message Signature

304602210087e784a0e4c7b77583df9e3f7b7ea9daf3a3bd14097c887c14a085e662852b92022100d440e3c6a11e8014a0c1ae784aa8e2f84baebbc

*Ilustración 35 - Ejemplo de firma de una transacción*

Si ahora intentamos verificar la transacción empleando la clave pública asociada a la clave privada que hemos generado, obtenemos un resultado correcto (Ilustración 36)

Sign    **Verify**

Message

\$ 20.00    From: 044fc069b3a84e52f516490636087397:    ->    04cc955bf8e359cc7ebbb66f4c2dc616:

**Signature**

304602210087e784a0e4c7b77583df9e3f7b7ea9daf3a3bd14097c887c14a085e662852b92022100d440e3c6a11e8014a0c1ae784aa8e2f84baebbc

Verify

*Ilustración 36 - Ejemplo verificación de transacción correcta*

Si, por el contrario, empleamos una firma distinta podemos ver como el proceso de verificación falla (Ilustración 37).

Sign    **Verify**

Message

\$ 20.00    From: 044fc069b3a84e52f516490636087397:    ->    04cc955bf8e359cc7ebbb66f4c2dc616:

Signature

000000

Verify

*Ilustración 37 - Ejemplo verificación de transacción incorrecta*

## 5.5 GAS

Dentro de la red de Ethereum hay un concepto llamado gas. El gas es la unidad de medida del esfuerzo computacional que requiere ejecutar una acción dentro de la red de Ethereum.

Cada transacción realizada dentro de la red de Ethereum tiene asociado un coste computacional para ser ejecutada, y, por lo tanto, todas las transacciones requieren de una tasa de gas.

Las tasas de gas son pagadas en ether (ETH), que como ya se ha mencionado es la moneda nativa de Ethereum. Generalmente, el precio del gas esta denotado en gwei, que es una unidad más pequeña que el ether. Un gwei equivale a 0.000000001 ETH.

## Capítulo 6. SISTEMA/MODELO DESARROLLADO

### 6.1 ARQUITECTURA

En este capítulo se va a comentar la arquitectura diseñada para el modelo de oráculo centralizado y para el modelo descentralizado. Además, se comentará el funcionamiento de algunas partes comunes para ambos modelos.

A continuación, se muestra una imagen de la arquitectura diseñada para el modelo de oráculo centralizado (Ilustración 38):

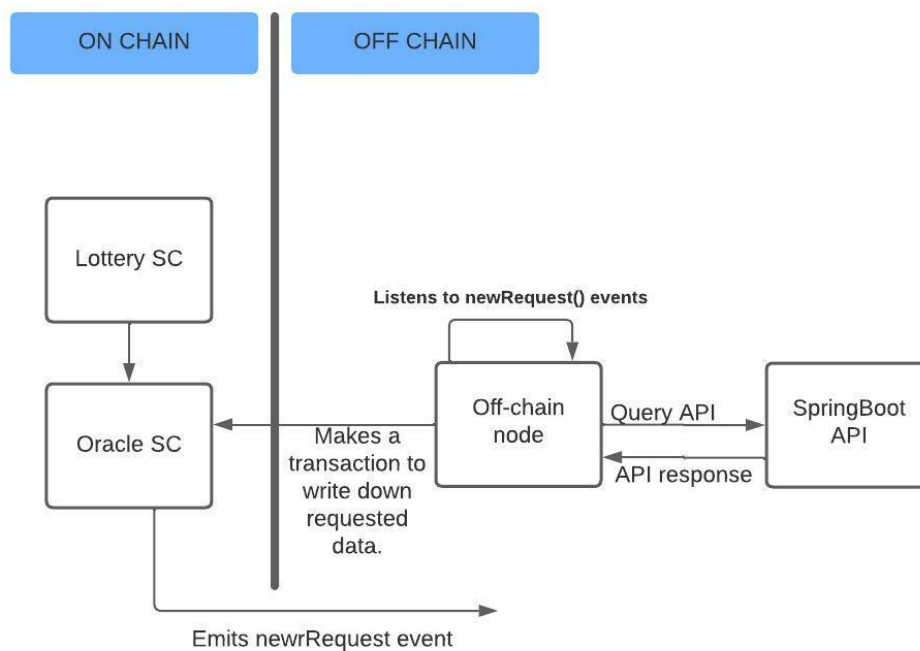


Ilustración 38 - Arquitectura oráculo centralizado

Como se puede observar en la imagen, la arquitectura del oráculo centralizado tiene dos partes principales. La primera parte, se encuentra dentro de la blockchain y se trata de los



Smart Contract y, la segunda parte, son el nodo y la API que se encuentra fuera de la blockchain.

Dentro de la blockchain hay dos Smart Contract: el Smart Contract de la aplicación de loterías y el Smart Contract oráculo. Nos vamos a referir a ellos como Lottery SC y Oracle SC respectivamente.

El lenguaje de Solidity soporta herencia múltiple de contratos, es decir, varios contratos pueden ser heredados en un único contrato. El Oracle SC se trata de un contrato que sirve para comunicarse con el mundo exterior y debe ser heredado por Lottery SC, es decir, el contrato de la aplicación.

Cuando una lotería acaba, empieza el proceso para solicitar un dato fuera de la cadena. En este proceso, Lottery SC llamará una función llamada `newRequest()` de Oracle SC. Cuando esta función es llamada, se genera una nueva solicitud y se emite un evento que ha sido nombrado como `newRequest()`. Hay que destacar que Oracle SC guardará en la petición la dirección del nodo que se encuentra fuera de la cadena y será del único nodo del que acepte una respuesta.

Una vez el evento `newRequest()` ha sido emitido, un nodo que se encuentra fuera de la blockchain y que ha sido configurado para estar escuchando eventos de este tipo se encargará de realizar una consulta en la API programada. Cuando obtiene la respuesta de la API en la que se encuentra el número aleatorio, realizará una transacción para escribir este dato dentro de la cadena.

El Oracle SC cuando recibe el dato por parte del nodo de fuera de la cadena, comprobará que ha sido dicho nodo, es decir, el nodo confiable, quién ha realizado la transacción. Si dicha comprobación es correcta, realizará una llamada a un método de Lottery SC llamado `fulfillRandomness()` para informar que ya se dispone del dato solicitado.

A continuación, se muestra una imagen con la arquitectura del modelo de oráculo descentralizado (Ilustración 39). La arquitectura es muy similar al modelo de oráculo centralizado, pero con unos ligeros cambios.

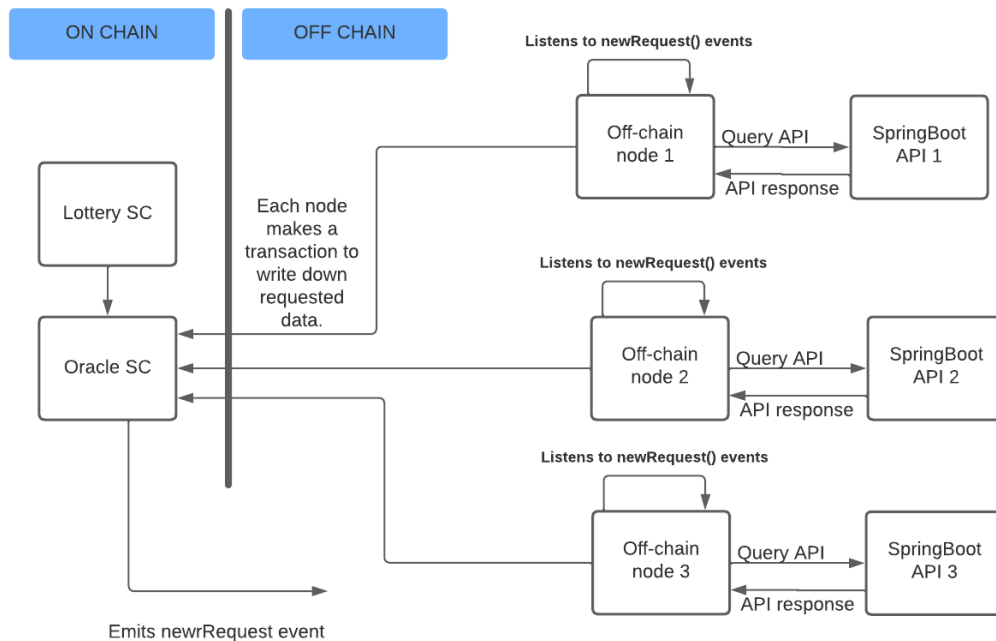


Ilustración 39 - Arquitectura oráculo descentralizado

Como se puede observar, ambas arquitecturas son muy similares. El principal cambio es que ya no habrá un único nodo fuera de la cadena, sino que serán varios los nodos que se encuentren escuchando a la espera de un nuevo evento del tipo newRequest().

En el modelo de oráculo descentralizado, cuando se produce un nuevo evento newRequest(), cada nodo que se encuentra fuera de la cadena realizará una consulta a una API distinta. Cuando obtengan respuesta por parte de la API, cada nodo realizará una transacción distinta para escribir todas las respuestas en la blockchain.

Con esto surge el problema de que existen varias respuestas distintas y el Oracle SC necesita decidir un valor único y verdadero. Para ello, se realiza un proceso de agregación de datos. En este caso, al tratarse de números aleatorios dicho proceso se realiza obteniendo la media de todos los datos que han sido introducidos a la cadena ya que dicho

número seguirá siendo un número aleatorio. Para otros tipos de datos, como por ejemplo el precio ETH/USD se puede obtener la mediana para conseguir un único valor fiable.

## **6.2 SMART CONTRACT: APLICACIÓN DE LOTERÍA**

Ahora que ya sabemos la arquitectura desarrollada para el modelo de oráculo centralizado y descentralizado, se va a explicar el funcionamiento del Smart Contract de la aplicación de loterías.

Este contrato va a simular un juego de loterías muy sencillo que tiene como objetivo poder poner a prueba los distintos oráculos.

El contrato que contiene la aplicación de loterías se llama Lottery.sol. Ha sido desarrollado en el lenguaje de programación Solidity para la versión del compilador 0.6.6 o superior. Además, dado que se importan contratos de OpenZeppelin y Chainlink está bajo la licencia del MIT.

```
// SPDX-License-Identifier: MIT  
pragma solidity ^0.6.6;
```

Este contrato emplea la característica de herencia múltiple permitida por el lenguaje Solidity. El contrato Lottery hereda dos contratos distintos: Ownable y Oracle.

El contrato Ownable ha sido desarrollado por OpenZeppelin y es importado de su Github. Este contrato permite determinar el propietario de un contrato y declarar algunas funciones en modo “onlyOwner”. Esto sirve para poder restringir el acceso a dichas funciones y que únicamente la dirección del propietario pueda tener acceso a estas.

El segundo contrato heredado es Oracle y se trata del contrato que se ha desarrollado para introducir datos desde fuera de la cadena. Este contrato será explicado en los capítulos correspondientes a cada modelo de oráculo.

```
import "@openzeppelin/contracts/access/Ownable.sol";  
import "./Oracle.sol"  
contract Lottery is Ownable, Oracle {
```

La lotería tiene tres estados posibles:

- **OPEN:** Indica que hay una lotería en curso y que está abierta para que nuevos jugadores puedan comprar tickets.
- **CLOSED:** Indica que no hay ninguna lotería en curso en este momento.
- **PROCESSING:** Indica que hay una lotería en curso y que está eligiendo un ganador. Durante este estado no se podrán comprar nuevos tickets.

```
enum STATE {  
OPEN,  
CLOSED,  
PROCESSING  
}
```

Ahora vamos a comentar el constructor de este Smart Contract. En el constructor, se recibe como argumento una dirección de una fuente de precios o priceFeed. Esto sirve para poder emplear el contrato de Chainlink AggregatorV3Interface y que en este caso será empleado para poder obtener el precio USD/ETH. Este es otro ejemplo de la necesidad que tienen los Smart Contract de recibir datos del exterior y, de hecho, es un oráculo. Además, en el constructor se establece el precio del ticket, que en este caso será de 5 dólares estadounidenses y se establece el estado de la lotería como CLOSED.

```
constructor(address _priceFeedAddress) public {  
    usdEntryFee = 5 * (10**18); // 5$  
    ethUsdPriceFeed = AggregatorV3Interface(_priceFeedAddress);  
    state = STATE.CLOSED;  
}
```

La lotería cuenta con una función llamada newGame() que ha sido declarada como onlyOwner. Es una función que únicamente el creador del contrato, es decir, el propietario, puede llamar. Esta función se encarga de crear una nueva lotería. Para ello, comprueba que no haya otra lotería en curso, en cuyo caso, respondería con un error.

```
function startLottery() public onlyOwner {
    require(
        state == STATE.CLOSED,
        "There is a lottery in progress! You can't open a new lottery"
    );
    state = STATE.OPEN;
}
```

Las funciones `getTicketPrice()` y `buyTicket()` sirven para que nuevos jugadores puedan participar en la lotería. En el caso de la función `getTicketPrice()` se trata de una función para obtener el precio del ticket en ETH y es una función destinada a que los jugadores puedan consultar el precio. Por otra parte, la función `buyTicket()` permite la compra de tickets para jugar en la lotería y añade a los jugadores a esta. Esta última función comprueba que la loteria se encuentre en el estado OPEN donde nuevos jugadores pueden participar y que estén enviando una cantidad de ETH igual o superior al precio del ticket, en caso contrario devuelve un error.

```
function getTicketPrice() public view returns (uint256) {
    //Calling priceFeed - Chainlink
    (, int256 price, , , ) = ethUsdPriceFeed.latestRoundData();
    uint256 adjustedPrice = uint256(price) * 10**10; //18 decimals
    uint256 ticketCost = (usdEntryFee * 10**18) / adjustedPrice;
    return ticketCost;
}
function buyTicket() public payable {
    //5$
    require(state == STATE.OPEN);
    require(msg.value >= getTicketPrice(), "Not enough ETH!");
    lottery_players.push(msg.sender);
}
```

Por último, la función `endGame()` que ha sido declarada como `onlyOwner` para que solo el propietario del contrato pueda ejecutarla. Esta función finaliza una lotería, cambiando el estado a `PROCESSING` y solicitando un nuevo dato a través de la función `newRequest()` del contrato `Oracle`. En dicha solicitud, se incluye la dirección de la API de la que preferiblemente se quiere obtener el número aleatorio.

```
function endLottery() public onlyOwner {
    state = STATE.PROCESSING;
    uint256 requestId=newRequest("http://localhost:8080/api/random");
}
```

Además, debido a que hereda el contrato `Oracle`, se ha implementado mediante `override` el método abstracto `fulfillRandomness`, que en Solidity es llamado virtual. Este método será llamado por el contrato `Oracle` cuando tenga disponible el dato solicitado y nuestro contrato de loterías podrá calcular un nuevo ganador y cambiar al estado `CLOSED`.

```
function fulfillRandomness(uint256 _requestId) internal override {
    require(state == STATE.PROCESSING, "Incorrect State");
    random_number = requests[_requestId].value;
    require(random_number > 0, "Random not found");
    random_index = random_number % lottery_players.length;
    last_winner = lottery_players[random_index];
    last_winner.transfer(address(this).balance);
    //Reset
    lottery_players = new address payable[](0);
    state = STATE.CLOSED;
}
```

### 6.3 *NODO OFF-CHAIN*

Ahora se va a comentar el código y funcionamiento del nodo off-chain, es decir, el nodo que se encuentra fuera de la blockchain. Este nodo está escuchando los eventos emitidos por el contrato de loterías, para ello necesita la dirección en la que ha sido desplegado el contrato y el abi (Application Binary Interface).

Para obtener la dirección del contrato, realizará una consulta a una API que contiene la dirección del último contrato de loterías desplegado.

La URL del HTTPProvider de Web3 cambia en función de si estamos realizando pruebas en la red local de Ganache o en Rinkeby. En el caso de la red local de Ganache, la dirección es local, mientras que en el caso de Rinkeby se trata de una API de Infura.

- Ganache: <http://localhost:8545>
- Rinkeby: <https://rinkeby.infura.io/v3/85255d60dc214513a8eef637d7c284ce>

```
# Connection information
url = ""
if (
    network.show_active() in LOCAL_BLOCKCHAIN_ENVIROMENTS
    or network.show_active() in FORKED_LOCAL_ENVIROMENTS
):
    print("LOCAL GANACHE")
    url = "http://localhost:8545"
    url_address = "http://localhost:8080/api/addressganache"
else:
    print("TESTNET RINKEBY")
    url = "https://rinkeby.infura.io/v3/85255d60dc214513a8eef637d7c284ce"
    url_address = "http://localhost:8080/api/addressrinkeby"
# Contract address and abi
response = requests.get(url_address)
data = response.text
parse_json = json.loads(data)
contract_address = parse_json["address"]
contract_abi = Lottery.abi
```

Con estos datos se establece el HTTPProvider de Web3 y el contrato:

```
web3 = Web3(Web3.HTTPProvider(url))
contract = web3.eth.contract(address=contract_address, abi=contract_abi)
```

Además, y de cara al sistema descentralizado, se ofrecen tres posibles direcciones de cuentas para el nodo. De este modo, podremos simular tres nodos distintos cada uno con una dirección de Ethereum correspondiente a una cuenta distinta. Para seleccionar la ID del nodo, se pregunta al usuario que elija una opción. Además, cada opción tiene en cuenta si se está ejecutando en la red local de Ganache o en Rinkeby.

```
# ID of the node: Account who manage the node.
account = ""
secret_key = ""
selected = False
while selected == False:
    value = input("Select node id (1 - 2 - 3)\n")
    if int(value) == 1:
        if (
            network.show_active() in LOCAL_BLOCKCHAIN_ENVIROMENTS
            or network.show_active() in FORKED_LOCAL_ENVIROMENTS
        ):
            # Local-Ganache
            account = "0xB5fB12fd8148441fE7Ad208135dC376923Ff349B"
            secret_key = (
                "0x2F45E72EAEDFB7F2F9AFA52C1B80D7E69C2BAC1CFD35EE746D2AED93917F397B"
            )
        else:
            # Testnet-Rinkeby
            account = "0xbB8147F66FaF71A5bA41E5bD074d6562bd9DB362"
            secret_key = config["wallets"]["from_key"]
            selected = True
    if int(value) == 2:
        if (
            network.show_active() in LOCAL_BLOCKCHAIN_ENVIROMENTS
            or network.show_active() in FORKED_LOCAL_ENVIROMENTS
        ):
            # Local-Ganache
            account = "0x3aD774db8f3d772f214Ae509C41D990b05221EAD"
```



```
secret_key = (  
"0x891fa6b28d036e9da394d9c3e986bc2c84ee2b821d62657dda5a827f272e5658"  
)  
else:  
    # Testnet-Rinkeby  
    account = "0xC9A5E426bC9af443A1D3Cb0539ef96d17db8bea1"  
    secret_key = config["wallets"]["from_key2"]  
    selected = True  
if int(value) == 3:  
    if (  
        network.show_active() in LOCAL_BLOCKCHAIN_ENVIROMENTS  
        or network.show_active() in FORKED_LOCAL_ENVIROMENTS  
    ):  
        # Local-Ganache  
        account = "0x8EC83D8Fd96A13beD7984796B3aC8A3B48FbAD0A"  
        secret_key = (  
"0xe2f5a02199825448892e321543e22757dab765927e14b8bb5c95f169305aeb96"  
)  
    else:  
        # Testnet-Rinkeby  
        account = "0xc9c68d75123Aa15dcFFcF52ad965bCDF0D3Ec216"  
        secret_key = config["wallets"]["from_key3"]  
        selected = True
```

Ahora, una vez que el nodo tiene asociada una cuenta de Ethereum, asignado el HTTPProvider y conoce el contrato, vamos a explicar como se ha programado para que escuche los eventos NewRequest() del contrato de loterías.

Para ello, se ha creado un filtro que busca en el último bloque en búsqueda de eventos del tipo NewRequest del contrato de loterías. Con este filtro, se inicializa un bucle asíncrono que es llamado cada dos segundos y se comprueba si hay un nuevo evento de ese tipo. En caso de encontrar un evento de ese tipo, entonces se llama al método handle\_event().

```
async def log_loop(event_filter, poll_interval):
    while True:
        for NewRequest in event_filter.get_new_entries():
            handle_event(NewRequest)
        await asyncio.sleep(poll_interval)
def main():
    event_filter =
contract.events.NewRequest.createFilter(fromBlock="latest")
    loop = asyncio.get_event_loop()
    try:
        loop.run_until_complete(asyncio.gather(log_loop(event_filter, 2)))
    finally:
        # close loop to free up system resources
        loop.close()
```

El método `handle_event()` lo primero que hará será obtener algunos datos a partir del evento `NewRequest`. Estos datos son la id de la petición y la url a la que realizar la query.

Una vez tiene estos datos, realiza una petición GET a la API indicada en la url y espera una respuesta. Con la respuesta, que en nuestro caso será el número aleatorio para el Smart Contract, crea una nueva transacción para escribir este dato en la cadena llamando al método `updateRequest()` del contrato de loterías (realmente es un método que el contrato de loterías hereda del contrato Oracle).

```
def handle_event(event):
    # First we extract id and url of the event
    event_json = Web3.toJSON(event)
    id = event["args"]["id"]
    url = event["args"]["url"]
    print("ID:", id)
    # Now, we send a get to URL
    response_API = requests.get("http://localhost:8080/api/random")
    # We extract data from response
    data = response_API.text
    parse_json = json.loads(data)
    # We get the random_number
    random_number = parse_json["value"]
    print("Random number", random_number)
```

```
# Store random_number in the smart contract
raw_transaction = contract.functions.updateRequest(
    id, random_number
).buildTransaction(
    {
        "gas": 6721975,
        "from": account,
        "nonce": web3.eth.getTransactionCount(account),
    }
)

signed = web3.eth.account.signTransaction(
    raw_transaction,
    secret_key,
)
receipt = web3.eth.sendRawTransaction(signed.rawTransaction)
web3.eth.waitForTransactionReceipt(receipt)

print("Random number stored in the blockchain")
```

En el caso del modelo de oráculo descentralizado, la url del evento se ignora y cada nodo realiza una query a una API distinta. De este modo, conseguimos que sea un sistema descentralizado y que no dependa de una única API.

## ***6.4 API EN SPRINGBOOT***

Además, se ha diseñado una aplicación de backend muy sencilla haciendo uso de Springboot con los siguientes objetivos:

- Proporcionar números aleatorios
- Poder almacenar la dirección del último contrato desplegado
- Poder consultar la dirección del último contrato desplegado

Para ello, se han diseñado varios endpoints que servirán para cumplir las distintas necesidades expuestas anteriormente.

En primer lugar, varios endpoints donde poder realizar un GET para obtener números aleatorios. A continuación, se muestra el código de uno de esos endpoints:

```
@GetMapping("/random")
public ResponseEntity < RandomNumber > getRandomNumber () {
    BigInteger random = new BigInteger(256, new Random());
    return new ResponseEntity < RandomNumber > (new RandomNumber(random),
    HttpStatus.OK);
}
```

Por otra parte, se han creado dos endpoints donde realizar un POST de la última dirección de Ethereum del último contrato de loterías desplegado. Se ha creado un endpoint para trabajar en local con Ganache y otro para la testnet de Rinkeby.

```
@PostMapping("/addressganache")
public ResponseEntity<String> newLocalAddress (@RequestBody String address)
{
    addressLocal= new ContractAddress (address);
    return new ResponseEntity<String>("Ok",HttpStatus.OK);
}
@PostMapping("/addressrinkeby")
public ResponseEntity<String> newRinkebyAddress (@RequestBody String
address) {
    addressRinkeby= new ContractAddress (address);
    return new ResponseEntity<String>("Ok",HttpStatus.OK);
}
```

Por último, los endpoints para poder realizar un GET y obtener dicha dirección. De nuevo, tanto para trabajar en local con Ganache como para la testnet de Rinkeby.

```
@GetMapping("/addressganache")
public ResponseEntity<ContractAddress> getLocalAddress () {
    return new ResponseEntity<ContractAddress>(addressLocal,
    HttpStatus.OK);
}
```

```
@GetMapping("/addressrinkeby")  
public ResponseEntity<ContractAddress> getRinkebyAddress () {  
    return new ResponseEntity<ContractAddress>(addressRinkeby,  
    HttpStatus.OK);  
}
```

Los endpoints destinados a realizar el POST y GET de la dirección del último contrato de lotería desplegado serán de gran ayuda para agilizar el proceso de testeo.

Por otra parte, se han generado distintos endpoints de números aleatorios, aunque aquí solo se muestra uno ya que son idénticos, con la finalidad de simular llamadas a distintas API para el modelo de oráculo descentralizado.

## Capítulo 7. ORÁCULO CENTRALIZADO

En este capítulo se va a explicar el modelo de oráculo centralizado y se van a mostrar los resultados obtenidos. Además, se va a realizar un ejemplo para demostrar que el modelo de oráculo centralizado supone un riesgo a la seguridad y por lo que no debería de ser empleado.

El contrato Oracle diseñado para este modelo centralizado ha sido creado en el lenguaje de programación Solidity para la versión del compilador 0.6.6 o superior.

```
pragma solidity ^0.6.6;
```

Lo primero que tiene el contrato Oracle es una estructura de datos llamada Request. Esta estructura de datos se emplea para poder agrupar toda la información de una petición. Además, se ha creado un array llamado requests del tipo Request para poder almacenar todas las peticiones.

La información necesaria en una petición es:

- Id: Identificador del número de petición
- Url: Se trata de la url de la API de la que preferiblemente se obtendrán los datos fuera de la cadena.
- Value: Es el valor definitivo del dato que ha sido solicitado. Se rellenará cuando el nodo de fuera de la cadena introduzca el dato realizando una transacción.
- Mapa de oráculos: En este mapa almacenaremos la dirección del nodo confiable asociado a un entero. De este modo, podremos saber si el nodo ha enviado una respuesta o aún no la ha enviado.

```
struct Request {  
    uint256 id;  
    string url;  
    uint256 value;  
    mapping(address => uint256) oracles; //1 if hasn't send answer, 2 if  
has send answer
```

```
}  
  
Request[] public requests;
```

Además, se ha declarado un evento llamado `NewRequest()`. Este evento se emitirá cuando haya una nueva petición para que el nodo que se encuentra fuera de la cadena escuchando sea informado. En este evento, se incluye la id de la petición y la url a la que preferiblemente se va a realizar la consulta fuera de la cadena.

```
event NewRequest(uint256 id, string url);
```

La primera función que tiene el contrato Oracle es `newRequest()`. Esta función está diseñada para que el contrato que contiene la aplicación la llame cuando quiere iniciar una petición para obtener un dato de fuera de la cadena.

La función crea una nueva `Request` y la almacena en el array de `requests`. Además, establece en la petición la dirección del nodo confiable que se encuentra fuera de la cadena. Como en este proyecto se van a realizar pruebas tanto en Ganache como en Rinkeby, se ha establecido una dirección para cada red. El nodo confiable será el único que pueda almacenar el valor del dato pedido en la petición.

Por último, la función emite un nuevo evento del tipo `NewRequest` e incrementa el número de id para la siguiente petición.

```
function newRequest(string memory _url) internal returns (uint256) {  
    requests.push(Request(currentId, _url, 0));  
    uint256 length = requests.length;  
    Request storage r = requests[length - 1];  
    //Trusted oracle address - Address of account that are trusted to  
    give the final solution.  
    //=====  
    //Rinkeby  
    r.oracles[address(0xbB8147F66FaF71A5bA41E5bD074d6562bd9DB362)] = 1;  
    //Rinkeby  
    //Local
```

```
    r.oracles[address(0xB5fB12fd8148441fE7Ad208135dC376923Ff349B)] = 1;
//Local ganache
//Emit new event. Oracles off-chain will be listening
emit NewRequest(currentId, _url);
//Increase request current id;
currentId++;
return (currentId - 1);
}
```

La siguiente función implementada en el contrato Oracle es `updateRequest()`. Esta función está diseñada para que el nodo que se encuentra fuera de la cadena pueda llamarla, realizando una transacción y escribiendo el valor del dato pedido en el contrato, que se encuentra en la blockchain.

Esta función recupera a través de la `id`, que ha sido pasada en la llamada por argumento, la petición en cuestión. Luego comprueba que se trate del nodo confiable y que, además, dicho nodo no haya escrito una respuesta previamente. En caso de que la comprobación sea correcta, almacena el valor enviado por el nodo en la petición.

Una vez el valor ha sido almacenado, llama a la función `fulfillRandomness()` con la `id` de la petición como argumento.

```
function updateRequest(uint256 _id, uint256 _value) public {
    Request storage currentRequest = requests[_id];
    //Check if the oracle is a trusted one.
    if (currentRequest.oracles[address(msg.sender)] == 1) {
        //Oracle has voted
        currentRequest.oracles[address(msg.sender)] = 2;
        currentRequest.value = _value;
        fulfillRandomness(_id);
    }
}
```

Por último, queda la función `fulfillRandomness()`. Esta función es abstracta, que en el lenguaje de Solidity se declara como `virtual`. Además, es una función interna, es decir, que solo puede ser llamada por el contrato. Se trata de una función abstracta porque debe ser



implementada por el contrato de la aplicación, que heredará el contrato Oracle. Esta función será llamada en el momento que se haya decidido un valor final para una petición.

```
function fulfillRandomness(uint256 _requestId) internal virtual {}
```

## 7.1 EJEMPLO EN RINKEBY

A continuación, se muestra un ejemplo en Rinkeby del correcto funcionamiento del oráculo centralizado.

```
Running 'scripts\deploy_lottery.py::main'...
Transaction sent: 0xea0d0a5eb7e8bac0ac3b12c7b49a24042c60c9ea1fd77b937822e4e074fda53f
Gas price: 1.000000035 gwei Gas limit: 1103147 Nonce: 255
Lottery.constructor confirmed Block: 10389544 Gas used: 1002861 (90.91%)
Lottery deployed at: 0xc91d6B4686fB656523aF48DB2c65f0af6C9D966f

Deployed lottery contract
Press Enter to continue...
0xbB8147F66FaF71A5bA41E5bD074d6562bd9DB362
Enter password for "Account2":
0xc9A5E426bC9af443A1D3Cb0539ef96d17db8bea1
Enter password for "Account3":
0xc9c68d75123Aa15dcFFcF52ad965bCDF0D3Ec216
Loteria 1
=====
Transaction sent: 0x9866a6588997c43e45c93ce71c5beb2b8be9c8dcca0b97afb910d9f0ad80932f
Gas price: 1.000000034 gwei Gas limit: 31572 Nonce: 256
Lottery.startLottery confirmed Block: 10389548 Gas used: 28702 (90.91%)

Lottery.startLottery confirmed Block: 10389548 Gas used: 28702 (90.91%)

The lottery is started
Transaction sent: 0x424c1d554dc72f6c4205442737a7241f032f68f1bafef36309b51f08ef0a29188
Gas price: 1.000000034 gwei Gas limit: 97485 Nonce: 257
Lottery.buyTicket confirmed Block: 10389549 Gas used: 88623 (90.91%)

Lottery.buyTicket confirmed Block: 10389549 Gas used: 88623 (90.91%)

You entered de lottery!
Enter password for "Account2":
Transaction sent: 0x035ac94070bf1dc243ca2739be52eb9fa2df561933d15c506449e635a8c2cc67b
Gas price: 1.000000033 gwei Gas limit: 78675 Nonce: 12
Lottery.buyTicket confirmed Block: 10389553 Gas used: 71523 (90.91%)

Lottery.buyTicket confirmed Block: 10389553 Gas used: 71523 (90.91%)

You entered de lottery!
Enter password for "Account3":
Transaction sent: 0x6bbc78222e3b90d7701f9104c1f391d7659bc83eb014a21b25b3d9849bf0cc8a
Gas price: 1.000000032 gwei Gas limit: 78675 Nonce: 13
Lottery.buyTicket confirmed Block: 10389554 Gas used: 71523 (90.91%)

Lottery.buyTicket confirmed Block: 10389554 Gas used: 71523 (90.91%)

You entered de lottery!
Transaction sent: 0x06a42b0bcf6cb80d3de7df166310c4b22187ee1e60c612f6078c8912a25db7c5
Gas price: 1.000000032 gwei Gas limit: 185854 Nonce: 258
Lottery.endLottery confirmed Block: 10389555 Gas used: 168959 (90.91%)

Lottery.endLottery confirmed Block: 10389555 Gas used: 168959 (90.91%)
```

Para este ejemplo, se muestran una serie de transacciones en la imagen anterior (Ilustración 40) que ahora explicaremos.

En primer lugar, se ha desplegado un nuevo contrato de loterías en la red de Rinkeby. Dicho contrato se ha desplegado en la dirección:

**0xc91d6B4686fB656523aF48DB2c65f0af6C9D966f**

Una vez el contrato ha sido desplegado, se ha inicializado un nodo fuera de la cadena para que escuche los eventos emitidos por este contrato (Ilustración 41).

```
Running 'scripts\node_offchain.py::main'...
ID: 0
Random number 90881474394412569536137920092521283569926460819614648427187526192266527921592
TESTNET
Random number stored in the blockchain
█
```

*Ilustración 41 - Ejemplo Rinkeby centralizado: nodo offchain (II)*

A partir de este momento, se ha inicializado una nueva lotería. En esta lotería tres participantes distintos han comprado tickets. Dichos participantes y el index que les corresponde son:

- **Index 0: 0xbB8147F66FaF71A5bA41E5bD074d6562bd9DB362**
- **Index 1: 0xC9A5E426bC9af443A1D3Cb0539ef96d17db8bea1**
- **Index 2: 0xc9c68d75123Aa15dcFFcF52ad965bCDF0D3Ec216**

Finalmente, se ha finalizado la lotería. En este momento, se ha generado un evento del tipo NewRequest() y el nodo off-chain ha realizado una consulta a la API para obtener un número aleatorio. Dicho número, se ha almacenado en el contrato realizando una transacción que llamaba al método updateRequest(). Una vez el número aleatorio se ha almacenado correctamente, el contrato de loterías ha decidido un nuevo ganador mediante el método fulfillRandomness().

En este ejemplo, y como se puede ver en la ilustración del nodo de fuera de la cadena (Ilustración 41), el número aleatorio ha sido:

90881474394412569536137920092521283569926460819614648427187526192266527921592

El resultado, si hacemos la operación módulo 3 debería ser que el ganador de esta lotería es el index 2.

Transcurrido un tiempo, para asegurar que se han finalizado todas las operaciones, se ha comprobado que el resultado es correcto (Ilustración 42).

```
0xc9c68d75123Aa15dcFFcF52ad965bCDF0D3Ec216 is the new winner
90881474394412569536137920092521283569926460819614648427187526192266527921592RANDOM
2index contract
```

*Ilustración 42 - Ejemplo Rinkeby centralizado: resultados (III)*

Además, podemos comprobar mediante Etherscan que todas las transacciones han sido realizadas correctamente (Ilustración 43).

Txn Hash	Method	Block	Age	From	To	Value	Txn Fee
<a href="#">0x55e978436ec2761644...</a>	Update Request	10389556	100 days 3 hrs ago	<a href="#">0xbb8147f66fa71a5ba4...</a>	<a href="#">0xc91d6b4686fb656523...</a>	0 Ether	0.00014374
<a href="#">0x06a42b0bcf6cb80d3d...</a>	End Lottery	10389555	100 days 3 hrs ago	<a href="#">0xbb8147f66fa71a5ba4...</a>	<a href="#">0xc91d6b4686fb656523...</a>	0 Ether	0.00016895
<a href="#">0x6bbc78222e3b90d770...</a>	Buy Ticket	10389554	100 days 3 hrs ago	<a href="#">0xc9c68d75123aa15dcff...</a>	<a href="#">0xc91d6b4686fb656523...</a>	0.0015995 Ether	0.00007152
<a href="#">0x035ac94070bf1dc243c...</a>	Buy Ticket	10389553	100 days 3 hrs ago	<a href="#">0xc9a5e426bc9af443a1...</a>	<a href="#">0xc91d6b4686fb656523...</a>	0.0015995 Ether	0.00007152
<a href="#">0x424c1d554dc72f6c420...</a>	Buy Ticket	10389549	100 days 3 hrs ago	<a href="#">0xbb8147f66fa71a5ba4...</a>	<a href="#">0xc91d6b4686fb656523...</a>	0.0015995 Ether	0.00008862
<a href="#">0x9866a6588997c43e45...</a>	Start Lottery	10389548	100 days 3 hrs ago	<a href="#">0xbb8147f66fa71a5ba4...</a>	<a href="#">0xc91d6b4686fb656523...</a>	0 Ether	0.0000287
<a href="#">0xea0d0a5eb7e8bac0ac...</a>	0x6006080	10389544	100 days 3 hrs ago	<a href="#">0xbb8147f66fa71a5ba4...</a>	Contract Creation	0 Ether	0.0100286

*Ilustración 43 - Ejemplo Rinkeby centralizado: Etherscan (IV)*

Y también podemos ver, que el premio ha sido pagado al ganador. El jugador con index 2 tenía la siguiente dirección de Ethereum:

- **Index 2: 0xc9c68d75123Aa15dcFFcF52ad965bCDF0D3Ec216**

Han participado tres jugadores con una cantidad de 0.0015995 ETH como podemos ver en la imagen anterior (Ilustración 43). Esto supone que el premio total debería ser de 0.0047985 ETH.

Si ahora comprobamos las transacciones internas, podemos ver como esa cantidad se le ha pagado a esa dirección (Ilustración 44). Por lo que ha funcionado correctamente.

Parent Txn Hash	Block	Age	From	To	Value
0x55e978436ec2761644...	10389566	100 days 3 hrs ago	0xc91d6b4686fb656523...	0xc9c68d75123aa15dcff...	0.0047985 Ether
0x6bbc78222e3b90d770...	10389554	100 days 3 hrs ago	0xc91d6b4686fb656523...	0x8a753747a1fa494ec9...	0 Ether
0x035ac94070bf1dc243c...	10389553	100 days 3 hrs ago	0xc91d6b4686fb656523...	0x8a753747a1fa494ec9...	0 Ether
0x424c1d554dc72f6c420...	10389549	100 days 3 hrs ago	0xc91d6b4686fb656523...	0x8a753747a1fa494ec9...	0 Ether

*Ilustración 44 - Ejemplo Rinkeby centralizado: Etherscan transacciones internas (V)*

A través del siguiente enlace se puede consultar toda esta información:

<https://rinkeby.etherscan.io/address/0xc91d6B4686fB656523aF48DB2c65f0af6C9D966f>

## 7.2 EJEMPLO DE INSEGURIDAD DEL MODELO CENTRALIZADO

Como se ha comentado previamente en el documento, el problema de los oráculos tiene partes. En primer lugar, la necesidad de las blockchain de emplear datos de fuera de la cadena, y, por otra parte, que, si dichos datos son introducidos mediante un sistema centralizado, se pierden las ventajas de dicha blockchain.

En el ejemplo anterior se ha empleado un nodo confiable para introducir un dato desde fuera de la cadena al contrato que se encuentra en la blockchain. El problema es que dicho nodo puede empezar a actuar de forma maliciosa o ser atacado. Al emplear un sistema centralizado, si esto ocurre, nuestro contrato depende de este nodo.

En este ejemplo, se quiere mostrar que, si el nodo de fuera de la cadena o la API comienzan a actuar de forma maliciosa o son hackeados, nuestra aplicación de loterías corre serios riesgos de seguridad y puede ser amañada.

Supongamos que el nodo de fuera de la cadena comienza a actuar de forma maliciosa. Dicho nodo quiere que gane la lotería su amigo. Además, como las transacciones pueden ser consultadas de forma pública y transparente, conoce el orden en el que los jugadores han comprado los tickets y, por lo tanto, el index asociado a cada uno. También dispone del número total de jugadores.

El contrato de loterías calcula el ganador realizando la operación módulo. Para ello, toma el número aleatorio y realiza la operación modulo con el número de jugadores.

Como el nodo de fuera de la cadena dispone de estos datos, puede forzar que gane su amigo introduciendo un número que el previamente sabe que dará como resultado el index de su amigo.

Supongamos que quiere que gane el index 0 porque está asociado a la dirección de su amigo. Para ello, si por ejemplo introduce como número aleatorio 12, habrá forzado a que gane su amigo la lotería.

A continuación, se muestra en un ejemplo:

```
Running 'scripts\deploy_lottery.py::main'...
Transaction sent: 0x8d7609956ee3998d96299dca8047bc394c6bf8432b10aee4c34b04b989ff01f4
Gas price: 1.500000008 gwei Gas limit: 1103147 Nonce: 336
Lottery.constructor confirmed Block: 10962092 Gas used: 1002861 (90.91%)
Lottery deployed at: 0xd07cafC22262bA36E3afc997857C4BeF3A879B57

Deployed lottery contract
TESTNET RINKEBY
```

*Ilustración 45- Ejemplo malicioso Rinkeby Centralizado: Desplegar el contrato (I)*

```
TESTNET RINKEBY
Contract address: 0xd07cafC22262bA36E3afc997857C4BeF3A879B57
Account: 0xbB8147F66FaF71A5bA41E5bD074d6562bd9DB362

Running 'scripts\node_offchain.py::main'...
```

*Ilustración 46 - Ejemplo malicioso Rinkeby centralizado: Inicializar nodo off chain (II)*

```
Loteria 1
=====
Transaction sent: 0xce0797bfafa92937d22278d79398177662d2ae6a26633b7ffa2a8620a4bc51c4
Gas price: 1.500000008 gwei Gas limit: 31572 Nonce: 337
Lottery.startLottery confirmed Block: 10962094 Gas used: 28702 (90.91%)

Lottery.startLottery confirmed Block: 10962094 Gas used: 28702 (90.91%)

The lottery is started
Transaction sent: 0x65ba18ac0f1635574ed0d373db66c11c53c2f9eac8dd6df0eb03dbdbe111e5eb
Gas price: 1.500000008 gwei Gas limit: 97485 Nonce: 338
Lottery.buyTicket confirmed Block: 10962095 Gas used: 88623 (90.91%)

Lottery.buyTicket confirmed Block: 10962095 Gas used: 88623 (90.91%)

You entered de lottery!
Enter password for "Account2":
Transaction sent: 0x132f62640b81f1c5818b34633cc3e55e4fc0696cafb41f9053e7942d3498ffe8
Gas price: 1.500000008 gwei Gas limit: 78675 Nonce: 28
Lottery.buyTicket confirmed Block: 10962098 Gas used: 71523 (90.91%)

Lottery.buyTicket confirmed Block: 10962098 Gas used: 71523 (90.91%)

You entered de lottery!
Enter password for "Account3":
Transaction sent: 0x96f65e0e475a2d998949db2cc04e7f5c8355c530f043542f75da5ae7b60ad686
Gas price: 1.500000008 gwei Gas limit: 78675 Nonce: 28
Lottery.buyTicket confirmed Block: 10962104 Gas used: 71523 (90.91%)

Lottery.buyTicket confirmed Block: 10962104 Gas used: 71523 (90.91%)

You entered de lottery!
Transaction sent: 0x5ec722d1648e995437cbb2cb27f4123bfb0c1c9f134cdb3b5232615dc3bf83ef
Gas price: 1.500000008 gwei Gas limit: 185854 Nonce: 339
Lottery.endLottery confirmed Block: 10962105 Gas used: 168959 (90.91%)

Lottery.endLottery confirmed Block: 10962105 Gas used: 168959 (90.91%)
```

Ilustración 47 - Ejemplo malicioso Rinkeby centralizado: transacciones (III)

```
Running 'scripts\node_offchain.py::main'...
ID: 0
Random number 12
Random number stored in the blockchain
█
```

Ilustración 49 - Ejemplo malicioso Rinkeby centralizado: nodo off-chain (IV)

```
0xbB8147F66FaF71A5bA41E5bD074d6562bd9DB362 is the new winner
12RANDOM
0index contract
0 Index
█
```

Ilustración 48 - Ejemplo malicioso Rinkeby centralizado: resultados (V)

De nuevo, en este ejemplo se despliega un nuevo contrato de loterías y se inicializa el nodo de fuera de la cadena (Ilustración 45 e Ilustración 46).

A continuación, se inicia una nueva lotería y se añaden tres jugadores a esta (Ilustración 47).

Una vez se termina la lotería y se emite el evento NewRequest, el nodo, esta vez de forma maliciosa escribe el número en la cadena mediante una transacción, en esta ocasión en vez de un número aleatorio, introduce directamente el número 12 (Ilustración 49).

Por último, se muestran los resultados de la lotería (Ilustración 48). Como era de esperar, en esta ocasión el ganador ha sido el jugador con index 0 tal y como habíamos predicho.

Toda la información de las transacciones se puede consultar en Etherscan mediante el siguiente enlace:

<https://rinkeby.etherscan.io/address/0xd07cafC22262bA36E3afc997857C4BeF3A879B57>

Como podemos comprobar, un modelo de oráculo centralizado supone grandes riesgos de seguridad a nuestra aplicación. Aunque es un modelo muy sencillo de implementar y barato, no debe ser empleado ya que se pierden todas las ventajas de emplear blockchain y Smart Contract.

## Capítulo 8. ORÁCULO DESCENTRALIZADO

Este capítulo está dedicado al modelo de oráculo centralizado. Se explicarán las diferencias en el contrato Oracle y luego se mostrarán los resultados.

El contrato Oracle aunque es muy similar al empleado en el modelo de oráculo centralizado, ha sufrido algunas modificaciones. Al tratarse de un oráculo descentralizado, debemos tener en cuenta que ya no habrá una única respuesta ya que ahora hay varios nodos fuera de la cadena. Por lo tanto, ahora el contrato Oracle tiene que buscar una manera de agregar todas las respuestas en un único valor.

El primer cambio es en la estructura de Request. En el modelo de oráculo centralizado, la estructura mantiene los campos del modelo centralizado, pero añade dos nuevos campos:

- `Votes_number`: El número de respuestas recibidas hasta el momento para una petición.
- `Answers`: Un array que contiene todas las respuestas que los nodos off-chain han ido rellenando.

```
struct Request {
    uint256 id;
    string url;
    uint256 votes_number;
    uint256 value;
    uint256[] answers;
    mapping(address => uint256) oracles; //1 if hasn't send answer, 2 if
has send answer
}
```

Además, se añade una nueva variable llamada `min_votes`. Esta variable establece el quorum, es decir, el mínimo número de respuestas que tiene que recibir el contrato Oracle antes de poder decidir un valor final para una petición.



Dicho valor se ha establecido en 3 para poder realizar pruebas de manera sencilla en este proyecto, pero se puede modificar en el caso de querer que más nodos de fuera de la cadena envíen una respuesta.

```
uint256 min_votes = 3;
```

La función `newRequest()` sigue funcionando de la misma manera que en el modelo centralizado. La diferencia principal está en que ahora, en lugar de haber una única dirección de un nodo confiable, hay tantas como el quorum sea.

Hay que destacar que, para nuestro ejemplo, al tratarse de un modelo de oráculo descentralizado ya no hay necesidad de establecer nodos confiables. En el modelo centralizado, esto era de vital importancia ya que el correcto funcionamiento dependía únicamente de un nodo, pero en el caso del modelo descentralizado ya no es tan importante. En este modelo, podríamos permitir que cualquier nodo, sea confiable o no, introduzca datos en la cadena, ya que con que un único nodo escriba un número aleatorio, el valor final será aleatorio.

Por otra parte, los cambios más notorios se producen en la función `updateRequest()`. Como hemos comentado antes, al tratarse de un modelo descentralizado, ahora existen varias respuestas para una misma petición. Esto significa que ya no podemos tomar el valor de la primera respuesta como valor final para la petición, tal y como hacíamos en el modelo centralizado.

El método `updateRequest()` ahora también cumple la función de agregar los datos. Como nuestro ejemplo busca introducir números aleatorios a la cadena, se ha buscado una manera sencilla de agregar todas las respuestas en un único valor.

Para agregar todas las respuestas en un único valor, y que dicho valor siga siendo un número aleatorio, se realiza la media de todas las respuestas. De este modo, aunque alguna

de las respuestas no fuera un número aleatorio, mientras una sola respuesta sí lo sea, obtendríamos como valor final un número aleatorio.

La forma de realizar esta operación y para evitar que desborde una variable es dividir cada una de las respuestas que se aceptan entre el quorum y añadir el resultado al valor final. De este modo, nos aseguramos de que la respuesta final sea un uint256 y que en ninguno de los pasos intermedios desborda una variable.

Finalmente, cuando se alcanza el quorum, se ha obtenido un valor final que será un número aleatorio. Cuando llega este momento, se llama al método fulfillRandomness() tal y como se hacía en el modelo centralizado.

```
function updateRequest(uint256 _id, uint256 _value) public {
    Request storage currentRequest = requests[_id];
    //Check if the oracle is a trusted one.
    if (currentRequest.oracles[address(msg.sender)] == 1) {
        //Oracle has voted
        currentRequest.oracles[address(msg.sender)] = 2;
        //Save answer
        currentRequest.answers.push(_value);
        //Aggregate all answers in the final value.
        currentRequest.value=currentRequest.value + _value/min_votes;
        //Increase count of oracles who have voted.
        currentRequest.votes_number++;
        //If enough oracles have voted
        if (currentRequest.votes_number >= min_votes) {
            //If enough oracles have voted, we have a valid random
            number.
            fulfillRandomness(_id);
        }
    }
}
```

Como podemos ver, se trata de un modelo más seguro y robusto que el oráculo centralizado. Con este modelo de oráculo descentralizado, aunque alguno de los nodos off-chain actúe de forma maliciosa o sea hackeado, mientras quede un único nodo que sí proporcione un número aleatorio, el valor final será aleatorio.

## 8.1 EJEMPLO RINKEBY

A continuación, se va a mostrar un ejemplo en la testnet de Rinkeby del correcto funcionamiento del modelo de oráculo centralizado.

En esta ocasión, tenemos un oráculo descentralizado que cuenta con tres nodos fuera de la cadena. Cada nodo, tiene una dirección de Ethereum distinta.

En primer lugar, desplegamos el contrato de loterías en la red de Ethereum (Ilustración 50).

```
Running 'scripts\deploy_lottery.py::main'...
Transaction sent: 0x021cfb05f920a84528c5d209cfd6203de614172a726700bc55aed8ffc3f8821
Gas price: 1.500000007 gwei Gas limit: 1202885 Nonce: 350
Lottery.constructor confirmed Block: 10962345 Gas used: 1093532 (90.91%)
Lottery deployed at: 0xdb351803acBcd68888466df028E0013EC3e434F

Deployed lottery contract
TESTNET RINKEBY
```

Ilustración 50 - Ejemplo Rinkeby descentralizado: despliegue del contrato (I)

Como podemos ver, el contrato de la aplicación de loterías ha sido desplegado en la dirección:

**0xdb351803acBcd68888466df028E0013EC3e434F**

Una vez ha sido desplegado el contrato, inicializamos tres nodos distintos para que empiecen a escuchar los eventos emitidos por el contrato.

```
Running 'scripts\node_offchain.py::main'...
TESTNET RINKEBY
Contract address: 0xdb351803acBcd68888466df028E0013EC3e434F
Select node id (1 - 2 - 3)
1
Account: 0xbB8147F66FaF71A5bA41E5bD074d6562bd9DB362

Running 'scripts\node_offchain.py::main'...
ID: 0
Random number 94827907280906097947651067372612763980544685073270665580441086923208562244768
Random number stored in the blockchain
```

Ilustración 51- Ejemplo Rinkeby descentralizado: nodo off-chain 1 (II)

```
TESTNET RINKEBY
Contract address: 0xdb351803acCbcd68888466df028E0013EC3e434F
Select node id (1 - 2 - 3)
2
Account: 0xc9A5E426bC9af443A1D3Cb0539ef96d17db8bea1

Running 'scripts\node_offchain.py::main'...
ID: 0
Random number 18257553844336689561105612355455024992923386006198841507091269374053961295715
Random number stored in the blockchain
□
```

*Ilustración 52 - Ejemplo Rinkeby descentralizado: nodo off-chain 2 (III)*

```
TESTNET RINKEBY
Contract address: 0xdb351803acCbcd68888466df028E0013EC3e434F
Select node id (1 - 2 - 3)
3
Account: 0xc9c68d75123Aa15dcFFcF52ad965bCDF0D3Ec216

Running 'scripts\node_offchain.py::main'...
ID: 0
Random number 89028703089413406498205731077205078152145007475621176950356856336246208888700
Random number stored in the blockchain
□
```

*Ilustración 53 - Ejemplo Rinkeby descentralizado: nodo off-chain 3 (IV)*

Como podemos ver en las imágenes anteriores (Ilustración 51, Ilustración 52 e Ilustración 53) los nodos tienen direcciones de Ethereum asociadas a distintas cuentas, pero todos los nodos están escuchando al mismo contrato.

De esta manera, estamos creando tres nodos independientes que más adelante cuando se emita un evento del tipo `NewRequest()` realizarán una consulta a distintas API e introducirán distintos datos en la cadena mediante una transacción independiente por cada nodo. Es decir, estamos generando un sistema descentralizado fuera de la cadena para introducir números aleatorios generados desde distintas fuentes y sin ningún punto de centralización.

Con el sistema totalmente inicializado, iniciamos una nueva lotería y añadimos tres jugadores a esta. Una vez los tres jugadores han comprado el ticket, cerramos la lotería. (Ilustración 54)

```

Press Enter to continue...
Loteria 1
=====
Transaction sent: 0xa8fb670854cf1b8e7d0f8a8e54354954a3b8afd392ddf0d81f5d7dce34a3ece9a
Gas price: 1.500000007 gwei Gas limit: 31572 Nonce: 351
Lottery.startLottery confirmed Block: 10962348 Gas used: 28702 (90.91%)

Lottery.startLottery confirmed Block: 10962348 Gas used: 28702 (90.91%)

The lottery is started
Transaction sent: 0xa858fcc5210fab1cb9b443b0dac62449149ac8745ccc78aecbe482dd98869ad2
Gas price: 1.500000007 gwei Gas limit: 97485 Nonce: 352
Lottery.buyTicket confirmed Block: 10962349 Gas used: 88623 (90.91%)

Lottery.buyTicket confirmed Block: 10962349 Gas used: 88623 (90.91%)

You entered de lottery!
Enter password for "Account2":
Transaction sent: 0xc43cd4e2ceba0258d1847ea32fde8db04113a08cb1facba8b967a6d1aa3d25e8
Gas price: 1.500000007 gwei Gas limit: 78675 Nonce: 31
Lottery.buyTicket confirmed Block: 10962352 Gas used: 71523 (90.91%)

Lottery.buyTicket confirmed Block: 10962352 Gas used: 71523 (90.91%)

You entered de lottery!
Enter password for "Account3":
Transaction sent: 0x6e7a8175239216a621fb02b4d9d09f93d2bfd12e87f09ee7770cb1d02668290a
Gas price: 1.500000007 gwei Gas limit: 78675 Nonce: 31
Lottery.buyTicket confirmed Block: 10962354 Gas used: 71523 (90.91%)

Lottery.buyTicket confirmed Block: 10962354 Gas used: 71523 (90.91%)

You entered de lottery!
Transaction sent: 0xc7e3837ed3c7099581ea1bb110b6eb8169d8d1a1194b1b9bebbf12c3834cb66b
Gas price: 1.500000008 gwei Gas limit: 288735 Nonce: 353
Lottery.endLottery confirmed Block: 10962355 Gas used: 262487 (90.91%)

Lottery.endLottery confirmed Block: 10962355 Gas used: 262487 (90.91%)

```

Ilustración 54 - Ejemplo Rinkeby descentralizado: transacciones (V)

Los participantes de esta lotería y el índice que les corresponde son:

- **Index 0: 0xbB8147F66FaF71A5bA41E5bD074d6562bd9DB362**
- **Index 1: 0xC9A5E426bC9af443A1D3Cb0539ef96d17db8bea1**
- **Index 2: 0xc9c68d75123Aa15dcFFcF52ad965bCDF0D3Ec216**

En el momento en el que se finaliza la lotería, el contrato Oracle (heredado por la aplicación de loterías) emite un nuevo evento `NewRequest()`. Cuando los nodos de fuera de la cadena escuchan este evento, realizan una petición HTTP del tipo GET a las distintas API que tienen asignadas.

Cuando reciben las respuestas de cada API, cada nodo tiene un número aleatorio generado por una API distinta. En ese momento, cada nodo realiza una transacción en la que llama al método `updateRequest()` y escribe dicho número en la cadena.

Si volvemos a observar las imágenes correspondientes a los nodos de fuera de la cadena (Ilustración 51, Ilustración 52 e Ilustración 53) podemos ver que los números aleatorios generados en cada nodo son:

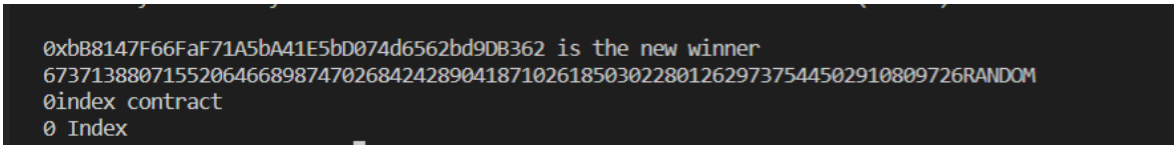
Nodo 1	94827907280906097947651067372612763980544685073270665580441086923208562244768
Nodo 2	18257553844336689561105612355455024992923386006198841507091269374053961295715
Nodo 3	89028703089413406498205731077205078152145007475621176950356856336246208888700

*Tabla 11 - Nodos off-chain y números aleatorios generados*

Cuando se han realizado las tres transacciones, se ha alcanzado el quorum mínimo del contrato Oracle para esta petición. Los tres datos, es decir, los números aleatorios han sido agregados en un único valor final. Como se explicaba previamente, para ello se realiza la media de los tres valores. En este caso, el valor final decidido es:

**67371388071552064668987470268424289041871026185030228012629737544502910809726**

Finalmente, tras haberse decidido un valor final, el contrato Oracle llama al método `fulfillRandomness()` y la aplicación de loterías decide un ganador. En este caso el ganador ha sido la dirección con índice 0 (Ilustración 55).



*Ilustración 55 - Ejemplo Rinkeby descentralizado: resultados (VI)*

Ahora, podemos comprobar mediante Etherscan que todas las transacciones han sido realizadas correctamente (Ilustración 56):

Txn Hash	Method	Block	Age	From	To	Value	Txn Fee
0x239275edabdec13864...	Update Request	10962356	15 hrs 8 mins ago	0xc9c68d75123aa15dcf...	IN 0xdb351803accbcd6888...	0 Ether	0.00021183
0x70032f8c84811d2bbc0...	Update Request	10962356	15 hrs 8 mins ago	0xc9a5e426bc9af443a1...	IN 0xdb351803accbcd6888...	0 Ether	0.00011669
0xe96ab5090546888b75...	Update Request	10962356	15 hrs 8 mins ago	0xbb8147f66faf71a5ba4...	IN 0xdb351803accbcd6888...	0 Ether	0.00019364
0xc7e3837ed3c7099581...	End Lottery	10962355	15 hrs 9 mins ago	0xbb8147f66faf71a5ba4...	IN 0xdb351803accbcd6888...	0 Ether	0.00039373
0x6e7a8175239216a621...	Buy Ticket	10962354	15 hrs 9 mins ago	0xc9c68d75123aa15dcf...	IN 0xdb351803accbcd6888...	0.00466308 Ether	0.00010728
0xc43cd4e2ceba0258d1...	Buy Ticket	10962352	15 hrs 9 mins ago	0xc9a5e426bc9af443a1...	IN 0xdb351803accbcd6888...	0.00466308 Ether	0.00010728
0xa858fcc5210fab1cb9b...	Buy Ticket	10962349	15 hrs 10 mins ago	0xbb8147f66faf71a5ba4...	IN 0xdb351803accbcd6888...	0.00466308 Ether	0.00013293
0x8fb670854cf1b8e7d0f...	Start Lottery	10962348	15 hrs 10 mins ago	0xbb8147f66faf71a5ba4...	IN 0xdb351803accbcd6888...	0 Ether	0.00004305
0x021cfb05f920a84528c...	0x80038055	10962345	15 hrs 11 mins ago	0xbb8147f66faf71a5ba4...	IN Contract Creation	0 Ether	0.00164029

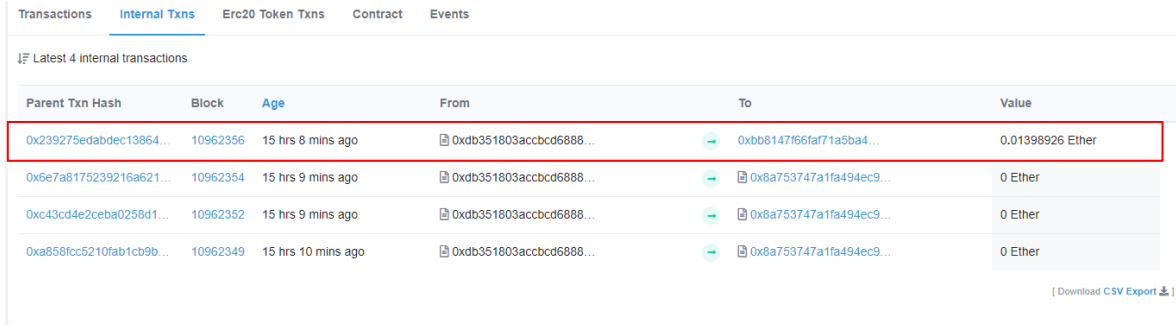
*Ilustración 56 - Ejemplo Rinkeby descentralizado: Etherscan (VII)*

Además, tal y como hicimos en el ejemplo del modelo centralizado, podemos comprobar que se ha pagado al ganador la cantidad adecuada. En este caso, han participado tres jugadores con una cantidad de 0.00466308 ETH. Esto significa, que el bote o cantidad total de premio es de 0.0139892 ETH.

El ganador de la lotería ha sido el jugador con index 0, que corresponde a la siguiente dirección de Ethereum:

- **Index 0: 0xbB8147F66FaF71A5bA41E5bD074d6562bd9DB362**

Si ahora comprobamos las transacciones internas del contrato en Etherscan, podemos observar cómo se ha pagado el premio a esa dirección de Ethereum.



Parent Txn Hash	Block	Age	From	To	Value
0x239275edabdec13864...	10962356	15 hrs 8 mins ago	0xdb351803accbcd6888...	0xbb8147f66fa71a5ba4...	0.01398926 Ether
0x6e7a8175239216a621...	10962354	15 hrs 9 mins ago	0xdb351803accbcd6888...	0x8a753747a1fa494ec9...	0 Ether
0xc43cd4e2ceba0258d1...	10962352	15 hrs 9 mins ago	0xdb351803accbcd6888...	0x8a753747a1fa494ec9...	0 Ether
0xa858fcc5210fab1cb9b...	10962349	15 hrs 10 mins ago	0xdb351803accbcd6888...	0x8a753747a1fa494ec9...	0 Ether

*Ilustración 57 - Ejemplo Rinkeby descentralizado: transacciones internas Etherscan (VIII)*

Toda la información de las transacciones se puede consultar en Etherscan mediante el siguiente enlace:

<https://rinkeby.etherscan.io/address/0xDb351803acCBcd68888466df028E0013EC3e434F>

Mediante este ejemplo, se ha mostrado el correcto funcionamiento del modelo de oráculo descentralizado.

## 8.2 SEGURIDAD DEL MODELO DESCENTRALIZADO

El modelo de oráculo descentralizado es un modelo más seguro que el modelo de oráculo centralizado. Al establecer un oráculo descentralizado y que no tiene ningún punto centralizado, sí que conseguimos mantener la descentralización, que es una de las principales ventajas de emplear blockchain y Smart Contracts. En este modelo de oráculo, al no existir ningún punto centralizado, nuestro Smart Contract no puede ser controlado por ningún organismo o entidad externa.

En el modelo de oráculo centralizado, hemos demostrado como en el caso de que el nodo externo sea atacado o actúe de forma maliciosa, puede modificar el resultado de la lotería a su antojo. A continuación, vamos a mostrar con el mismo ejemplo como si uno de los nodos decide actuar de forma maliciosa en el modelo de oráculo descentralizado, este no



consigue alterar el resultado de la lotería a su antojo. Esto se debe a que solo modifica uno de los datos que se introducen a la cadena, pero mientras uno de los otros nodos sí haya dado un número aleatorio, el valor final resultante de la agregación de todos los datos será un número aleatorio.

Como en todos los ejemplos, el primer paso será desplegar el contrato en la red de Rinkeby (Ilustración 58).

```
Running 'scripts\deploy_lottery.py::main'...
Transaction sent: 0x9aec5d9ca21682889e047a9dc613985cecbde258ef49e0d6adba32527195020
Gas price: 1.500000011 gwei Gas limit: 1202885 Nonce: 397
Lottery.constructor confirmed Block: 11249614 Gas used: 1093532 (90.91%)
Lottery deployed at: 0x36A6CDB9C35b79c1eAGd6697d26D546a81e40d53

Deployed lottery contract
TESTNET RINKEBY
Press Enter to continue...
```

Ilustración 58 - Ejemplo Rinkeby descentralizado seguro: desplegar contrato

Una vez desplegado el contrato, empezamos la lotería y añadimos tres jugadores:

```
Loteria 1
=====
Transaction sent: 0x0f203881d658a4843641d3f2af43bf041aac8ddcd42aa14fff33058bba261d55
Gas price: 1.500000011 gwei Gas limit: 31572 Nonce: 398
Lottery.startLottery confirmed Block: 11249615 Gas used: 28702 (90.91%)

Lottery.startLottery confirmed Block: 11249615 Gas used: 28702 (90.91%)

The lottery is started
Transaction sent: 0x92571f47ccd6ebccf432b85c6100392f0772f689661fa99a14bfb2e6a770d3b0
Gas price: 1.500000011 gwei Gas limit: 97485 Nonce: 399
Lottery.buyTicket confirmed Block: 11249616 Gas used: 88623 (90.91%)

Lottery.buyTicket confirmed Block: 11249616 Gas used: 88623 (90.91%)

You entered de lottery!
Enter password for "Account2":
Transaction sent: 0xb7fa414372bafb9712a07e7cda0bf54d0895b426ed2d1518729168f4ecc86a42
Gas price: 1.500000011 gwei Gas limit: 78675 Nonce: 47
Lottery.buyTicket confirmed Block: 11249621 Gas used: 71523 (90.91%)

Lottery.buyTicket confirmed Block: 11249621 Gas used: 71523 (90.91%)

You entered de lottery!
Enter password for "Account3":
Transaction sent: 0x4729eaf3912630a8a0f85ef82cffcc6d36e79b625ccb14c1f0099b021bdd08a6
Gas price: 1.500000012 gwei Gas limit: 78675 Nonce: 47
Lottery.buyTicket confirmed Block: 11249624 Gas used: 71523 (90.91%)

Lottery.buyTicket confirmed Block: 11249624 Gas used: 71523 (90.91%)

You entered de lottery!
```

Ilustración 59 - Ejemplo Rinkeby descentralizado seguro: iniciar lotería y compra de tickets

En este momento acabaremos la lotería y esperaremos la respuesta de los nodos externos:

```
Transaction sent: 0x011098d100d2acff1523c75e0e9d367a135d76d5107b23267d34f1fa9d279603
Gas price: 1.500000012 gwei Gas limit: 288735 Nonce: 400
Lottery.endLottery confirmed Block: 11249625 Gas used: 262487 (90.91%)

Lottery.endLottery confirmed Block: 11249625 Gas used: 262487 (90.91%)
```

*Ilustración 60 - Ejemplo Rinkeby descentralizado seguro: acabar lotería*

Al igual que en el ejemplo de seguridad del modelo centralizado, el primer nodo en lugar de un número aleatorio introducirá el 12, buscando que gane el jugador con índice 0. ()

```
Contract address: 0x36A6CDb9C35b79c1eA6d6697d26D546a81e40d53
Select node id (1 - 2 - 3)
1
Account: 0xbB8147F66FaF71A5bA41E5bD074d6562bd9DB362

Running 'scripts\node_offchain.py::main'...
ID: 0
NODO MALICIOSO
Random number 12
Random number stored in the blockchain
█
```

*Ilustración 61- Ejemplo Rinkeby descentralizado: nodo externo 1*

Por otra parte, el resto de los nodos sí que introducen un número aleatorio (Ilustración 61 e Ilustración 62):

```
TESTNET RINKEBY
Contract address: 0x36A6CDb9C35b79c1eA6d6697d26D546a81e40d53
Select node id (1 - 2 - 3)
2
Account: 0xc9A5E426bC9af443A1D3Cb0539ef96d17db8bea1

Running 'scripts\node_offchain.py::main'...
ID: 0
Random number 26641649202618897933134310168063878827676750136812794202952596693112407017979
Random number stored in the blockchain
█
```

*Ilustración 62 - Ejemplo Rinkeby descentralizado: nodo externo 2*

```
TESTNET RINKEBY
Contract address: 0x36A6CDb9C35b79c1eA6d6697d26D546a81e40d53
Select node id (1 - 2 - 3)
3
Account: 0xc9c68d75123Aa15dcFFcF52ad965bCDF0D3Ec216

Running 'scripts\node_offchain.py::main'...
ID: 0
Random number 93410616969236472564480627329650418730594225296282069859488307798198695949512
Random number stored in the blockchain
█
```

*Ilustración 63- Ejemplo Rinkeby descentralizado: nodo externo 3*

Al agregar los tres datos el resultado es un número aleatorio y el ganador en este caso no es el jugador con índice 0 sino el jugador con índice 1.

```
0xC9A5E426bC9af443A1D3Cb0539ef96d17db8bea1 is the new winner
40017422057285123499204979165904765852756991811031621354146968163770367655833RANDOM
1index contract
1 Index
Press Enter to continue...█
```

Como podemos comprobar, el atacante no consigue alterar el resultado a su favor.

Aunque el modelo de oráculo descentralizado se trata de un modelo mucho más seguro que el oráculo centralizado, lo cierto es que no es totalmente seguro. Si un atacante tomase el control de todos los nodos que se encuentran fuera de la cadena, podría dar un valor a su antojo y cuando dichos datos se agregasen la media daría como resultado el número que el atacante desea.

A continuación, se muestra un ejemplo en el que el atacante ha conseguido tomar el control de todos los nodos externos (en este caso tres) y en lugar de un número aleatorio, ha proporcionado el 12 en todos. De este modo, el valor final al agregar los datos obteniendo la media será 12. Como el atacante conocía el número de jugadores y el index en el que se encuentra la dirección de la cuenta de cada jugador, ha conseguido forzar la lotería para que gane el jugador con la dirección de cuenta asociada al index 0.

En primer lugar, se despliega el contrato de lotería (Ilustración 64). Luego, se inicia un nuevo juego y se añaden tres jugadores (Ilustración 65).

```
Running 'scripts\deploy_lottery.py::main'...
Transaction sent: 0x6570036a34022d82c6e6a306f0035de0a8c38f4be0eb22f2ec0fe5ec7a875335
Gas price: 1.381396626 gwei Gas limit: 1202885 Nonce: 362
Lottery.constructor confirmed Block: 11247435 Gas used: 1093532 (90.91%)
Lottery deployed at: 0xf1A269403123C7a4dD875f1c6Ac40f2857402b9c

Deployed lottery contract
TESTNET RINKEBY
Press Enter to continue...
```

*Ilustración 64 - Ejemplo Rinkeby descentralizado inseguro: despliegue de contrato*

```
Loteria 1
=====
Transaction sent: 0x12cf3899926311e59ce7d6e8a2a8baef3cc54cc4429b61574fab80833fc1bb2f
Gas price: 1.358646716 gwei Gas limit: 31572 Nonce: 363
Lottery.startLottery confirmed Block: 11247439 Gas used: 28702 (90.91%)

Lottery.startLottery confirmed Block: 11247439 Gas used: 28702 (90.91%)

The lottery is started
Transaction sent: 0x569beff6b1c753cdf56553ac11113cc52506c63bd4e9f6ad1275c87aaff36a85
Gas price: 1.354346985 gwei Gas limit: 97485 Nonce: 364
Lottery.buyTicket confirmed Block: 11247441 Gas used: 88623 (90.91%)

Lottery.buyTicket confirmed Block: 11247441 Gas used: 88623 (90.91%)

You entered de lottery!
Enter password for "Account2":
Transaction sent: 0x8974d1bd130fd0937cdf97f1672a46d325535cb9ab6a925525da1132b32b6d5a
Gas price: 1.406722424 gwei Gas limit: 78675 Nonce: 33
Lottery.buyTicket confirmed Block: 11247443 Gas used: 71523 (90.91%)

Lottery.buyTicket confirmed Block: 11247443 Gas used: 71523 (90.91%)

You entered de lottery!
Enter password for "Account3":
Transaction sent: 0x21f596fefc5346b6556bb883cd269ac4daddf51ec5a0b6ba420d04b0f3833c91
Gas price: 1.384782695 gwei Gas limit: 78675 Nonce: 33
Lottery.buyTicket confirmed Block: 11247445 Gas used: 71523 (90.91%)

Lottery.buyTicket confirmed Block: 11247445 Gas used: 71523 (90.91%)

You entered de lottery!
```

*Ilustración 65 - - Ejemplo Rinkeby descentralizado inseguro: inicialización lotería y compra de tickets*

Se finaliza la lotería y se espera a que los nodos externos introduzcan el dato (Ilustración 66):

```
Transaction sent: 0x1c3fb2e0425eb37d19fa2046168a7234810a2843e2251c76252653d0f486a099
Gas price: 1.363295915 gwei Gas limit: 288735 Nonce: 365
Lottery.endLottery confirmed Block: 11247446 Gas used: 262487 (90.91%)

Lottery.endLottery confirmed Block: 11247446 Gas used: 262487 (90.91%)
```

*Ilustración 66- Ejemplo Rinkeby descentralizado inseguro: finalizar lotería*

En esta ocasión, TODOS los nodos externos tienen intenciones maliciosas y todos introducen el mismo dato para forzar que al agregarlos el resultado final sea este. El número que introducen es el 12 para forzar que gane el jugador con índice 0. (Ilustración 67)

```
TESTNET RINKEBY
Contract address: 0xf1A269403123C7a4dD875f1c6Ac40f2857402b9c
Select node id (1 - 2 - 3)
1
Account: 0xbB8147F66FaF71A5bA41E5bD074d6562bd9DB362

Running 'scripts\node_offchain.py::main'...
ID: 0
Random number 12
Random number stored in the blockchain
TESTNET RINKEBY
Contract address: 0xf1A269403123C7a4dD875f1c6Ac40f2857402b9c
Select node id (1 - 2 - 3)
2
Account: 0xc9A5E426bC9af443A1D3Cb0539ef96d17db8bea1

Running 'scripts\node_offchain.py::main'...
ID: 0
Random number 12
Random number stored in the blockchain
TESTNET RINKEBY
Contract address: 0xf1A269403123C7a4dD875f1c6Ac40f2857402b9c
Select node id (1 - 2 - 3)
3
Account: 0xc9c68d75123Aa15dcFFcF52ad965bCDF0D3Ec216

Running 'scripts\node_offchain.py::main'...
ID: 0
Random number 12
Random number stored in the blockchain
```

*Ilustración 67- - Ejemplo Rinkeby descentralizado inseguro:nodos externos*

Como es de esperar, el número aleatorio resultante es 12 y el jugador con índice 0 ha ganado la lotería.

```
0xbB8147F66FaF71A5bA41E5bD074d6562bd9DB362 is the new winner
12RANDOM
0index contract
0 Index
Press Enter to continue...|
```

*Ilustración 68 - - Ejemplo Rinkeby descentralizado inseguro: Resultados lotería*

Con este ejemplo podemos ver que, aunque el modelo de oráculo descentralizado es un modelo bastante más seguro que la versión centralizada, aún no es un modelo totalmente seguro.

Cabe destacar que al necesitar que todos los nodos introduzcan el mismo dato o se pongan de acuerdo para forzar un resultado este sistema es más seguro cuantos más nodos externos introducen un dato. Recordemos que con que un solo nodo externo introduzca un número aleatorio, el resultado será aleatorio.

## Capítulo 9. CHAINLINK VRF

En primer lugar, debemos de entender que es y cómo funciona Chainlink. Se trata de uno de los mayores proyectos de oráculo descentralizado que se ejecuta sobre la blockchain de Ethereum. Actualmente, es el más usado y se ha convertido en un puente para introducir datos desde fuera del blockchain a aplicaciones descentralizadas (DApps) y al ecosistema DeFi en esta blockchain.

La idea detrás de Chainlink surge en 2014 con el proyecto SmartContract.com cuando los investigadores Steve Ellis, Arl Juels y Sergey Nazarov buscaban crear una infraestructura descentralizada que permitiera conectar los eventos del mundo real con las blockchain públicas. En el año 2017 se publica finalmente el whitepaper de Chainlink en el que se describía un protocolo de oráculo descentralizado ejecutado sobre la red Ethereum. Finalmente, en el año 2019 se lanza con éxito la red principal de Chainlink tras numerosas pruebas.

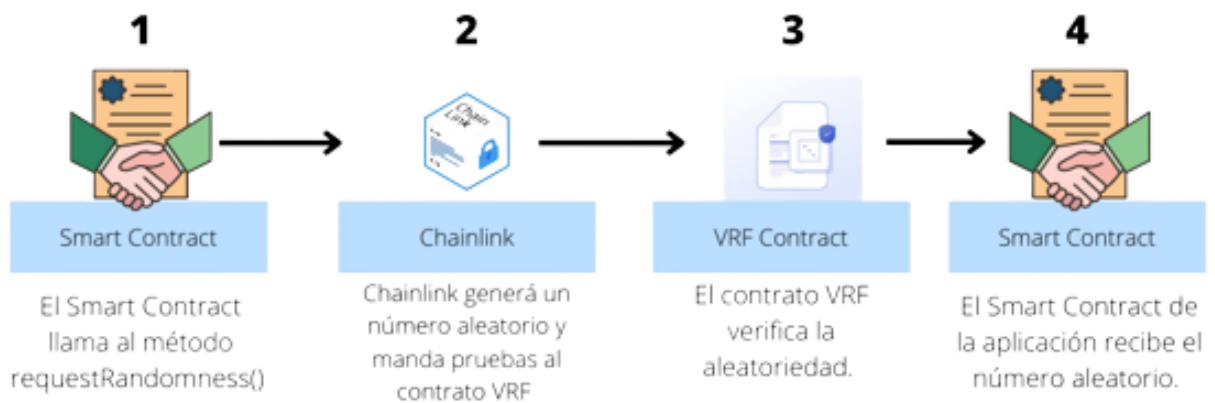
El funcionamiento de Chainlink se apoya principalmente en una red de nodos que ejecutan un programa para alimentar Smart Contracts que se encuentran en la red de Ethereum con datos provenientes de eventos del mundo real. Para asegurar que los datos que provistos por los nodos son correctos, Chainlink solicita la misma información a muchos nodos seleccionados de forma aleatoria y genera consenso entre todas las respuestas obtenidas decantándose por la que mayor número de votos tenga. Además, Chainlink emplea un sistema de incentivos basados en su propio token llamado LINK. El LINK es un token ERC-20 que funciona sobre la red Ethereum y sirve como moneda para pagar a los nodos a modo de incentivo para que estos provean los mejores datos posibles.

Por otra parte, dentro de la red de Chainlink existen varios Smart Contract en la red de Ethereum encargados de asegurarse que la información recibida por los nodos es correcta y válida. Estos contratos son: un contrato de reputación encargado de valorar los nodos de la red con una reputación en función de la calidad de la información que provee; un contrato

de coincidencia de pedidos que se encarga de almacenar la información de nodos, servicios, tipos de información solicitada, parámetros y fuentes de datos; y, por último, un contrato de agregación cuya misión es agregar todas las respuestas de los distintos nodos en una única respuesta final.

Dentro de los servicios ofrecidos por Chainlink se encuentra Chainlink VRF (Verifiable Random Function). Este servicio es uno de los más demandados y se trata de una red de oráculos descentralizada encargada de proporcionar aleatoriedad criptográficamente asegurada, es decir, es un proveedor de números aleatorios para Smart Contracts.

El funcionamiento de Chainlink VRF es el siguiente (Ilustración 69):



*Ilustración 69 - Funcionamiento Chainlink VRF*

A continuación, vamos a explicar las modificaciones que hemos realizado al contrato de loterías para poder implementar y usar Chainlink VRF.

En primer lugar, para emplear Chainlink VRF nuestro contrato deberá heredar el contrato `VRFConsumerBase` para ser compatible con este servicio. Además, se deben declarar dos nuevos atributos: `fee` y `keyHash`.

- `fee`: Los servicios de Chainlink cobran una pequeña tasa en su token llamado LINK. En esta variable se almacena la cantidad de LINK necesario para realizar una petición.



- **keyHash:** Se trata de una clave que sirve como un identificador único para la saber que tareas se deben realizar.

Por otra parte, el contrato `VRFConsumerBase` necesita dos argumentos en su constructor:

- `_vrfCoordinator`: Dirección del contrato que comprueba si el número aleatorio generado es realmente aleatorio.
- `_link`: La dirección del token LINK.

Todos estos datos se pueden obtener en la documentación de Chainlink en el siguiente enlace: [Supported Networks | Chainlink Documentation](#)

```
contract Lottery is Ownable, VRFConsumerBase {
  constructor(
    address _priceFeedAddress,
    address _vrfCoordinator,
    address _link,
    uint256 _fee,
    bytes32 _keyhash
  ) public VRFConsumerBase(_vrfCoordinator, _link) {
    usdEntryFee = 5 * (10**18); // 5$
    ethUsdPriceFeed = AggregatorV3Interface(_priceFeedAddress);
    state = STATE.CLOSED;
    fee = _fee;
    keyhash = _keyhash;
  }
}
```

Una vez hemos realizado estos cambios en el contrato de loterías, el siguiente paso será modificar la función `endLottery()`. Ahora, una vez acabada la lotería, deberemos solicitar un número aleatorio mediante Chainlink VRF. Para ello, usaremos el método `requestRandomness` al que deberemos llamar con las variables `keyHash` y `fee` como parámetros. Cabe destacar que para realizar este paso es necesario haber añadido LINK al contrato para poder realizar esta operación ya que como hemos comentado antes, habrá que pagar una tasa para realizar esta transacción.

```
function endLottery() public onlyOwner {
    state = STATE.PROCESSING;
    bytes32 requestId = requestRandomness(keyhash, fee);
}
```

Una vez se ha generado un número aleatorio, automáticamente se llamará a la función fulfillRandomness() que en nuestro caso no tendrá modificaciones respecto a los modelos anteriores.

```
function fulfillRandomness(uint256 _requestId) internal override {
    require(state == STATE.PROCESSING, "Incorrect State");
    random_number = requests[_requestId].value;
    require(random_number > 0, "Random not found");
    random_index = random_number % lottery_players.length;
    last_winner = lottery_players[random_index];
    last_winner.transfer(address(this).balance);
    //Reset
    lottery_players = new address payable[](0);
    state = STATE.CLOSED;
}
```

## 9.1 EJEMPLO RINKEBY

A continuación, se va a mostrar un ejemplo en la testnet de Rinkeby del correcto funcionamiento de la implementación del servicio Chainlink VRF.

Lo primero será desplegar el Smart Contract de la aplicación de loterías en la red Rinkeby (Ilustración 70).

```
Running 'scripts\deploy_lottery.py::main'...
Transaction sent: 0x35087de60d52848521c074894cdf7c54dea8172dc475e7af580fa781a1af84e2
Gas price: 2.653927746 gwei Gas limit: 1007832 Nonce: 313
Lottery.constructor confirmed Block: 10676723 Gas used: 916211 (90.91%)
Lottery deployed at: 0x7993037BD9c30b41e70ecAb36eD9F0DA53B3D169

Deployed lottery contract
Press Enter to continue...
```

*Ilustración 70 - Ejemplo Rinkeby Chainlink VRF: desplegar contrato*

Como podemos ver, el contrato de la aplicación de loterías ha sido desplegado en la dirección:

**0x7993037BD9c30b41e70ecAb36eD9F0DA53B3D169**

El siguiente paso será empezar una nueva lotería y añadir tres jugadores (Ilustración 71).

```
Loteria 1
=====
Transaction sent: 0x4f82dfed271f64e33916b22310b352eee78fbcf5f4a860456e269832bf750a31
Gas price: 2.582378631 gwei Gas limit: 31548 Nonce: 314
Lottery.startLottery confirmed Block: 10676736 Gas used: 28680 (90.91%)

Lottery.startLottery confirmed Block: 10676736 Gas used: 28680 (90.91%)

The lottery is started
Transaction sent: 0xbf481c72d174b8cd112ec2f746b1f70a4aeefa266d62dfe379429aa5ba0e0512
Gas price: 2.54974714 gwei Gas limit: 97458 Nonce: 315
Lottery.buyTicket confirmed Block: 10676737 Gas used: 88599 (90.91%)

Lottery.buyTicket confirmed Block: 10676737 Gas used: 88599 (90.91%)

You entered de lottery!
Enter password for "Account2":
Transaction sent: 0x218aba400044673edc8ee3d930e2d85295b8b292c8200935835d5ec9cad1202e
Gas price: 2.54180064 gwei Gas limit: 78648 Nonce: 24
Lottery.buyTicket confirmed Block: 10676743 Gas used: 71499 (90.91%)

Lottery.buyTicket confirmed Block: 10676743 Gas used: 71499 (90.91%)

You entered de lottery!
Enter password for "Account3":
Transaction sent: 0xa13b72dcdeec92f12c7b7d32c038adbb940e3837e38599f097e2f5f82f83855c
Gas price: 2.667651144 gwei Gas limit: 78648 Nonce: 25
Lottery.buyTicket confirmed Block: 10676750 Gas used: 71499 (90.91%)

Lottery.buyTicket confirmed Block: 10676750 Gas used: 71499 (90.91%)
```

*Ilustración 71 - Ejemplo Rinkeby Chainlink VRF: nueva loterías y entradas de jugadores*

Los participantes y el índice que les corresponde son:

**Index 0: 0xbB8147F66FaF71A5bA41E5bD074d6562bd9DB362**

**Index 1: 0xC9A5E426bC9af443A1D3Cb0539ef96d17db8bea1**

**Index 2: 0xc9c68d75123Aa15dcFFcF52ad965bCDF0D3Ec216**

Una vez todos los jugadores han comprado sus tickets, deberemos financiar el contrato con LINK para que al finalizar la lotería se pueda realizar la petición al servicio de Chainlink VRF (Ilustración 72).

```
Transaction sent: 0x58597ae07a0f64f671ff901ac185eb6be70d983b562cca9d74f576b89e94eac6
Gas price: 2.553457637 gwei Gas limit: 56992 Nonce: 316
LinkToken.transfer confirmed Block: 10676752 Gas used: 51811 (90.91%)

LinkToken.transfer confirmed Block: 10676752 Gas used: 51811 (90.91%)

Fund contract!
LinkToken.transfer confirmed Block: 10676752 Gas used: 51811 (90.91%)
```

*Ilustración 72 - Ejemplo Rinkeby Chainlink VRF: Financiar contrato con LINK*

Por último, finalizamos la lotería y esperamos los resultados (Ilustración 73).

```
LinkToken.transfer confirmed Block: 10676752 Gas used: 51811 (90.91%)

Transaction sent: 0xc821798aa305089eccf2a2773e8e322c427da29b4906b03d6fed7f8dd6987d12
Gas price: 2.522264265 gwei Gas limit: 172892 Nonce: 317
Lottery.endLottery confirmed Block: 10676755 Gas used: 152375 (88.13%)

Lottery.endLottery confirmed Block: 10676755 Gas used: 152375 (88.13%)

0xC9A5E426bC9af443A1D3Cb0539ef96d17db8bea1 is the new winner
91833118767026369558466131950777074928392035471710903911381142133204196239560RANDOM
1index contract
1 Index
```

*Ilustración 73 - Ejemplo Rinkeby Chainlink VRF: finalizar lotería*

El número aleatorio proporcionado por el servicio de Chainlink VRF es:

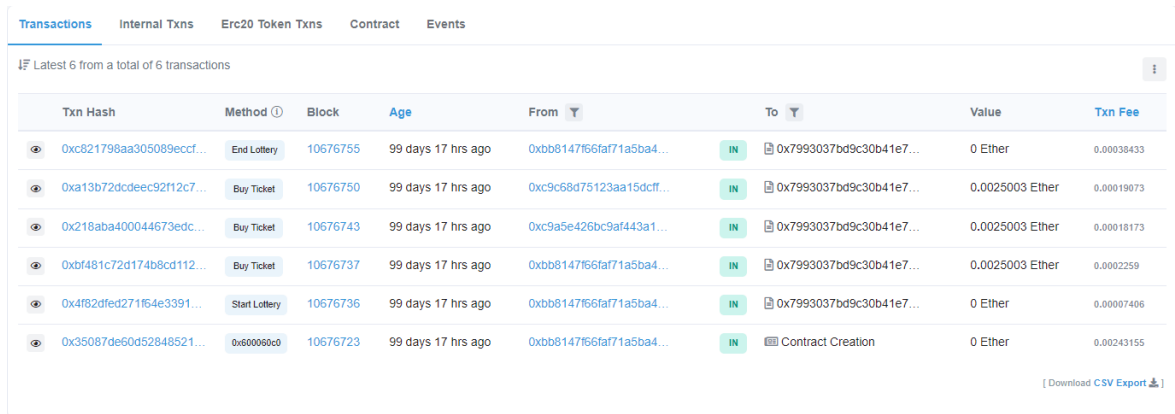
**91833118767026369558466131950777074928392035471710903911381142133204196239560**

Al realizar la operación módulo 3 obtenemos que el índice del ganador de la lotería es el 1. Si recordamos dicho índice pertenecía a la siguiente cuenta:

**Index 1: 0xC9A5E426bC9af443A1D3Cb0539ef96d17db8bea1**

Podemos comprobar en Etherscan que todas las transacciones se han realizado de forma correcta.

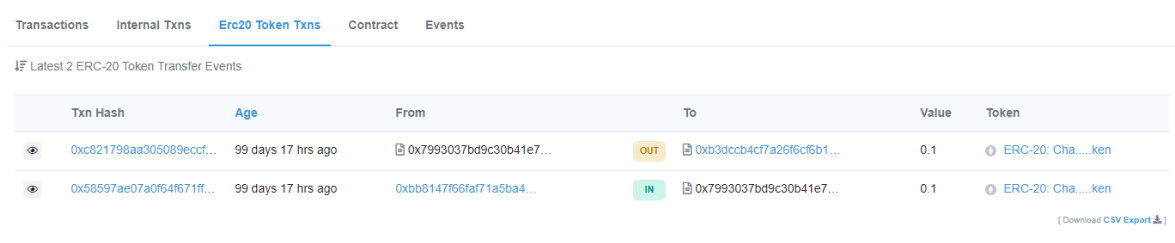
En la siguiente imagen podemos ver como se han comprado tres tickets de lotería cada uno por un valor de 0.0025003 Ether. Esto significa que el premio final es de de 0.00750091 Ether (Ilustración 74).



Txn Hash	Method	Block	Age	From	To	Value	Txn Fee
0xc821798aa305089eccf...	End Lottery	10676755	99 days 17 hrs ago	0xbb8147f66faf71a5ba4...	IN 0x7993037bd9c30b41e7...	0 Ether	0.00038433
0xa13b72dcdec92f12c7...	Buy Ticket	10676750	99 days 17 hrs ago	0xc9c68d75123aa15dcff...	IN 0x7993037bd9c30b41e7...	0.0025003 Ether	0.00019873
0x218aba40044673edc...	Buy Ticket	10676743	99 days 17 hrs ago	0xc9a5e426bc9af443a1...	IN 0x7993037bd9c30b41e7...	0.0025003 Ether	0.00018173
0xbf481c72d174b8cd112...	Buy Ticket	10676737	99 days 17 hrs ago	0xbb8147f66faf71a5ba4...	IN 0x7993037bd9c30b41e7...	0.0025003 Ether	0.0002259
0x4f82dfed271f54e3391...	Start Lottery	10676736	99 days 17 hrs ago	0xbb8147f66faf71a5ba4...	IN 0x7993037bd9c30b41e7...	0 Ether	0.00007406
0x35087de60d52848521...	0x600096c0	10676723	99 days 17 hrs ago	0xbb8147f66faf71a5ba4...	IN Contract Creation	0 Ether	0.00243155

Ilustración 74 - Ejemplo Rinkeby Chainlink VRF: Etherscan (I)

Además, si observamos las transacciones de tokens ERC-20 realizadas, podemos ver como se ha financiado el contrato con 0.1 LINK que posteriormente se han pagado cuando hemos realizado la petición de un número aleatorio a Chainlink VRF (Ilustración 75).



Txn Hash	Age	From	To	Value	Token
0xc821798aa305089eccf...	99 days 17 hrs ago	0x7993037bd9c30b41e7...	OUT 0xb30ccb4c7a26f6cfb1...	0.1	ERC-20: Cha....ken
0x58597ae07a0f64f71ff...	99 days 17 hrs ago	0xbb8147f66faf71a5ba4...	IN 0x7993037bd9c30b41e7...	0.1	ERC-20: Cha....ken

Ilustración 75 - Ejemplo Rinkeby Chainlink VRF: Etherscan (II)

Podemos ver como se le ha pagado el premio total de 0.00750091 Ether a la cuenta del ganador cuya dirección es **0xC9A5E426bC9af443A1D3Cb0539ef96d17db8bea1**. (Ilustración 76).

Parent Txn Hash	Block	Age	From	To	Value
0x722793bd4324f032f69...	10676766	99 days 17 hrs ago	0x7993037bd9c30b41e7...	0xc9a5e426bc9af443a1...	0.00750091 Ether
0x722793bd4324f032f69...	10676766	99 days 17 hrs ago	0xb3dccb4cf7a26fc6b1...	0x7993037bd9c30b41e7...	0 Ether
0xc821798aa305089eccf...	10676755	99 days 17 hrs ago	0x7993037bd9c30b41e7...	0x01be23585060835e02...	0 Ether
0xa13b72dcdec92f12c7...	10676750	99 days 17 hrs ago	0x7993037bd9c30b41e7...	0x8a753747a1fa494ec9...	0 Ether
0x218aba400044673edc...	10676743	99 days 17 hrs ago	0x7993037bd9c30b41e7...	0x8a753747a1fa494ec9...	0 Ether
0xbf481c72d174b8cd112...	10676737	99 days 17 hrs ago	0x7993037bd9c30b41e7...	0x8a753747a1fa494ec9...	0 Ether

[ Download CSV Export ]

*Ilustración 76 - Ejemplo Rinkeby Chainlink VRF: Etherscan (III)*

Como podemos comprobar, todas las transacciones han sido realizadas correctamente.

Todas las transacciones se pueden consultar en el siguiente enlace:

<https://rinkeby.etherscan.io/address/0x7993037BD9c30b41e70ecAb36eD9F0DA53B3D169>

Con este ejemplo hemos podido comprobar el correcto funcionamiento e implementación del servicio de Chainlink VRF en nuestra aplicación de loterías.

## Capítulo 10. CONCLUSIONES Y TRABAJOS

### FUTUROS

En este proyecto se ha desarrollado una sencilla aplicación de loterías para poder programar, probar y contrastar distintos métodos para introducir datos desde fuera de la cadena a un Smart Contract que se encuentra en la blockchain.

Como se ha explicado previamente se ha empleado una aplicación de loterías porque un Smart Contract no puede generar números aleatorios. Esto se debe a que se trata de un sistema determinista y cada nodo debe poder devolver un mismo resultado dada una misma entrada de datos. Si cada nodo de la red genera un número aleatorio, se obtendría un resultado distinto y no se llegaría a un consenso. Es decir, el dato que vamos a necesitar introducir desde fuera de la cadena es un número aleatorio. Esta aplicación sirve como un ejemplo sencillo para poder emplear distintos métodos para introducir dicho dato desde fuera de la cadena.

Para introducir dicho dato externo, se plantean 3 opciones diferentes:

- Oráculo centralizado
- Oráculo descentralizado
- Chainlink VRF.

Una vez se han programado y testeado las tres opciones, vamos a realizar una comparativa para ver las ventajas e inconvenientes de emplear un modelo u otro.

En primer lugar, el modelo de oráculo centralizado ofrece la solución más sencilla y fácil de implementar. Además, al realizar una única transacción para introducir el dato en el Smart Contract, se trata de una solución que tendrá un coste mínimo. Debemos recordar

que las transacciones que modifican la cadena tienen un coste asociado debido a las “gas fees” o tasas de gas que la red de Ethereum tiene.

Por otra parte, la solución de emplear un oráculo centralizado es una mala práctica y deriva en un fallo de seguridad. Esto se debe a que, al tener un único punto de fallo centralizado, perdemos las ventajas de estar empleando Smart Contract y Blockchain que son un sistema descentralizado. Al emplear un oráculo centralizado, el nodo que se encuentra en el exterior de la cadena, aunque en un principio se trate de un nodo confiable, puede ser hackeado o tener intenciones maliciosas. Esto resulta en que nuestra aplicación de loterías puede ser controlada por dicho nodo, quien puede seleccionar un ganador a su antojo simplemente consultando el index en el que se ha almacenado la dirección de la cuenta de un participante e introduciendo un número que al realizar la operación módulo con el número de jugadores resulte en dicho index. Es decir, el Smart Contract es controlado por el nodo centralizado que se encuentra en el exterior de la cadena.

El segundo método empleado se trata del modelo de oráculo descentralizado. Este modelo resuelve el problema de seguridad que genera el modelo centralizado, pero a cambio aumenta la complejidad y el coste. Cuando empleamos un modelo de oráculo descentralizado, no permitimos que exista ningún punto centralizado en nuestra aplicación de loterías, por lo que no se puede tomar el control de nuestro Smart Contract.

Aunque se trata de un modelo más seguro que el anterior, aumenta la complejidad ya que al tener varios nodos fuera de la cadena introduciendo datos, debemos encontrar un modo de agregar todos los datos en un único valor final verdadero. En el caso de los números aleatorios, esto puede realizarse calculando la media de todos los números aleatorios proporcionados. De este modo, con que un solo nodo del exterior haya proporcionado un número aleatorio, el resultado final será aleatorio. Por otra parte, para otro tipo de datos como puede ser la consulta de precio USD/ETH, se pueden realizar otras operaciones para agregar los distintos datos que proporcionan los nodos desde el exterior como hallar la mediana.



El modelo de oráculo descentralizado resulta más costoso que el modelo centralizado. Esto se debe a que cada nodo que se encuentra en el exterior realizará una transacción para introducir su dato a la cadena. Esto significa que deberemos pagar una tasa de gas o “gas fee” por cada nodo externo. Además, en nuestro caso a mayor número de nodos externos, mayor seguridad podremos obtener ya que con que un solo nodo proporcione como dato un número aleatorio, el valor final tras agregar todos los datos será aleatorio y, por lo tanto, para atacar el Smart Contract y seleccionar un ganador se deberá tomar el control de todos los nodos externos lo que es muy improbable cuando existe un gran número de estos.

Por último, Chainlink que es una de las mayores redes de oráculos descentralizados que existe en la actualidad, ofrece un servicio llamado Verifiable Random Function (VRF). Este servicio permite utilizar funciones aleatorias verificables para generar aleatoriedad. Chainlink es totalmente descentralizado y se trata de uno de los sistemas de oráculos que se consideran más seguros en el mundo. Para emplear Chainlink deberemos de pagar tasas con su token ERC-20 llamado LINK. Cada transacción o petición que realizamos en su red tendrá un coste asociado y antes de realizarla deberemos financiar nuestro contrato con este token.

A continuación, se muestra una tabla resumen y comparativa de los diferentes modelos. En esta tabla, se han incluido tres características principales para evaluar cada modelo:

- Dificultad de implementación
- Coste
- Seguridad

Modelo	Dificultad implementación	Coste	Seguridad
Oráculo centralizado	Sencillo	Poco	Inseguro
Oráculo descentralizado	Complejo	Depende del número de nodos externos	Seguro
Chainlink VRF	Muy sencillo	Fijo - LINK	Muy seguro

*Tabla 12 - Comparativa distintos modelos implementados*

## Capítulo 11. BIBLIOGRAFÍA

- [1] *Blockchain Explained*. (2022, 24 junio). Investopedia.  
<https://www.investopedia.com/terms/b/blockchain.asp>
- [2] C. (2021, 8 diciembre). *Hybrid Smart Contracts Explained*. Chainlink Blog.  
<https://blog.chain.link/hybrid-smart-contracts-explained/>
- [3] Collins, P. (2022, 4 febrero). What is a blockchain oracle? What is the oracle problem? Why can't blockchains make API calls? This is everything you need to know about off-chain data | Better Programming. Medium. [https://betterprogramming.pub/what-is-a-blockchain-oracle-f5ccab8dbd72?source=friends\\_link&sk=d921a38466df8a9176ed8dd767d8c77d](https://betterprogramming.pub/what-is-a-blockchain-oracle-f5ccab8dbd72?source=friends_link&sk=d921a38466df8a9176ed8dd767d8c77d)
- [4] The crypto wallet for Defi, Web3 Dapps and NFTs | MetaMask. (2016). Metamask.  
<https://metamask.io/>
- [5] Ethereum. (2015a). *Ethereum Whitepaper*. Ethereum.Org.  
<https://ethereum.org/en/whitepaper/>
- [6] Badr, B., Horrocks, R., & Wu, X. (2018). *Blockchain By Example: A developer's guide to creating decentralized applications using Bitcoin, Ethereum, and Hyperledger* (English Edition) (1.<sup>a</sup> ed.). Packt Publishing.
- [7] *Frequently Asked Questions*. (s. f.). Infura. <https://infura.io/faq/general>
- [8] *Introduction — Web3.py 5.29.2 documentation*. (2022). Web3.Py.  
<https://web3py.readthedocs.io/en/stable/>
- [9] *Rinkeby: Network Dashboard*. (2017). Rinkeby. <https://www.rinkeby.io/#stats>

- [10] *Why Blockchain's Smart Contracts Aren't Ready for the Business World.* (s. f.). Gartner. <https://www.gartner.com/smarterwithgartner/why-blockchains-smart-contracts-arent-ready-for-the-business-world>
- [11] Ethereum. (2015b). *¿Qué es Ethereum?* ethereum.org. <https://ethereum.org/es/what-is-ethereum/>
- [12] *The DAO: What Was the DAO Hack?* (s. f.). Gemini. <https://www.gemini.com/cryptopedia/the-dao-hack-makerdao#section-origins-of-the-dao>
- [13] Brownworth, A. (s. f.). *Blockchain Demo.* Blockchain Demo. <https://andersbrownworth.com/blockchain/>
- [14] *Bitcoin.* (2008). Bitcoin Whitepaper. <https://bitcoin.org/bitcoin.pdf>
- [15] C. (2021, 21 abril). *¿Qué es el Nonce?* Crypto4Dummy. <https://crypto4dummy.com/ques-el-nonce/>
- [16] Academy, B. (2022, 16 marzo). *¿Qué es un Algoritmo de Consenso?* Binance Academy. <https://academy.binance.com/es/articles/what-is-a-blockchain-consensus-algorithm>
- [17] *Polkadot Consensus · Polkadot Wiki.* (2022, 14 abril). Consensus. <https://wiki.polkadot.network/docs/learn-consensus>
- [18] Ethereum. (s. f.). *Proof-of-work (PoW).* Ethereum.Org. <https://ethereum.org/en/developers/docs/consensus-mechanisms/pow/#pow-and-mining>

- [19] Ethereum. (s. f.-a). *Proof-of-stake (PoS)*. Ethereum.Org.  
<https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/>
- [20] Ethereum. (s. f.-a). *Gas and fees*. Ethereum.Org.  
<https://ethereum.org/en/developers/docs/gas/>
- [21] Nick Szabo -- *The Idea of Smart Contracts*. (s. f.). The idea of Smart Contracts.  
<https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/idea.html>
- [22] *What are smart contracts on blockchain? | IBM*. (s. f.). Smart Contracts IBM.  
<https://www.ibm.com/topics/smart-contracts>
- [23] Blockchain. (2020, 27 mayo). *Oráculos*. Master Ingeniería Blockchain.  
<https://masterblockchain.net/oraculos-master-blockchain-online/>
- [24] *Carbonsig*. (2021, 18 octubre). Carbonsig. <https://carbonsig.com/>
- [25] *Objetivos de Desarrollo Sostenible*. (s. f.). ODS.  
<https://www.un.org/sustainabledevelopment/es/development-agenda/>
- [26] Academy, B. (2022, 28 abril). *¿Qué es Chainlink?* Bit2Me Academy.  
<https://academy.bit2me.com/que-es-chainlink-link/>
- [27] *Generating Randomness using Chainlink VRF and Brownie*. (s. f.). Engineering Education (EngEd) Program | Section. <https://www.section.io/engineering-education/generating-randomness-with-chainlink-vrf-using-brownie/>

- [28] Español, C. D. C. E. (2022, 1 enero). *Chainlink VRF: Aleatoriedad Verificable On-chain - Chainlink Community*. Medium. <https://medium.com/chainlink-community/chainlink-vrf-aleatoriedad-verificable-on-chain-63964a28954c>
- [29] Merchant, M. (2022, 7 agosto). *¿Qué es Chainlink VRF y cómo funciona?* Cointelegraph. <https://es.cointelegraph.com/news/what-is-chainlink-vrf-and-how-does-it-work>
- [30] *Introduction to Chainlink VRF | Chainlink Documentation*. (s. f.). Chainlink. <https://docs.chain.link/docs/vrf/v2/introduction/>
- [31] Elgarte, F. (2018, 14 marzo). *Números aleatorios en Solidity: mejores y peores prácticas*. Medium. <https://medium.com/@shuffledex/n%C3%BAmeros-aleatorios-en-solidity-mejores-y-peores-pr%C3%A1cticas-85c553d7ba60>
- [32] Reutov, A. (2018, 21 junio). *Predicting Random Numbers in Ethereum Smart Contracts*. Medium. <https://blog.positive.com/predicting-random-numbers-in-ethereum-smart-contracts-e5358c6b8620>
- [33] Rodriguez, A. (2022, 30 marzo). *Solidity: Números aleatorios - Adrian Rodriguez*. Medium. <https://adr-rod87.medium.com/solidity-n%C3%BAmeros-aleatorios-f597bc0d4f8e>
- [34] Costa, P. (2021, 10 diciembre). *Implementing a Blockchain Oracle on Ethereum - Pedro Costa*. Medium. <https://medium.com/@pedrodc/implementing-a-blockchain-oracle-on-ethereum-cedc7e26b49e>
- [35] Hovsepyan, N. (2018, 20 junio). *Lottery Smart Contract: Can we generate random numbers in Solidity?* Medium. <https://medium.com/@promentol/lottery-smart-contract-can-we-generate-random-numbers-in-solidity-4f586a152b27>

# ANEXO I: ALINEACIÓN DEL PROYECTO CON LOS ODS

Los Objetivos de Desarrollo Sostenible (ODS) son 17 objetivos que se han fijado de forma global para intentar lograr alcanzar un futuro mejor y más sostenible para todos. Estas metas fueron marcadas en el año 2015 por la Asamblea General de las Naciones Unidas y la intención es alcanzarlos para el año 2030. Dichos 17 objetivos son los siguientes:

1. Fin de la Pobreza
2. Hambre cero
3. Salud y Bienestar
4. Educación de calidad
5. Igualdad de género
6. Agua limpia y saneamiento
7. Energía sostenible y no contaminante
8. Trabajo y crecimiento económico
9. Industria, innovación e infraestructura
10. Reducción de las desigualdades
11. Ciudades y comunidades sostenibles
12. Producción y consumo responsables
13. Acción por el clima
14. Vida submarina
15. Vida de ecosistemas terrestres
16. Paz, justicia e instituciones solidas
17. Alianzas para alcanzar objetivos



Ilustración 77 - ODS

Este proyecto está alineado con los Objetivos de Desarrollo Sostenible (ODS) y contribuye a la consecución de estos. Este proyecto trata sobre Blockchain y Smart Contract, por lo que a continuación se va a explicar cómo estos pueden ayudar a alcanzar los ODS.

En primer lugar, el Blockchain y los Smart Contract se tratan de tecnologías innovadoras que están revolucinando y cambiando el mundo. Por ello, contribuyen al punto 9. Al ser un sector que requiere desarrollo e investigación, también contribuye al objetivo 8.

Por otra parte, estas tecnologías al tratarse de sistemas descentralizados que no están gobernados por una única autoridad pueden ayudar a resolver conflictos de forma imparcial y más justa. De hecho, el Blockchain puede servir como una herramienta de justicia descentralizada. Por ello, podemos decir que contribuyen a alcanzar los objetivos 10, 16 y 17.

Además, existen muchos proyectos que emplean Smart Contract y Blockchain para frenar las emisiones globales. Por ejemplo, el proyecto de Nick Cogerty llamado Carbonsig.com permite conocer la huella de carbono de un producto desde que nace hasta que desaparece. Para ello, registra todas las emisiones en una blockchain y permite su visualización. Por ello, podemos decir que también contribuye a los objetivos 11, 12 y 15.

Otro campo en el que los Smart Contract y Blockchain pueden ayudar a conseguir los objetivos 1 y 2. Al poder registrar todas las transacciones en una cadena de bloques, se puede saber el dinero que reciben las organizaciones internacionales y consiguiendo que haya una gestión más transparente y de este modo reducir la pobreza y el hambre en el mundo.



## ANEXO II

### *11.1 MANUAL DE INSTALACIÓN*

A continuación, se ofrece un manual para la instalación de los diferentes elementos necesarios para poder ejecutar y probar el código de este proyecto.

#### **11.1.1 METAMASK**

En primer lugar, debemos instalar la extensión de Metamask. En este tutorial se muestra el proceso para instalar Metamask en el navegador Google Chrome.

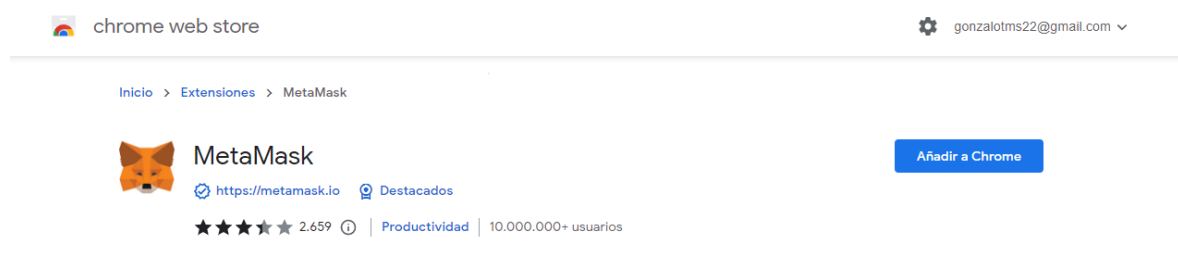
Primero, debemos acceder a su página web: <https://metamask.io/>

Una vez hayamos accedido, buscaremos el botón para descargar para Google Chrome, que será algo similar a la siguiente imagen (Ilustración 78):



*Ilustración 78 - Botón para descargar extensión de Metamask para Google Chrome*

Esto nos llevará a la Chrome Web Store donde podremos añadir a nuestro navegador la extensión (Ilustración 79).



*Ilustración 79 - Añadir extensión de Metamask desde Chrome Web Store*

Una vez añadida la extensión al navegador, nos saldrá la siguiente pantalla (Ilustración 80):



Bienvenido a MetaMask

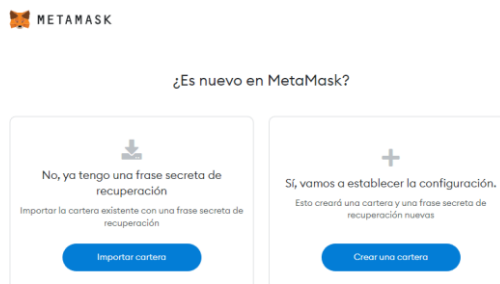
Conectándolo a Ethereum y a la Web descentralizada.

Nos alegra verlo.

Empezar

*Ilustración 80 - Primera pantalla de la extensión de Metamask*

Al presionar el botón empezar, nos preguntará si deseamos crear o importar una cartera (Ilustración 81). Para este tutorial, se muestra el proceso de creación de una nueva cartera.



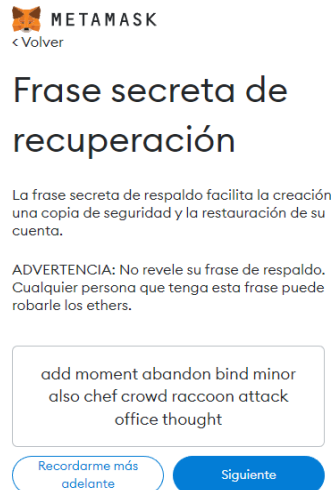
*Ilustración 81 - Metamask: importar o crear cartera*

Ahora, nos pedirá que creamos una nueva contraseña (Ilustración 82)



*Ilustración 82- Metamask: crear contraseña*

Por último, nuestra nueva cartera estará creada y nos mostrará nuestra frase secreta de recuperación (Ilustración 83):



*Ilustración 83 - Metamask: frase secreta de recuperación*

Con esto ya tendremos instalada la extensión de Metamask y habremos creado una nueva cartera.

### 11.1.2 VISUAL STUDIO CODE

Para la instalación de Visual Studio Code (VSC) debemos acceder al siguiente enlace: <https://code.visualstudio.com/download>.

Se nos mostrarán diferentes opciones para descargar el instalador para distintos sistemas operativos (Ilustración 84):

Una vez hayamos descargado el instalador para nuestro sistema operativo, solo debemos seguir los pasos que este nos indica para instalar Visual Studio Code.

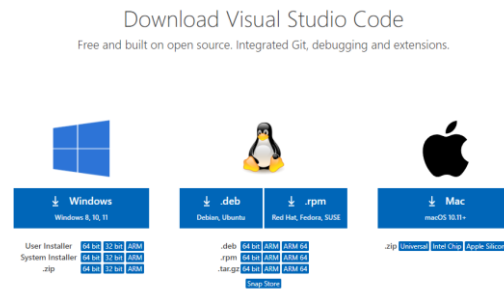


Ilustración 84 - VSC opciones de instalación

Cuando tenemos VSC instalado, si lo ejecutamos podremos acceder a la pantalla de bienvenida de la aplicación:

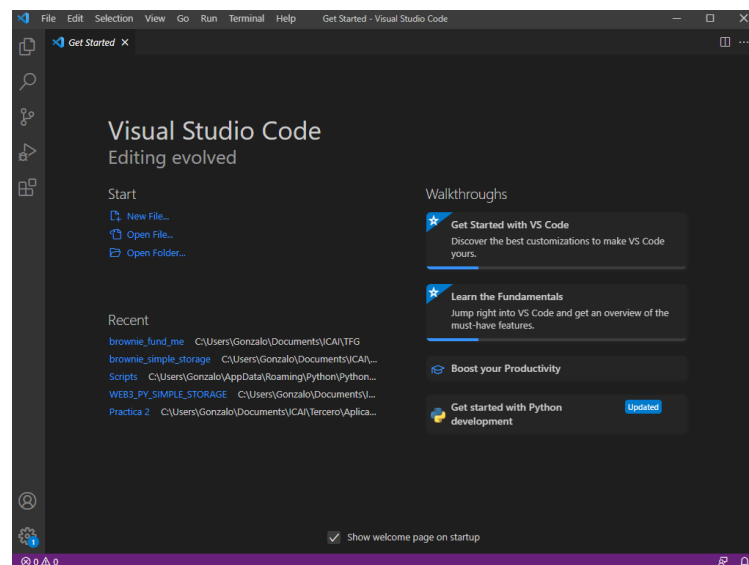


Ilustración 85 - VSC: pantalla de inicio

Aunque no es necesario, recomiendo instalar las siguientes extensiones en VSC para facilitar la visualización y resaltar el código.

- Solidity
- Python
- Bracket Pair Colorized

### 11.1.3 INSTALACIÓN DE PYTHON Y PAQUETES NECESARIOS

Para este proyecto será necesario instalar Python (lo más recomendable es de la versión 3.8 en adelante).

La instalación de Python la podemos realizar descargando el instalador directamente desde su sitio web: <https://www.python.org/downloads/>

En la siguiente imagen se muestra su sitio web (Ilustración 86) y solo debemos presionar en el botón de descargar. Una vez descargado el instalador, lo ejecutamos y seguimos los pasos que nos recomiendan.



*Ilustración 86 - Python: Descarga para la instalación*

Para este proyecto es necesario instalar las siguientes librerías de Python:

- ❖ **web3**: Para interactuar con los Smart Contract y poder realizar transacciones
- ❖ **py-solc-x**: Para poder usar diferentes versiones de solc, que podemos instalar y compilar cuando se necesiten. Solc es el compilador de Solidity.
- ❖ **python-dotenv**: Para poder leer pares de datos del estilo llave-valor de un archivo .env y emplearlos como variables de entorno

La instalación de estos paquetes se realiza mediante pip y para ello podemos usar los siguientes comandos en consola:

```
pip install py-solc-x  
pip install web3  
pip install python-dotenv
```

### 11.1.4 BROWNIE

Para instalar Brownie, lo haremos a través de la consola. Para realizar la instalación de Brownie, se recomienda el uso de PIPX, que es una herramienta que ayuda a la instalación y ejecución de aplicaciones escritas en Python.

Para instalar pipx:

```
python3 -m pip install --user pipx  
python3 -m pipx ensurepath
```

Una vez tenemos instalado pipx, para instalar Brownie usaremos el siguiente comando:

```
pipx install eth-brownie
```

Para comprobar que se ha instalado correctamente escribiremos:

```
brownie
```

Si todo ha funcionado correctamente, deberíamos obtener el siguiente resultado:

```
C:\Users\Gonzalo>brownie  
INFORMACIÓN: no se pudo encontrar ningún archivo para los patrones dados.  
Brownie v1.16.4 - Python development framework for Ethereum
```

Para más información o en caso de querer emplear otro método de instalación, se puede consultar el siguiente enlace:

<https://eth-brownie.readthedocs.io/en/stable/install.html>

## 11.2 CÓDIGO FUENTE

El Código de este proyecto puede encontrarse en el siguiente repositorio de Github:

**GonzaloTMS/smartcontracts\_project**

### 11.2.1 DECENTRALIZEDAPPROACH/NODE\_OFFCHAIN.PY

```
import sys
import json
import requests
from web3 import Web3
import asyncio
from brownie import Lottery, Oracle, network, config, Contract
from scripts.helpful_scripts import (
    get_account,
    FORKED_LOCAL_ENVIROMENTS,
    LOCAL_BLOCKCHAIN_ENVIROMENTS,
)

# Connection information
url = ""
if (
    network.show_active() in LOCAL_BLOCKCHAIN_ENVIROMENTS
    or network.show_active() in FORKED_LOCAL_ENVIROMENTS
):
    print("LOCAL GANACHE")
    url = "http://localhost:8545"
    url_address = "http://localhost:8080/api/addressganache"
else:
    print("TESTNET RINKEBY")
    url = "https://rinkeby.infura.io/v3/85255d60dc214513a8eef637d7c284ce"
    url_address = "http://localhost:8080/api/addressrinkeby"

web3 = Web3(Web3.HTTPProvider(url))
# Contract address and abi
response = requests.get(url_address)
data = response.text
parse_json = json.loads(data)
contract_address = parse_json["address"]
contract_abi = Lottery.abi
```

```
contract = web3.eth.contract(address=contract_address, abi=contract_abi)
print("Contract address: " + contract_address)

# ID of the node: Account who manage the node.
account = ""
secret_key = ""
selected = False
while selected == False:
    value = input("Select node id (1 - 2 - 3)\n")
    if int(value) == 1:
        if (
            network.show_active() in LOCAL_BLOCKCHAIN_ENVIROMENTS
            or network.show_active() in FORKED_LOCAL_ENVIROMENTS
        ):
            # Local-Ganache
            account = "0xB5fB12fd8148441fE7Ad208135dC376923Ff349B"
            secret_key = (
                "0x2F45E72EAEDFB7F2F9AFA52C1B80D7E69C2BAC1CFD35EE746D2AED93917F397B"
            )
        else:
            # Testnet-Rinkeby
            account = "0xbB8147F66FaF71A5bA41E5bD074d6562bd9DB362"
            secret_key = config["wallets"]["from_key"]
            selected = True
    if int(value) == 2:
        if (
            network.show_active() in LOCAL_BLOCKCHAIN_ENVIROMENTS
            or network.show_active() in FORKED_LOCAL_ENVIROMENTS
        ):
            # Local-Ganache
            account = "0x3aD774db8f3d772f214Ae509C41D990b05221EAD"
            secret_key = (
                "0x891fa6b28d036e9da394d9c3e986bc2c84ee2b821d62657dda5a827f272e5658"
            )
        else:
            # Testnet-Rinkeby
            account = "0xC9A5E426bC9af443A1D3Cb0539ef96d17db8bea1"
            secret_key = config["wallets"]["from_key2"]
            selected = True
    if int(value) == 3:
        if (
```



```
network.show_active() in LOCAL_BLOCKCHAIN_ENVIROMENTS
or network.show_active() in FORKED_LOCAL_ENVIROMENTS
):
    # Local-Ganache
    account = "0x8EC83D8Fd96A13beD7984796B3aC8A3B48FbAD0A"
    secret_key = (
"0xe2f5a02199825448892e321543e22757dab765927e14b8bb5c95f169305aeb96"
    )
    else:
        # Testnet-Rinkeby
        account = "0xc9c68d75123Aa15dcFFcF52ad965bCDF0D3Ec216"
        secret_key = config["wallets"]["from_key3"]
        selected = True
print("Account: ", account)

def handle_event(event):
    # First we extract id and url of the event
    event_json = Web3.toJSON(event)
    id = event["args"]["id"]
    url = event["args"]["url"]
    print("ID:", id)
    # Now, we send a get to URL
    response_API = requests.get("http://localhost:8080/api/random")
    # We extract data from response
    data = response_API.text
    parse_json = json.loads(data)
    # We get the random_number
    random_number = parse_json["value"]
    print("Random number", random_number)
    # Store random_number in the smart contract
    raw_transaction = contract.functions.updateRequest(
        id, random_number
    ).buildTransaction(
        {
            "gas": 6721975,
            "from": account,
            "nonce": web3.eth.getTransactionCount(account),
        }
    )

    signed = web3.eth.account.signTransaction(
```

```
        raw_transaction,
        secret_key,
    )
    receipt = web3.eth.sendRawTransaction(signed.rawTransaction)
    web3.eth.waitForTransactionReceipt(receipt)

    print("Random number stored in the blockchain")

# asynchronous defined function to loop
# this loop sets up an event filter and is looking for new entries for the
"NewRequest" event
# this loop runs on a poll interval
async def log_loop(event_filter, poll_interval):
    while True:
        for NewRequest in event_filter.get_new_entries():
            handle_event(NewRequest)
            await asyncio.sleep(poll_interval)

# when main is called
# create a filter for the latest block and look for the "NewRequest" event
for Lottery contract
# run an async loop
# try to run the log_loop function above every 2 seconds
def main():
    event_filter =
contract.events.NewRequest.createFilter(fromBlock="latest")
    loop = asyncio.get_event_loop()
    try:
        loop.run_until_complete(asyncio.gather(log_loop(event_filter, 2)))
    finally:
        # close loop to free up system resources
        loop.close()

if __name__ == "__main__":
    main()
```

## 11.2.2 DEPLOY\_LOTTERY.PY

```
import time
import requests
from scripts.helpful_scripts import (
    get_account,
    get_contract,
    fund_with_link,
    FORKED_LOCAL_ENVIROMENTS,
    LOCAL_BLOCKCHAIN_ENVIROMENTS,
)

from brownie import Lottery, config, network, Oracle

def deploy_lottery():
    account = get_account()
    lottery = Lottery.deploy(
        get_contract("eth_usd_price_feed").address,
        {"from": account},

    publish_source=config["networks"][network.show_active()].get("verify",
False),
    )
    print("Deployed lottery contract")
    return lottery

def deploy_oracle():
    account = get_account()
    oracle = Oracle.deploy(
        {"from": account},

    publish_source=config["networks"][network.show_active()].get("verify",
False),
    )
    print("Deployed oracle contract")
    return oracle

def start_lottery():
    account = get_account()
    lottery = Lottery[-1]
    starting_tx = lottery.startLottery({"from": account})
```

```
starting_tx.wait(1)
print("The lottery is started")

def buy_lotteryTicket(id=None, index=None):
    account = get_account(id=id, index=index)
    lottery = Lottery[-1]
    value = lottery.getTicketPrice() + 100000000
    tx = lottery.buyTicket({"from": account, "value": value})
    tx.wait(1)
    print("You entered de lottery!")

def end_lottery():
    account = get_account()
    lottery = Lottery[-1]
    ending_transaction = lottery.endLottery({"from": account})
    ending_transaction.wait(1)
    # write_random_number(index=2, value=12)
    time.sleep(5)
    print(f"{lottery.last_winner()} is the new winner")
    print(f"{lottery.random_number()}RANDOM")
    print(f"{lottery.random_index()}index contract")
    print(f"{lottery.random_number()%6} Index")

def write_random_number(id=None, index=None, value=3):
    account = get_account(id=id, index=index)
    lottery = Lottery[-1]
    tx = lottery.updateRequest(0, value, {"from": account})
    tx.wait(1)
    print("ACCOUNT:" + str(get_account(index=2)))
    print("MALICIOUS NUMBER: " + str(value))

def main():
    lottery = deploy_lottery()
    url = ""
    if (
        network.show_active() in LOCAL_BLOCKCHAIN_ENVIROMENTS
        or network.show_active() in FORKED_LOCAL_ENVIROMENTS
    ):
        print("LOCAL GANACHE")
        url = "http://localhost:8080/api/addressganache"
```

```
else:
    print("TESTNET RINKEBY")
    url = "http://localhost:8080/api/addressrinkeby"

response = requests.post(url, data=str(Lottery[-1]))

input("Press Enter to continue...")

print("Loteria 1")
print("=====")
start_lottery()
buy_lotteryTicket(index=0)
buy_lotteryTicket(index=1)
buy_lotteryTicket(index=2)
buy_lotteryTicket(index=3)
buy_lotteryTicket(index=4)
buy_lotteryTicket(index=5)
end_lottery()
input("Press Enter to continue...")
print("Loteria 2")
print("=====")
start_lottery()
buy_lotteryTicket(index=0)
buy_lotteryTicket(index=1)
buy_lotteryTicket(index=2)
buy_lotteryTicket(index=3)
buy_lotteryTicket(index=4)
buy_lotteryTicket(index=5)
end_lottery()
input("Press Enter to continue...")
print("Loteria 3")
print("=====")
start_lottery()
buy_lotteryTicket(index=0)
buy_lotteryTicket(index=1)
buy_lotteryTicket(index=2)
buy_lotteryTicket(index=3)
buy_lotteryTicket(index=4)
buy_lotteryTicket(index=5)
end_lottery()
```

### 11.2.3 HELPFUL\_SCRIPTS.PY

```
from brownie import (
    accounts,
    network,
    config,
    MockV3Aggregator,
    Contract,
)

FORKED_LOCAL_ENVIROMENTS = ["mainnet-fork", "mainnet-fork-dev"]
LOCAL_BLOCKCHAIN_ENVIROMENTS = ["development", "ganache-local"]

def get_account(index=None, id=None):
    # accounts[0] --> brownie ganache accounts
    # accounts.add("env") --> from env
    # accounts.load("id")
    if index:
        return accounts[index]
    if id:
        return accounts.load(id)
    if (
        network.show_active() in LOCAL_BLOCKCHAIN_ENVIROMENTS
        or network.show_active() in FORKED_LOCAL_ENVIROMENTS
    ):
        return accounts[0]

    else:
        return accounts.add(config["wallets"]["from_key"])

contract_to_mock = {
    "eth_usd_price_feed": MockV3Aggregator,
}

def get_contract(contract_name):
    """This function will grab the contract addresses from the brwonie
    config if defined,
```

```
    otherwise, it will deploy a mock version of that contract, and return
    that mock contract.

    Args:
        contract_name (string)

    Returns:
        brownie.network.contract.ProjectContract: The most recently deployed
        version of this contract
    """
    contract_type = contract_to_mock[contract_name]
    if network.show_active() in LOCAL_BLOCKCHAIN_ENVIROMENTS:
        if len(contract_type) <= 0:
            deploy_mocks()
            contract = contract_type[-1]
        else:
            contract_address =
config["networks"][network.show_active()][contract_name]
            # address
            # ABI
            contract = Contract.from_abi(
                contract_type._name, contract_address, contract_type.abi
            )
    return contract

DECIMALS = 8
INITIAL_VALUE = 200000000000

def deploy_mocks(decimals=DECIMALS, initial_value=INITIAL_VALUE):
    account = get_account()
    MockV3Aggregator.deploy(decimals, initial_value, {"from": account})
    print("Deployed mocks")

def fund_with_link(
    contract_address, account=None, link_token=None,
    amount=1000000000000000000
): # 0.1 LINK
    account = account if account else get_account()
    link_token = link_token if link_token else get_contract("link_token")
    tx = link_token.transfer(contract_address, amount, {"from": account})
    # link_token_contract = interface.LinkTokenInterface(link_token.address)
```

```
# tx=link_token_contract.transfer(contract_address, amount,  
{"from":account})  
tx.wait(1)  
print("Fund contract!")  
return tx
```



## 11.2.4 LOTTERY.SOL

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.6.6;

import "@chainlink/contracts/src/v0.6/interfaces/AggregatorV3Interface.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "./Oracle.sol";

contract Lottery is Ownable, Oracle {
    //0=Open, 1=Closed, 2=Selecting
    enum STATE {
        OPEN,
        CLOSED,
        PROCESSING
    }
    address payable[] public lottery_players = new address payable[](0);
    uint256 public usdEntryFee;
    AggregatorV3Interface internal ethUsdPriceFeed;
    STATE public state;
    address payable public last_winner;
    uint256 public random_number;
    uint256 public random_index;

    constructor(address _priceFeedAddress) public {
        usdEntryFee = 5 * (10**18); // 5$
        ethUsdPriceFeed = AggregatorV3Interface(_priceFeedAddress);
        state = STATE.CLOSED;
    }

    function buyTicket() public payable {
        //5$
        require(state == STATE.OPEN);
        require(msg.value >= getTicketPrice(), "Not enough ETH!");
        lottery_players.push(msg.sender);
    }

    function getTicketPrice() public view returns (uint256) {
        //Calling priceFeed - Chainlink
        (, int256 price, , , ) = ethUsdPriceFeed.latestRoundData();
        uint256 adjustedPrice = uint256(price) * 10**10; //18 decimals
        uint256 ticketCost = (usdEntryFee * 10**18) / adjustedPrice;
        return ticketCost;
    }
}
```

```
}

function startLottery() public onlyOwner {
    require(
        state == STATE.CLOSED,
        "There is a lottery in progress!You can't open a new lottery"
    );
    state = STATE.OPEN;
}

function endLottery() public onlyOwner {
    state = STATE.PROCESSING;
    uint256 requestId = newRequest("http://localhost:8080/api/random");
}

function fulfillRandomness(uint256 _requestId) internal override {
    require(state == STATE.PROCESSING, "Incorrect State");
    random_number = requests[_requestId].value;
    require(random_number > 0, "Random not found");
    random_index = random_number % lottery_players.length;
    last_winner = lottery_players[random_index];
    last_winner.transfer(address(this).balance);
    //Reset
    lottery_players = new address payable[](0);
    state = STATE.CLOSED;
}
}
```

## 11.2.5 CENTRALIZED APPROACH/ORACLE.SOL

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.6.6;

contract Oracle {
    struct Request {
        uint256 id;
        string url;
        uint256 value;
        mapping(address => uint256) oracles; //1 if hasn't send answer, 2 if
has send answer
    }

    event NewRequest(uint256 id, string url);
    event UpdatedRequest(uint256 id, string url, uint256 value);

    Request[] public requests;
    uint256 currentId;

    function newRequest(string memory _url) internal returns (uint256) {
        requests.push(Request(currentId, _url, 0));
        uint256 length = requests.length;
        Request storage r = requests[length - 1];
        //Trusted oracle address - Address of account that are trusted to
give the final solution.
        //=====
        //Rinkeby
        r.oracles[address(0xbB8147F66FaF71A5bA41E5bD074d6562bd9DB362)] = 1;
//Rinkeby
        //Local
        r.oracles[address(0xB5fB12fd8148441fE7Ad208135dC376923Ff349B)] = 1;
//Local ganache
        //Emit new event. Oracles off-chain will be listening
        emit NewRequest(currentId, _url);
        //Increase request current id;
        currentId++;
        return (currentId - 1);
    }

    function updateRequest(uint256 _id, uint256 _value) public {
        Request storage currentRequest = requests[_id];
        //Check if the oracle is a trusted one.
```

```
if (currentRequest.oracles[address(msg.sender)] == 1) {
    //Oracle has voted
    currentRequest.oracles[address(msg.sender)] = 2;
    currentRequest.value = _value;
    fulfillRandomness(_id);
    emit UpdatedRequest(
        currentRequest.id,
        currentRequest.url,
        currentRequest.value
    );
}
}

function fulfillRandomness(uint256 _requestId) internal virtual {}

function getValue(uint256 _id) public view returns (uint256) {
    return requests[_id].value;
}
}
```

## 11.2.6 DECENTRALIZED APPROACH/ORACLE.SOL

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.6.6;

contract Oracle {
    struct Request {
        uint256 id;
        string url;
        uint256 votes_number;
        uint256 value;
        uint256[] answers;
        mapping(address => uint256) oracles; //1 if hasn't send answer, 2 if
has send answer
    }
    event NewRequest(uint256 id, string url);
    event UpdatedRequest(uint256 id, string url, uint256 value);

    Request[] public requests;
    uint256 currentId;
    uint256 min_votes = 3;

    function newRequest(string memory _url) internal returns (uint256) {
        requests.push(Request(currentId, _url, 0, 0, new uint256[](0)));
        uint256 length = requests.length;
        Request storage r = requests[length - 1];
        //Trusted oracle address - Address of account that are trusted to
give the final solution.
        //=====
        //Rinkeby
        r.oracles[address(0xb8147F66FaF71A5bA41E5bD074d6562bd9DB362)] = 1;
//Rinkeby
        r.oracles[address(0xC9A5E426bC9af443A1D3Cb0539ef96d17db8bea1)] = 1;
//Rinkeby
        r.oracles[address(0xc9c68d75123Aa15dcFFcF52ad965bCDF0D3Ec216)] = 1;
//Rinkeby
        //Ganache
        r.oracles[address(0xB5fB12fd8148441fE7Ad208135dC376923Ff349B)] = 1;
//Ganache
        r.oracles[address(0x3aD774db8f3d772f214Ae509C41D990b05221EAD)] = 1;
//Ganache
        r.oracles[address(0x8EC83D8Fd96A13beD7984796B3aC8A3B48FbAD0A)] = 1;
//Ganache
```

```
//Emit new event. Oracles off-chain will be listening
emit NewRequest(currentId, _url);
//Increase request current id;
currentId++;
return (currentId - 1);
}

function updateRequest(uint256 _id, uint256 _value) public {
    Request storage currentRequest = requests[_id];
    //Check if the oracle is a trusted one.
    if (currentRequest.oracles[address(msg.sender)] == 1) {
        //Oracle has voted
        currentRequest.oracles[address(msg.sender)] = 2;
        //Save answer
        currentRequest.answers.push(_value);
        //Aggregate all answers in the final value.
        currentRequest.value = currentRequest.value + _value /
min_votes;
        //Increase count of oracles who have voted.
        currentRequest.votes_number++;
        //If enough oracles have voted
        if (currentRequest.votes_number >= min_votes) {
            //If enough oracles have voted, we have a valid random
number.
            fulfillRandomness(_id);
        }
        emit UpdatedRequest(
            currentRequest.id,
            currentRequest.url,
            currentRequest.value
        );
    }
}

function fulfillRandomness(uint256 _requestId) internal virtual {}

function getValue(uint256 _id) public view returns (uint256) {
    return requests[_id].value;
}
}
```

## 11.2.7 CHAINLINK/LOTTERY.SOL

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.6.6;

import "@chainlink/contracts/src/v0.6/interfaces/AggregatorV3Interface.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "@chainlink/contracts/src/v0.6/VRFConsumerBase.sol";

contract Lottery is Ownable, VRFConsumerBase {
    //0=Open, 1=Closed, 2=Selecting
    enum STATE {
        OPEN,
        CLOSED,
        PROCESSING
    }
    address payable[] public lottery_players = new address payable[](0);
    uint256 public usdEntryFee;
    AggregatorV3Interface internal ethUsdPriceFeed;
    STATE public state;
    address payable public last_winner;
    uint256 public random_number;
    uint256 public random_index;
    uint256 public fee;
    bytes32 public keyhash;

    constructor(
        address _priceFeedAddress,
        address _vrfCoordinator,
        address _link,
        uint256 _fee,
        bytes32 _keyhash
    ) public VRFConsumerBase(_vrfCoordinator, _link) {
        usdEntryFee = 5 * (10**18); // 5$
        ethUsdPriceFeed = AggregatorV3Interface(_priceFeedAddress);
        state = STATE.CLOSED;
        fee = _fee;
        keyhash = _keyhash;
    }

    function buyTicket() public payable {
        //5$
        require(state == STATE.OPEN);
    }
}
```

```
require(msg.value >= getTicketPrice(), "Not enough ETH!");
lottery_players.push(msg.sender);
}

function getTicketPrice() public view returns (uint256) {
    //Calling priceFeed - Chainlink
    (, int256 price, , , ) = ethUsdPriceFeed.latestRoundData();
    uint256 adjustedPrice = uint256(price) * 10**10; //18 decimals
    uint256 ticketCost = (usdEntryFee * 10**18) / adjustedPrice;
    return ticketCost;
}

function startLottery() public onlyOwner {
    require(
        state == STATE.CLOSED,
        "There is a lottery in progress!You can't open a new lottery"
    );
    state = STATE.OPEN;
}

function endLottery() public onlyOwner {
    state = STATE.PROCESSING;
    bytes32 requestId = requestRandomness(keyhash, fee);
}

function fulfillRandomness(bytes32 _requestId, uint256 _randomness)
    internal
    override
{
    require(state == STATE.PROCESSING, "Incorrect State");
    require(_randomness > 0, "Random not found");
    random_number = _randomness;
    random_index = random_number % lottery_players.length;
    last_winner = lottery_players[random_index];
    last_winner.transfer(address(this).balance);
    //Reset
    lottery_players = new address payable[](0);
    state = STATE.CLOSED;
}
}
```



## 11.2.8 BROWNIE-CONFIG.YAML

```
dotenv: .env
dependencies:
  # - <organization/repo>@<version>
  - smartcontractkit/chainlink-brownie-contracts@1.1.1
  - OpenZeppelin/openzeppelin-contracts@3.4.0
compiler:
  solc:
    remappings:
      - '@chainlink=smartcontractkit/chainlink-brownie-contracts@1.1.1'
      - '@openzeppelin=OpenZeppelin/openzeppelin-contracts@3.4.0'
networks:
  default: development
  development:
    keyhash:
      '0x2ed0feb3e7fd2022120aa84fab1945545a9f2ffc9076fd6156fa96eaff4c1311'
      fee: 1000000000000000000
    rinkeby:
      eth_usd_price_feed: '0x8A753747A1Fa494EC906cE90E9f37563A8AF630e'
      vrf_coordinator: '0xb3dCcb4Cf7a26f6cf6B120Cf5A73875B7BBc655B'
      link_token: '0x01BE23585060835E02B77ef475b0Cc51aA1e0709'
      keyhash:
        '0x2ed0feb3e7fd2022120aa84fab1945545a9f2ffc9076fd6156fa96eaff4c1311'
        fee: 1000000000000000000
        verify: False
    mainnet-fork:
      eth_usd_price_feed: '0x5f4eC3Df9cbd43714FE2740f5E3616155c5b8419'
wallets:
  from_key: ${PRIVATE_KEY}
  from_key2: ${PRIVATE_KEY2}
  from_key3: ${PRIVATE_KEY3}
```

### 11.2.9 CENTRALIZEDAPPROACH/NODE\_OFFCHAIN.PY

```
import sys
import json
import requests
from web3 import Web3
import asyncio
from brownie import Lottery, Oracle, network, config, Contract
from scripts.helpful_scripts import (
    get_account,
    FORKED_LOCAL_ENVIROMENTS,
    LOCAL_BLOCKCHAIN_ENVIROMENTS,
)

# Connection information
url = ""
if (
    network.show_active() in LOCAL_BLOCKCHAIN_ENVIROMENTS
    or network.show_active() in FORKED_LOCAL_ENVIROMENTS
):
    print("LOCAL GANACHE")
    url = "http://localhost:8545"
    url_address = "http://localhost:8080/api/addressganache"
else:
    print("TESTNET RINKEBY")
    url = "https://rinkeby.infura.io/v3/85255d60dc214513a8eef637d7c284ce"
    url_address = "http://localhost:8080/api/addressrinkeby"

web3 = Web3(Web3.HTTPProvider(url))
# Contract address and abi
response = requests.get(url_address)
data = response.text
parse_json = json.loads(data)
contract_address = parse_json["address"]
contract_abi = Lottery.abi
contract = web3.eth.contract(address=contract_address, abi=contract_abi)
print("Contract address: " + contract_address)

# ID of the node: Account who manage the node.
account = ""
secret_key = ""
if (
```

```
network.show_active() in LOCAL_BLOCKCHAIN_ENVIROMENTS
or network.show_active() in FORKED_LOCAL_ENVIROMENTS
):
# Local-Ganache
account = "0xB5fB12fd8148441fE7Ad208135dC376923Ff349B"
secret_key =
"0x2F45E72EAEDFB7F2F9AFA52C1B80D7E69C2BAC1CFD35EE746D2AED93917F397B"
else:
# Testnet-Rinkeby
account = "0xbB8147F66FaF71A5bA41E5bD074d6562bd9DB362"
secret_key = config["wallets"]["from_key"]
print("Account: ", account)

def handle_event(event):
# First we extract id and url of the event
event_json = Web3.toJSON(event)
id = event["args"]["id"]
url = event["args"]["url"]
print("ID:", id)
# Now, we send a get to URL
response_API = requests.get("http://localhost:8080/api/random")
# We extract data from response
data = response_API.text
parse_json = json.loads(data)
# We get the random_number
random_number = parse_json["value"]
print("Random number", random_number)
# Store random_number in the smart contract
raw_transaction = contract.functions.updateRequest(
    id, random_number
).buildTransaction(
    {
        "gas": 6721975,
        "from": account,
        "nonce": web3.eth.getTransactionCount(account),
    }
)

signed = web3.eth.account.signTransaction(
    raw_transaction,
    secret_key,
)
```

```
receipt = web3.eth.sendRawTransaction(signed.rawTransaction)
web3.eth.waitForTransactionReceipt(receipt)

print("Random number stored in the blockchain")

# asynchronous defined function to loop
# this loop sets up an event filter and is looking for new entries for the
"NewRequest" event
# this loop runs on a poll interval
async def log_loop(event_filter, poll_interval):
    while True:
        for NewRequest in event_filter.get_new_entries():
            handle_event(NewRequest)
            await asyncio.sleep(poll_interval)

# when main is called
# create a filter for the latest block and look for the "NewRequest" event
for Lottery contract
# run an async loop
# try to run the log_loop function above every 2 seconds
def main():
    event_filter =
contract.events.NewRequest.createFilter(fromBlock="latest")
    loop = asyncio.get_event_loop()
    try:
        loop.run_until_complete(asyncio.gather(log_loop(event_filter, 2)))
    finally:
        # close loop to free up system resources
        loop.close()

if __name__ == "__main__":
    main()
```