MÁSTER UNIVERSITARIO EN INGENIERÍA DE TELECOMUNICACIONES

TRABAJO FIN DE MÁSTER

DESARROLLO E IMPLEMENTACION DE UN SISTEMA DESCENTRALIZADO DE GESTIÓN DE APUESTAS ON-CHAIN

Autor: Diego Escondrillas

Director: Alonso Rodríguez Rodríguez

Madrid

Agosto de 2023

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

Desarrollo e implementación de un sistema descentralizado de gestión de apuestas on-chain

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el

curso académico 2022/2023 es de mi autoría, original e inédito y

no ha sido presentado con anterioridad a otros efectos.

El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido

tomada de otros documentos está debidamente referenciada.

Fdo.: Diego Escondrillas Romera Fecha: ……/ ……/ ……

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO

Fdo.: Alonso Rodríguez Rodríguez Fecha: ……/ ……/ ……

MÁSTER UNIVERSITARIO EN INGENIERÍA DE TELECOMUNICACIONES

TRABAJO FIN DE MÁSTER

DESARROLLO E IMPLEMENTACION DE UN SISTEMA DESCENTRALIZADO DE GESTIÓN DE APUESTAS ON-CHAIN

Autor: Diego Escondrillas

Director: Alonso Rodríguez Rodríguez

Madrid

Agosto de 2023

Agradecimientos

A mi familia y amigos por aguantarme a lo largo de este camino.

A mi tutor D. Alonso Rodríguez Rodríguez por ayudarme y guiarme en este mundo en el que hasta hace poco no tenía demasiados conocimientos con paciencia y dedicación.

DESARROLLO E IMPLEMENTACION DE UN SISTEMA DESCENTRALIZADO DE GESTIÓN DE APUESTAS ON-CHAIN

Autor: Escondrillas Romera, Diego.

Director: Rodríguez Rodríguez, Alonso.

Entidad Colaboradora: ICAI – Universidad Pontificia Comillas.

RESUMEN DEL PROYECTO

I. INTRODUCCIÓN Y OBJETIVOS

La industria de las apuestas ha evolucionado considerablemente a lo largo de la historia, adaptándose a los avances tecnológicos y a las demandas cambiantes de los jugadores. Desde los primeros juegos de azar hasta los complejos sistemas de apuestas en línea, el sector ha experimentado una constante búsqueda de mayor seguridad, transparencia y eficiencia.

En este contexto, la tecnología Blockchain ha surgido como una innovación disruptiva que tiene el potencial de revolucionar la forma en que se gestionan las apuestas. La introducción de la web 2.0, caracterizada por la interactividad y la participación del usuario, sentó las bases para la exploración de nuevas soluciones tecnológicas en el ámbito de las apuestas.

La web 2.0 permitió la creación de plataformas en línea donde los jugadores podían realizar apuestas de forma más accesible y conveniente. Sin embargo, estas plataformas todavía dependían de intermediarios centralizados para gestionar las transacciones y garantizar la integridad de las apuestas.

La llegada de la tecnología Blockchain ha permitido cambiar completamente la forma en la que hasta entonces era la más habitual a la hora de realizar apuestas. Con esta tecnología, se ha adoptado un enfoque completamente descentralizado y transparente en la gestión de apuestas. Para ello, se requiere el uso de smart contracts. Estos contratos permiten eliminar intermediarios y proporcionar una mayor confianza en las transacciones.

En el presente trabajo, se requiere también el uso de oráculos. Los oráculos desempeñan un papel crucial al permitir que los contratos inteligentes interactúen con fuentes de datos externas, como resultados deportivos. La utilización de oráculos junto con smart contracts busca asegurar la confiabilidad de los resultados y la ejecución automática de las apuestas de acuerdo con los términos establecidos en los contratos. Todo lo anterior garantiza que las apuestas de los jugadores se resolverán de manera justa y transparente y reduciendo los riesgos de manipulación o fraude.

En este contexto, el presente trabajo se enmarca en el contexto histórico de la evolución de las apuestas y la llegada de la tecnología Blockchain. El objetivo es desarrollar e implementar un sistema descentralizado de gestión de apuestas on-chain que capitalice los avances tecnológicos recientes y brinde una solución más segura, transparente y confiable para los jugadores.

Los objetivos específicos que se buscan conseguir en este trabajo son:

- Explorar el impacto de la tecnología Blockchain y los contratos inteligentes junto con el uso de oráculos en la industria de las apuestas, analizando casos de uso relevantes y las ventajas que ofrecen tanto en términos de seguridad como de transparencia.
- Diseñar e implementar un sistema descentralizado de gestión de apuestas basado en Blockchain y los contratos inteligentes para garantizar la integridad y la confianza en las transacciones.

Al lograr estos objetivos, se busca sentar las bases para un nuevo paradigma en la gestión de apuestas, donde la tecnología Blockchain y los contratos inteligentes se conviertan en elementos clave para brindar mayor confianza, transparencia y seguridad a los jugadores.

II. METODOLOGÍA

En el desarrollo del proyecto se procederá de la siguiente manera:

- **A)** En primer lugar, se introducirá el trabajo mediante una sección dedicada a la presentación de tecnología Blockchain. Se comentarán las distintas redes que hacen uso de dicha tecnología y se elegirá la red apropiada para la realización de este trabajo. Asimismo, se describirán los elementos que hacen de dicha red la idónea para llevar a cabo este proyecto. Finalmente, se comentarán todos los elementos que emplearán un rol importante para cumplir los objetivos previamente marcados.
- **B)** El capítulo siguiente se dedicará al desarrollo del contrato inteligente que será el encargado de gestionar los fondos que depositen los usuarios para la realización de apuestas. Con su correspondiente llamada a un oráculo para obtener los resultados de los eventos a los que los usuarios han apostado. Asimismo, se encargará de calcular las cuotas asociadas a las distintas partes involucradas en cada partido y de repartir las ganancias correspondientes en caso de acierto.
- **C)** El capítulo cuatro implicará el levantamiento de un nodo Chainlink y la creación de un servidor intermedio cuya tarea principal será almacenar la API Key para que no tenga que ser incluida en cada llamada que el contrato haga al nodo Chainlink.
- **D)** El capítulo cinco comentará ciertas complicaciones que se han encontrado a lo largo del desarrollo del proyecto que han impedido lograr los objetivos previamente establecidos.
- **E)** Finalmente, se establecerán las conclusiones y las posibles líneas de mejora en el futuro.

III. RESULTADOS

Durante la realización del proyecto se han encontrado una serie de obstáculos que han impedido lograr los objetivos del proyecto.

Las complicaciones que se han encontrado se mencionan a continuación:

- Imposibilidad de añadir ETH al nodo Chainlink
- Incapacidad de lograr la comunicación entre el contrato y el nodo Chainlink para la obtención de los resultados deportivos.

IIII. CONCLUSIONES

A pesar de que el enfoque de la realización de apuestas utilizando la metodología Blockchain no se ha podido completar de la manera que se esperaba, se pueden sacar ciertas conclusiones:

1. **Ventajas de la tecnología Blockchain en apuestas online**.
   Como se ha comentado anteriormente, una de las características de la propia tecnología Blockchain es la transparencia, la cual permite evitar el fraude y la manipulación de resultados.
2. **Eficiencia y automatización**.
   Dado que los contratos inteligentes se pueden programar de manera que se ejecuten cuando se cumpla cierta condición, se automatizan las operaciones y se agiliza todo el proceso de reparto de ganancias a los ganadores de las apuestas.
3. **Interacción con el contrato inteligente**.
   A diferencia de las casas de apuestas online no basadas en Blockchain, el sistema realizado en este trabajo no cuenta con una interfaz de usuario para interactuar con el contrato. De esta manera, los usuarios que quieran apostar deben llamar a una función de un contrato inteligente, lo cual puede hacer que la experiencia del usuario no sea la mejor debido a que resulta más cómodo tener una interfaz donde el usuario elija ahí el tipo de apuesta que desea hacer y no andar introduciendo los parámetros uno a uno en la llamada a la función (para lo cual además se necesita saber el contenido del contrato).
4. **Falta de contenido para apostar**.
   Pese a que en el sistema realizado en este trabajo se pueden realizar apuestas a dos deportes, convendría que se pudieran realizar apuestas a la mayoría de deportes posibles. Alguno de los deportes que sería interesante incluir pueden ser baloncesto o tenis.
5. **Desafíos regulatorios**.
   Las regulaciones en muchas jurisdicciones requieren la verificación de la identidad de los usuarios en los sistemas de apuestas en línea. Esto puede ser un desafío en un entorno descentralizado, ya que se debe encontrar un equilibrio entre la privacidad de los usuarios y el cumplimiento de los requisitos de identificación.
6. **Impacto en la industria**.
   Este trabajo, puede suponer una evolución en la industria de las apuestas en línea. Dadas las numerosas ventajas que ofrece la tecnología Blockchain y la continua adopción de activos digitales, es posible que las casas de apuestas se replanteen la posibilidad de utilizar dicha tecnología para los usuarios realicen sus apuestas.
7. **Falta de información sobre el tema**.
   Una de las principales razones por las cuales no se han podido lograr los objetivos del proyecto ha sido la falta de información sobre el tema que se ha encontrado. Concretamente todo lo relacionado con Chainlink. Es cierto que el sitio web de Chainlink tiene ejemplos sobre cómo realizar interacción con contratos, pero en muchos casos esos ejemplos son erróneos. Asimismo, la página web cuenta con varios ejemplos en los que muestra un contrato y la especificación de un Job que deberá ser llamado por el contrato. Sin embargo, en muchas ocasiones el contrato no correspondía con la especificación del Job, lo que no ha hecho más que crear dudas a la hora de realizar el trabajo, lo que ha provocado que no se haya podido completar el trabajo.

III. FUTURAS LÍNEAS DE INVESTIGACIÓN

Se pueden destacar algunas futuras y posibles líneas de investigación:

1. Averiguar donde se encuentra el error que ha impedido lograr los objetivos.
2. Reducir el gasto computacional y, por ende, de gas requerido para la ejecución del código que alberga el contrato inteligente.
3. Que el cálculo de cuotas no se realice en el propio contrato, sino que también se utilice otro oráculo para obtener unas cuotas asociadas a un evento.
4. Implementación de un *front-end* con el que el usuario pueda interactuar directamente con el contrato.

DEVELOPMENT AND IMPLEMENTATION OF A DECENTRALIZED ON-CHAIN BETTING MANAGEMENT SYSTEM

ABSTRACT

I. INTRODUCTION AND OBJECTIVES

The betting industry has evolved considerably throughout history, adapting to technological advances and changing player demands. From the first games of chance to complex online betting systems, the sector has experienced a constant search for greater security, transparency and efficiency.

In this context, blockchain technology has emerged as a disruptive innovation that has the potential to revolutionize the way gambling is managed. The introduction of web 2.0, characterized by interactivity and user participation, laid the groundwork for the exploration of new technological solutions in the field of betting.

Web 2.0 enabled the creation of online platforms where players could place bets in a more accessible and convenient way. However, these platforms still relied on centralized intermediaries to manage transactions and ensure betting integrity.

The advent of blockchain technology has allowed for a complete change in the way betting was previously done. With this technology, a completely decentralized and transparent approach to betting management has been adopted. This requires the use of smart contracts. These contracts make it possible to eliminate intermediaries and provide greater trust in transactions.

In the present work, the use of oracles is also required. Oracles play a crucial role in allowing smart contracts to interact with external data sources, such as sports results. The use of oracles together with smart contracts seeks to ensure the reliability of the results and the automatic execution of bets according to the terms established in the contracts. All the above guarantees that players' bets will be settled in a fair and transparent manner and reducing the risks of manipulation or fraud.

In this context, the present work is framed in the historical context of the evolution of betting and the advent of blockchain technology. The objective is to develop and implement a decentralized on-chain betting management system that capitalizes on recent technological advances and provides a more secure, transparent, and reliable solution for players.

The specific objectives sought to be achieved in this work are:

- Explore the impact of blockchain technology and smart contracts together with the use of oracles in the betting industry, analyzing relevant use cases and the advantages they offer both in terms of security and transparency.
- Design and implement a decentralized betting management system based on blockchain and smart contracts to ensure integrity and trust in transactions.

By achieving these objectives, we seek to lay the foundations for a new paradigm in betting management, where blockchain technology and smart contracts become key elements to provide greater trust, transparency, and security to players.

II. METHODOLOGY

The development of the project will proceed as follows:

- **A)** First, the work will be introduced through a section dedicated to the presentation of blockchain technology. The different networks that make use of this technology will be discussed and the appropriate network for this work will be chosen. Likewise, the elements that make this network the ideal one to carry out this project will be described. Finally, all the elements that will play an important role in achieving the objectives previously set will be discussed.
- **B)** The following chapter will be dedicated to the development of the smart contract that will be in charge of managing the funds deposited by the users to place bets. With its corresponding call to an oracle to obtain the results of the events to which the users have bet. It will also oversee calculating the odds associated with the different parties involved in each match and of distributing the corresponding winnings in case of success.
- **C)** Chapter four will involve the raising of a Chainlink node and the creation of an intermediate server that its main task will be storing the API Key so that it doesn´t have to be included in every call that the contract makes to the Chainlink node.
- **D)** Chapter five will discuss certain complications that have been encountered throughout the development of the project that have prevented the achievement of the previously established objectives.
- **E)** Finally, conclusions and possible lines of improvement for the future will be drawn.

III. RESULTS

During the implementation of the project, several obstacles have been encountered that have prevented the achievement of the project objectives.

The complications encountered are listed below:

- Inability to add ETH to the Chainlink node.
- Inability to achieve communication between the contract and the Chainlink node for sports results.

IV. CONCLUSIONS

The approach to placing bets using the Blockchain methodology has proved to be successful. That is why certain conclusions can be drawn after the completion of the work:

1. **Advantages of Blockchain technology in online betting.**
   As mentioned above, one of the characteristics of Blockchain technology itself is transparency, which prevents fraud and manipulation of results.
2. **Efficiency and automation**.
   Since smart contracts can be programmed in such a way that they are executed when a certain condition is met, operations are automated and the whole process of distributing winnings to bet winners is streamlined.
3. **Interaction with the smart contract**.
   Unlike non-Blockchain-based online bookmakers, the system realized in this work does not have a user interface to interact with the contract. Thus, users who want to bet must call a function of a smart contract, which can make the user experience not the best because it is more convenient to have an interface where the user chooses there the type of bet he/she wants to make and not to go around entering the parameters one by one in the function call (for which it is also necessary to know the content of the contract).
4. **Lack of betting content**.
   Although in the system used in this work it is possible to place bets on two sports, it would be desirable to be able to place bets on most of sports. Some of the sports that would be interesting to include could be basketball or tennis.
5. **Regulatory challenges**.
   Regulations in many jurisdictions require verification of the identity of users in online betting systems. This can be a challenge in a decentralized environment, as a balance must be struck between user privacy and compliance with identification requirements.
6. **Impact on the industry**.
   This work may lead to an evolution in the online gambling industry. Given the many advantages offered by Blockchain technology and the continued adoption of digital assets, it is possible that bookmakers will rethink the possibility of using such technology for users to place their bets.
7. **Lack of information on the subject**.
   One of the main reasons why it has not been possible to achieve the objectives of the project has been the lack of information on the subject that has been found. Specifically, everything related to Chainlink. It is true that the Chainlink website has examples on how to perform interaction with contracts, but in many cases those examples are wrong. Also, the website has several examples where it shows a contract and the specification of a Job to be called by the contract. However, in many cases the contract did not correspond to the Job specification, which has only created doubts when performing the job, resulting in the job not being able to be completed.

V. FUTURE RESEARCH LINES

Some future and possible lines of research can be highlighted:

1. Find out where the error is that has prevented the achievement of the objectives.
2. Reducing the computational and thus gas overhead required for the execution of the code that hosts the smart contract.
3. That the calculation of quotas is not performed in the contract itself, but that another oracle is also used to obtain quotas associated with an event.
4. Implementation of a front-end with which the user can interact directly with the contract.

Organización del Proyecto

El proyecto se encuentra dividido en varios capítulos. El contenido de cada uno de ellos se muestra a continuación:

I. Memoria

- **Capítulo 1**. Se centra en realizar una introducción y estado del arte al sujeto estudiado y al presente trabajo. En ella se marcan los objetivos propuestos y la metodología a seguir para alcanzarlos. Asimismo, se detallan los objetivos de desarrollo sostenibles relacionados con el trabajo, el plan seguido para la realización satisfactoria del proyecto, y los recursos empleados.
- **Capítulo 2**. Consiste en una introducción a la tecnología Blockchain y a todos los elementos involucrados en dicha tecnología, los cuales son necesarios para la realización del proyecto.
- **Capítulo 3**. Se trata de una breve explicación del lenguaje de programación utilizado para la creación del contrato inteligente, así como el desarrollo del contrato en sí. También incluye tanto la explicación del código como unas pruebas que se han realizado con motivo de comprobar el correcto funcionamiento del contrato.
- **Capítulo 4**. Se detalla el levantamiento del nodo Chainlink y de los servidores intermedios que son los encargados de interactuar con la API que contiene los resultados de los eventos deportivos. Además, se comenta la utilización de TMUX para que los servidores estén continuamente levantados.
- **Capítulo 5**. Se comentan algunos problemas que se han encontrado y que han impedido lograr los objetivos del proyecto, así como posibles razones por las cuales han surgido esas complicaciones
- **Capítulo 6**. Se centra en comentar más en detalle las conclusiones obtenidas en el desarrollo de este proyecto. Además, se tratan nuevamente las futuras líneas de investigación que se desarrollarán en un futuro.

II. Anexo

- **Código**. Se detalla todo el código escrito durante la realización del trabajo con el objetivo de que el lector que esté interesado pueda aprender sobre el tema

Desarrollo e implementación de un sistema descentralizado de gestión de apuestas on-chain

Índice

Tabla de Ilustraciones

Capítulo 1

1. Introducción

1.1 Introducción

El juego en línea y las apuestas deportivas han experimentado un crecimiento significativo en los últimos años gracias a la facilidad y comodidad que brinda Internet. Sin embargo, a pesar de este crecimiento, la industria aún enfrenta una serie de desafíos incluyendo problemas de seguridad y transparencia, así como la falta de confianza de los usuarios en los resultados de los juegos.

Es aquí donde entran en juego los contratos inteligentes, que permiten la creación de sistemas de apuestas en línea seguros y transparentes. Un contrato inteligente es un programa autónomo que se ejecuta en una red blockchain y que permite llevar a cabo transacciones y acuerdos de manera segura y sin intermediación.

En este trabajo, se presenta un sistema de apuestas en línea basado en contratos inteligentes que utiliza la tecnología Blockchain para ofrecer una experiencia de juego segura y transparente a los usuarios. Este sistema permite a los usuarios realizar apuestas en deportes y eventos, y utiliza contratos inteligentes para garantizar la transparencia y la veracidad de los resultados. Con este sistema, los usuarios pueden confiar en que los resultados son justos y precisos, lo que aumenta la confianza en el sistema y atrae a una base de usuarios más amplia.

Una de las características principales de los *smart contracts* es que se ejecutan de manera aislada, lo que significa que la máquina virtual de Ethereum (EVM) ejecuta los contratos en un *sandbox* en donde solo tienen acceso a los datos locales o datos provistos por otros contratos en la misma red. Aquí es donde entran en juego los oráculos, los cuales permiten conectar información del mundo real con los contratos.

De esta manera, para garantizar que los resultados son precisos se empleará el oráculo *Chainlink*. Éste es un oráculo, el cual tiene como objetivo conectar información del mundo real con el contrato inteligente. El objetivo será que el contrato reciba de forma automática la información de los distintos eventos deportivos a los que los jugadores han apostado y haga el reparto de ganancias correspondiente en caso de que uno de ellos sea ganador tras realizar un cálculo de la cuota a la que se pagaba que ganaría el ganador.

Para que el sistema esté funcionalmente operativo, se debe tener el *smart contract* desplegado en la cadena de bloques. Para ello, el contrato debe estar escrito en un lenguaje de bajo nivel para que pueda ser ejecutado por los nodos que componen la red. El procedimiento de pasar el contrato de un lenguaje de alto nivel a otro de bajo nivel que puede ser interpretado por los nodos que componen la red se denomina compilación.

1.2 Estado de la cuestión

El uso de contratos inteligentes para la creación de sistemas de apuestas en línea es un área de creciente interés en el mundo de la tecnología. Diversas soluciones tecnológicas han sido propuestas para abordar los desafíos que enfrenta la industria de las apuestas en línea, incluyendo la falta de transparencia y seguridad en los resultados de los juegos.

Entre las soluciones tecnológicas más destacadas, se encuentran las plataformas basadas en Blockchain que utilizan contratos inteligentes para garantizar la transparencia e integridad de los resultados de los juegos. Además de poder automatizar los procesos de pago y distribución de ganancias a los ganadores, lo que reduce la posibilidad de errores humanos y acelera el proceso. Estas plataformas permiten a los usuarios realizar apuestas en deportes y eventos en tiempo real, y utilizan contratos inteligentes para asegurar que los resultados sean justos y precisos gracias a la utilización de oráculos. Asimismo, estas plataformas suelen ser altamente seguras desde el punto de vista de la privacidad, ya que utilizan tecnologías de cifrado avanzadas para proteger los datos de los usuarios y las transacciones. Cabe destacar que, en las cadenas de bloques públicas como Ethereum se utilizan direcciones de carteras para identificar a los usuarios, las cuales no están necesariamente vinculadas a la identidad real de un individuo, lo que brinda a los usuarios con cierto anonimato.

Otras soluciones tecnológicas son las plataformas de apuestas que utilizan tecnologías de inteligencia artificial y aprendizaje automático para optimizar la experiencia de juego y mejorar la precisión de los resultados. Estas soluciones también utilizan técnicas de seguridad avanzadas para proteger la privacidad de los usuarios y garantizar la integridad de los resultados.

En conclusión, existen varias soluciones tecnológicas que abordan los desafíos que enfrenta la industria de las apuestas en línea. Aunque todas estas soluciones ofrecen diferentes enfoques y tecnologías, todas comparten el objetivo común de brindar una experiencia de juego privada, inmutable y transparente a los usuarios. Con el continuo desarrollo de tecnologías avanzadas, es probable que la industria continúe evolucionando y mejorando para ofrecer soluciones cada vez más innovadoras y efectivas a los desafíos que enfrenta.

1.3 Motivación

La realización de este proyecto se justifica por la necesidad de mejorar la eficiencia y la confianza en este tipo de plataformas. Como se ha mencionado anteriormente, los *smart contracts* proporcionan una solución transparente, automatizada, sin intermediarios y "anónima" para el proceso de apuestas en línea. Estas características permiten reducir los costes, acelerar el proceso, garantizar la protección de los fondos de los usuarios y mejorar la confianza en el sistema.

En el caso de los fondos de los usuarios, éstos se encuentran protegidos debido a que no están bajo el control de una única entidad centralizada que pueda ser vulnerable a ataques, sino que están protegidos por la red completa, lo cual es precisamente la principal característica de la tecnología Blockchain. Asimismo, la transparencia en Blockchain se logra a través de la inmutabilidad de los registros, lo que significa que una vez que se han registrado las transacciones, estas no pueden ser modificadas o eliminadas.

El mercado de apuestas en línea sigue creciendo a nivel mundial, lo que significa que hay una demanda creciente de soluciones eficientes y confiables para este tipo de plataformas. La implementación de *smart contracts* en un sistema de apuestas en línea es un paso importante en la dirección de mejorar la experiencia del usuario y la eficiencia de la plataforma en general. Al mismo tiempo, esto puede atraer a un público más amplio que busca una alternativa más segura (por las razones comentadas anteriormente) y justa a los sistemas de apuestas tradicionales.

De esta manera, se concluye que, la realización de este proyecto se justifica por la necesidad de mejorar la eficiencia, la confianza y la transparencia en el mercado de apuestas en línea, lo que a su vez contribuirá a la satisfacción del usuario y al éxito del proyecto en sí. La eficiencia y la transparencia se deben en gran parte a la descentralización que ofrece la propia tecnología Blockchain. Esto se debe a que, con dicha tecnología, se reducen costes de infraestructura debido a que se elimina la necesidad de intermediarios. Finalmente, eliminar la dependencia de una autoridad centralizada puede reducir la posibilidad de censura basada en factores como la raza, religión, género, etc. ya que la información y el control sobre los fondos se distribuyen entre muchos nodos.

1.4 Objetivos

El objetivo principal de la utilización de *smart contracts* en un sistema de apuestas en línea es mejorar la protección de los fondos de los usuarios, proporcionar una mayor integridad de los resultados, otorgar transparencia a los mismos y la automatización del proceso de apuestas.

Mediante la implementación de *smart contracts*, se garantiza que las reglas y condiciones del juego sean claras y no puedan ser modificadas posteriormente, lo que reduce el riesgo de fraude y manipulación de resultados, garantizándose la integridad de estos. Además, los *smart contracts* permiten la automatización de procesos, como el cálculo de ganancias y pagos, lo que reduce la posibilidad de errores humanos y acelera los tiempos de procesamiento.

Otro aspecto importante es la transparencia, ya que todas las transacciones y resultados son registrados en la cadena de bloques, lo que permite una auditoría fácil y confiable.

Por todo lo que se ha comentado, la seguridad se mejora al utilizar smart contracts, ya que son almacenados en una blockchain, la cual es resistente a la manipulación. Esto significa que los fondos de los usuarios estarán protegidos y seguros en todo momento, lo que aumenta la confianza de los usuarios en el sistema.

De esta manera, los objetivos específicos que se tienen para poder llevar a cabo el proyecto son:

1. Desarrollo de un *smart contract* que reciba los pagos asociados a las apuestas que realizan los usuarios en criptomonedas y los almacenará en su dirección asociada.
   El contrato deberá también ser capaz de calcular las cuotas correspondientes a cada equipo involucrado en el partido apostado en función del número de apostantes a dicho evento.

2. Implementar la interacción de dicho *smart contract* otro oráculo para que los resultados de los distintos eventos a los que los jugadores han apostado le lleguen al contrato y éste realice los pagos correspondientes.
3. Compilación y despliegue del contrato en una red blockchain.

1.5 Metodología

Para una realización exitosa del proyecto es necesario llevar a cabo una serie de procedimientos para asegurar que nuestro sistema cumple con las características requeridas. El incumplimiento de alguno de los requisitos resultaría en un resultado final incompleto y mejorable.

En primer lugar, habrá que realizar un análisis sobre las distintas redes que hacen uso de Blockchain que soportan el despliegue de *smart contracts* y elegir una de ellas. Esto tiene como objetivo familiarizarse con los conceptos de dicha cadena de bloques, ya que no todas son iguales, teniendo cada una distintas peculiaridades.

Acto seguido, se procederá a aprender el lenguaje de programación que se utiliza para la creación de los contratos inteligentes. Dicho lenguaje es *Solidity*, y se trata de un lenguaje de alto nivel que tiene un parecido razonable a *Javascript*.

A continuación, se procederá a buscar investigar sobre los oráculos, los cuales jugarán una parte fundamental para que nuestro sistema se comporte como se espera. Los oráculos son utilizados para que el contrato que se diseñe reciba información del exterior. En este caso se usarán para recopilar información acerca de eventos deportivos en los que los usuarios han apostado.

Sin embargo, debido a que existen diferentes tipos de oráculos, se deberá analizar cada tipo y determinar cuál de ellos se adapta mejor para los distintos casos de uso recién mencionados.

Una vez que se tenga una comprensión sólida de los conceptos clave, incluyendo los *smart contracts*, los oráculos, y cómo funcionan juntos, se puede proceder al desarrollo del contrato inteligente. En este proceso, se definirán las funciones y los atributos del contrato y se escribirá el código en *Solidity* siguiendo buenas prácticas de programación y seguridad.

Cuando se tenga el contrato escrito, habrá que compilarlo para que posteriormente pueda ser desplegado en una red Blockchain. En este caso se utilizará un compilador llamado *Remix*. Para desplegar un contrato inteligente en una red Blockchain, es necesario enviarlo a la red y almacenarlo en un bloque específico. Hay distintas alternativas para desplegar un contrato en la red, éstas se comentarán más detenidamente en la memoria.

A continuación, se realizarán una serie de tests para comprobar que el funcionamiento del contrato inteligente es el que se esperaba y se procederá a realizar la prueba definitiva para comprobar que el un usuario es capaz de realizar una apuesta ganadora y recibirá las ganancias correspondientes.

1.6 Alineación con ODS

Este proyecto puede alinearse con varios de los Objetivos de Desarrollo Sostenible (ODS) de las Naciones Unidas:

Anexo B

1. ODS 9: Industria, innovación e infraestructura:

   El uso de Blockchain en el sistema de apuestas en línea puede fomentar la innovación en la industria de las apuestas, al permitir la creación de soluciones más seguras y confiables para las transacciones, lo que también mejoraría la infraestructura tecnológica.

   Asimismo, se fomenta el desarrollo de la industria y atraer más inversión a la misma.

2. ODS 10: Reducción de las desigualdades:

   El uso de Blockchain puede contribuir a reducir las desigualdades al proporcionar transparencia y acceso igualitario a las apuestas en línea para todas las personas, independientemente de su ubicación o situación financiera.

   Además, el uso de esta tecnología en este proyecto puede mejorar la transparencia en las transacciones y procesos de apuestas, lo que puede reducir las desigualdades en el acceso a información y la confianza en el sistema.

3. ODS 16: Paz, justicia e instituciones fuertes:

   Al permitir una transacción segura y confiable, el uso de Blockchain en este proyecto puede contribuir a la construcción de un entorno más seguro y justo, dada la transparencia de las transacciones y procesos de apuestas.

Además, al garantizar la integridad de los datos el sistema puede ayudar a prevenir y combatir la corrupción en el sector.

Finalmente, es importante que el sistema de apuestas en línea se desarrolle y opere de manera justa e inclusiva, garantizando la igualdad de oportunidades y acceso a todos los usuarios, sin importar su origen o condición socioeconómica.

1.7 Plan de trabajo

A continuación, se incluye un cronograma con las distintas actividades recién mencionadas y la duración que se estima para su terminación.

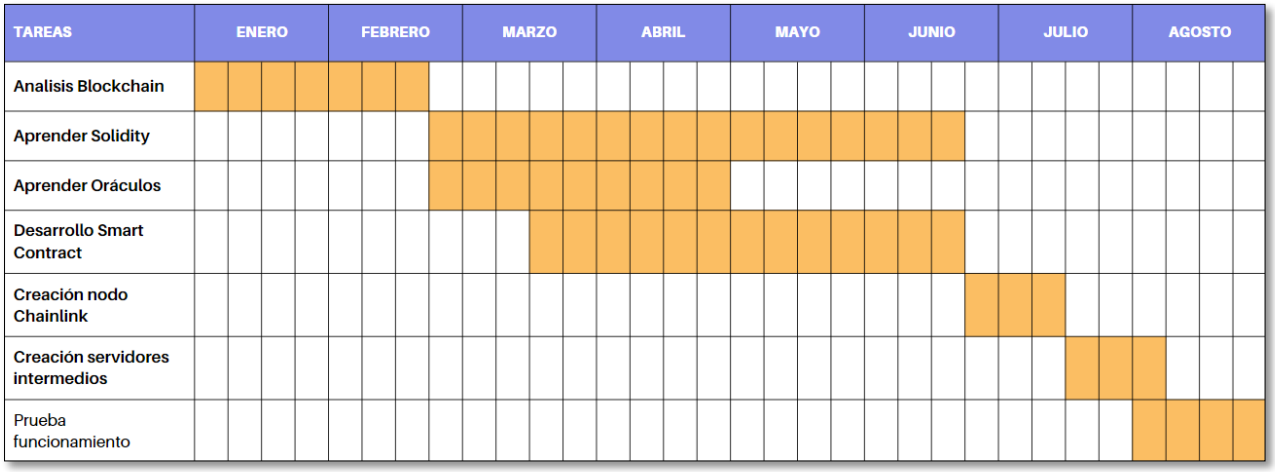

Ilustración 1: Planificación proyecto

1.8 Recursos

Se enumeran los diferentes recursos a emplear durante la realización del presente trabajo:

- Herramienta Remix para escribir, compilar y desplegar contratos inteligentes directamente desde el navegador.

Anexo B

- Oráculo Chainlink para obtener resultados de eventos deportivos a los que los usuarios han apostado.
- Azure Cloud para desplegar una máquina virtual que hospede el nodo Chainlink y dos servidores intermedios cuya misión será descrita durante el trabajo.
- Herramienta Truffle para la realización de tests unitarios que comprueben el correcto comportamiento del contrato escrito.
- Wallet Metamask para el despliegue del contrato en una red de Ethereum de prueba y la propia interacción con el contrato.

Capítulo 2

2. Blockchain

2.1 Introducción Blockchain

El mundo está dividido en lo que respecta a las criptodivisas, muchos creen que se trata de un activo difícil de valorar, ya que no generan para los propietarios de ese activo unos ingresos futuros, unos flujos de caja, de manera que no es posible saber lo que valen en realidad. Asimismo, se trata de un sector con una gran incertidumbre, ya que ¿Quién nos dice que ahora las redes que más éxito están teniendo ahora como pueden ser Bitcoin o Ethereum sean las que se vayan a imponer finalmente a medio o largo plazo?

Por otro lado, otros defienden que se trata del futuro, debido a que cada vez más nos vamos trasladando a un mundo más digital, como se puede ver ahora con la explosión del metaverso, los *eSports*, y un mayor cambio hacia la vida virtual. Actualmente, se están estudiando los beneficios de poder unir la tecnología Blockchain y las apuestas deportivas, y este trabajo tiene precisamente dicho objetivo. Sin embargo, la tecnología Blockchain puede ser aplicada a distintos campos como pueden ser cadena de suministro, salud, identidad digital, coleccionables, apuestas, etc.

Cabe destacar que, dependiendo de la red utilizada, la tecnología puede variar ligeramente.

Blockchain se traduce como cadena bloques, es decir, cada uno de los bloques que componen la cadena se encuentra encadenado con el anterior mediante el uso de criptografía, pero ¿Qué es un bloque? Lo primero que hay que saber es que la estructura de los bloques puede cambiar en función de la red que se emplee.

En primer lugar, analizaremos la estructura de un bloque en la red de Bitcoin, la cual utiliza un algoritmo de consenso llamado *Proof of Work* (PoW) del que se hablará más adelante. Un bloque es un "contenedor" que contiene un encabezado de bloque, unos datos de bloque y una firma digital.

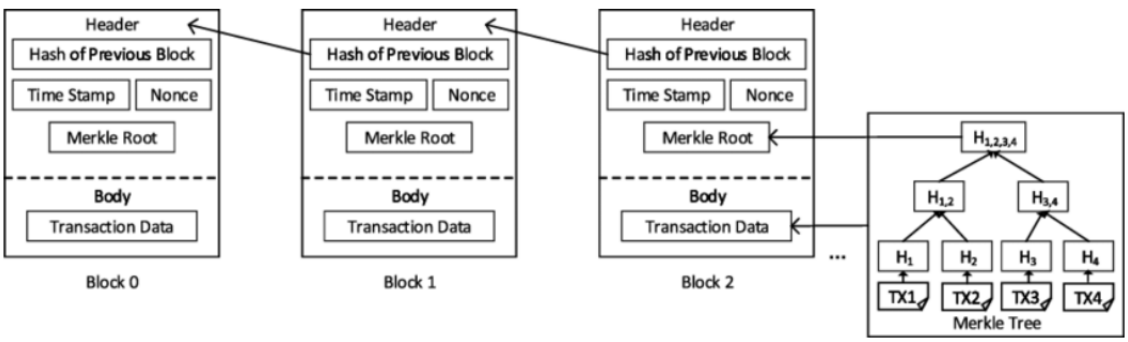

Ilustración 2: Estructura Bloque

El encabezado del bloque (*Block Header*) es la parte superior del bloque y contiene información importante sobre el bloque, como el hash del bloque anterior, la marca de tiempo de cuando se generó el bloque, el Nonce, y la raíz de Merkle. El hash del bloque anterior es lo que permite la creación de una cadena de bloques.

La raíz de Merkle es el hash raíz de un árbol de Merkle que se almacena en el cuerpo del bloque. El árbol de Merkle permite verificar si una transacción se encuentra dentro de un bloque utilizando únicamente hashes. El árbol de Merkle permite a los participantes de la red el no descargar toda la información de la cadena de bloques. De manera que se reducen los requerimientos para que una persona participe en la red al no ser necesario que cuente con un equipo con tanta capacidad de almacenamiento.

La función principal del árbol de Merkle en la cadena es permitir que los nodos de la red puedan verificar rápidamente que los datos en un bloque específico no han sido manipulados o alterados de ninguna manera. En lugar de verificar todas las transacciones del bloque, lo que sería muy ineficiente, los nodos pueden simplemente verificar la raíz del árbol de Merkle, que es un hash criptográfico único generado a partir de todos los datos incluidos en el bloque.

Si un solo byte de datos en el bloque se modifica, el hash de la raíz del árbol de Merkle cambiará drásticamente, lo que indica que los datos han sido alterados. Esto permite que los nodos de la red detecten cualquier intento de fraude o manipulación y rechacen los bloques que no pasan la verificación de la integridad.

En cuanto al Nonce, se trata de un número aleatorio de 32 bits que se utiliza en el proceso de minería de prueba de trabajo (PoW) para crear un hash válido que cumpla con la dificultad requerida para agregar un nuevo bloque a la cadena de bloques.

En términos simples, los mineros de Bitcoin compiten para resolver un problema criptográfico complejo que requiere un gran poder de procesamiento para encontrar una solución. El problema consiste en encontrar un hash que cumpla con un cierto nivel de dificultad, que se ajusta automáticamente cada 2016 bloques para mantener un tiempo promedio de 10 minutos por bloque.

El proceso de minería consiste en intentar diferentes valores del nonce junto con otros datos en el bloque, como la lista de transacciones, y aplicar la función hash SHA-256 hasta que se encuentre un hash válido que cumpla con la dificultad requerida. El valor del nonce es un número aleatorio que los mineros cambian constantemente hasta encontrar un hash válido que cumpla con los requisitos.

Finalmente, el bloque contiene una lista de transacciones. Ésta es una lista de todas las transacciones confirmadas que ocurrieron en la red de Bitcoin (o en otra red que emplee PoW como algoritmo de consenso) desde el último bloque minado. Contiene información sobre quién envió y recibió bitcoins, la cantidad de bitcoins enviados y recibidos, y las tarifas de transacción asociadas.

Cada transacción en la lista de transacciones tiene su propio identificador único conocido como "hash de transacción" que se utiliza para identificar la transacción de manera única en la red de Bitcoin.

Una vez que se agrega un bloque a la cadena de bloques de Bitcoin, todas las transacciones confirmadas en ese bloque se consideran permanentes y no se pueden eliminar o cambiar.

Esto hace que la cadena de bloques de Bitcoin sea inmutable y segura para almacenar y transferir valor de una manera descentralizada.

Ahora se procederá a hablar sobre la estructura de los bloques de la red de Ethereum.

Los componentes clave de la estructura de un bloque en Ethereum con algoritmo de consenso *Proof of stake* (PoS) son:

- Número de versión: El número de versión de la estructura de bloques que se está utilizando.
- Hash del bloque anterior: El hash del bloque anterior en la cadena de bloques, que crea la conexión entre los bloques y los convierte en una cadena.
- Número de bloque: Número del bloque actual.
- Límite de gas: Máxima cantidad de gas permitida en el bloque.
- Gas utilizado: Cantidad de gas utilizado en el bloque.
- transactions_root: Hash raíz de las transacciones en la carga útil.
- Lista de transacciones: Una lista de todas las transacciones incluidas en el bloque.
- Lista de registros de voto de consenso: Una lista de registros de voto de los validadores elegidos para validar el bloque.
- Lista de omisiones: Una lista de omisiones de validadores que no cumplieron con sus obligaciones o cometieron errores.
- Timestamp: La marca de tiempo en la que se creó el bloque.
- Firma del validador: La firma digital del validador que ha sido elegido para validar el bloque.

En *Proof of Stake* (PoS), la selección aleatoria de validadores se realiza en función de la cantidad de criptomoneda que han depositado en la red, lo que se conoce como "apuesta" o "staking". Los validadores que han “apostado” más criptomoneda tienen más probabilidades de ser elegidos para validar un bloque y, por lo tanto, recibir una recompensa por hacerlo. El funcionamiento detallado de este mecanismo de consenso de detallará más adelante.

Los hashes se derivan de los datos del bloque. Estos son una cadena codificada de letras y números que se vincula de forma única y permanente a cada bloque individual en la secuencia encadenada. De esta manera se evita el fraude, ya que un cambio en un bloque ya integrado en la cadena de bloques invalidaría la todos los siguientes (muy parecido al caso de Bitcoin mencionado anteriormente); asimismo, todos los hashes subsecuentes cambiarían y todos los nodos que ejecutasen la Blockchain lo notarían.

En cuanto al hash de identificación del bloque anterior, evita que se agregue un bloque en el medio de la cadena.

Finalmente, las transacciones no son más que las operaciones que se realizan para agregar datos a una cadena de bloques. Su contenido puede ser muy variado y puede depender de la red en la que se esté operando. En el caso de Ethereum, puede consistir en una transferencia de fondos entre un remitente y un destinatario, el despliegue de un contrato inteligente, etc.

Cada una de las transacciones se transmiten a la red, y a medida que son validadas, se juntan y ordenan para formar los bloques en una cadena de bloques.

2.2 Sistema descentralizado

Una característica de Blockchain es que todas las transacciones pueden ser verificadas por cualquier persona, de manera que la trazabilidad se garantiza. Por tanto, se trata de un registro abierto y transparente de todo el historial de una criptomoneda. Esto se debe a que la tecnología Blockchain se basa en la descentralización, de manera que la autoridad y el control se distribuyen entre una red de nodos en lugar de estar en manos de una sola entidad central. Asimismo, la descentralización también permite la creación de un ecosistema más democrático, donde todos los nodos tienen un papel igualitario en la validación de las transacciones y la toma de decisiones en la red. A continuación, se muestra una ilustración comparando un sistema centralizado de uno descentralizado:

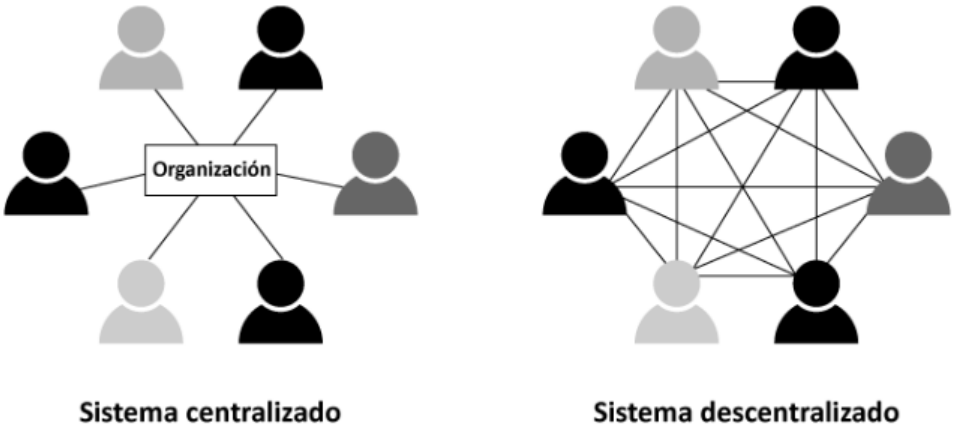

Ilustración 3: Sistema centralizado vs descentralizado

Un sistema centralizado es aquel en el que una entidad central, como una empresa o una organización gubernamental, tiene el control y la autoridad sobre el sistema y su funcionamiento. En contraste, un sistema descentralizado es aquel en el que el control y la toma de decisiones se distribuyen entre los participantes del sistema, sin la intervención de una entidad central.

Blockchain es un ejemplo de sistema descentralizado en el que las transacciones y los registros se almacenan en una red de nodos distribuidos en todo el mundo. Cada participante tiene una copia completa y actualizada del registro de transacciones, lo que aumenta la transparencia y la resistencia a la manipulación.

A continuación, se presentan algunas de las principales diferencias entre un sistema centralizado y un sistema descentralizado como Blockchain:

1. Control y autoridad: En un sistema centralizado, la entidad central tiene el control y la autoridad sobre el sistema. En un sistema descentralizado como Blockchain, el control y la autoridad se distribuyen entre los nodos de la red.

2. Transparencia: En un sistema centralizado, la transparencia depende de la voluntad de la entidad central para compartir información. Si ésta no quiere compartir esa información no le quedará otra a los usuarios que confiar en dicha entidad. Por otro lado, en un sistema descentralizado como Blockchain, la transparencia se logra mediante la verificación y validación de las transacciones por parte de los nodos de la red.

3. Seguridad: En un sistema centralizado, la seguridad depende de la capacidad de la entidad central para proteger los datos. En un sistema descentralizado, la seguridad se basa en la criptografía y la distribución de datos entre los nodos de la red, lo que hace que sea muy difícil de manipular o atacar.

4. Velocidad: En un sistema centralizado, la velocidad de procesamiento de las transacciones puede ser más rápida debido a que hay menos nodos involucrados en el proceso de validación, mientras que, en un sistema descentralizado, la velocidad puede ser más lenta debido al proceso de validación distribuida.

5. Costos: En un sistema centralizado, los costos pueden ser más bajos debido a la centralización de recursos y procesos. En un sistema descentralizado como Blockchain, los costos pueden ser más altos debido a la necesidad de más nodos para validar y verificar las transacciones.

6. Simplicidad: Los sistemas centralizados cuentan con un diseño bastante más sencillo que será más fácil de implementar. Además, estos sistemas pueden proporcionar una mayor eficiencia cuando se tienen pocos usuarios a los que dar servicio.

De esta manera, los sistemas centralizados tienen el problema llamado “Single Point of Failure” o “Punto único de fallo”. Este problema se basa en que todos los servicios o aplicaciones dependen de un solo servidor, base de datos, enrutador u otro componente crítico. Si este componente falla, todo el sistema se detendrá y no podrá proporcionar servicios a los usuarios.

Si alguien intenta manipular una transacción, hará que se rompa el eslabón y, en consecuencia, toda la red verá lo que ocurrió. Además, tanto las criptomonedas como la tecnología Blockchain que las impulsa posibilitan la transferencia de valores en línea sin la necesidad de un intermediario como un banco o una empresa de tarjetas de crédito.

2.3 Red Ethereum

Una vez introducida un poco la tecnología Blockchain, se va a pasar a hablar de una red en concreto que implemente dicha tecnología. Se trata de la red de Ethereum, y será la que elegida para la realización de este trabajo debido a que es la mayor red que permite el despliegue de *smart contracts*, los cuales jugarán un papel importante en el desarrollo del trabajo. Dichos *smart contracts* serán descritos más adelante.

Como bien se ha comentado, Ethereum es una plataforma digital que adopta la tecnología de cadena de bloques y Ether, su criptomoneda nativa, es la segunda más grande del mercado.

La plataforma Ethereum fue creada en 2015 por el programador ruso Vitalik Buterin, con la perspectiva de crear un instrumento para aplicaciones descentralizadas y colaborativas. Ether (ETH), es un token que puede ser utilizado en transacciones que usen este software. Como bitcoin, Ether existe como parte de un sistema financiero autónomo de pares, libre de intervención gubernamental.

2.3.1 Elementos principales

2.3.1.1 Ethereum Virtual Machine

Como todas las demás redes, Ethereum también está compuesta por nodos. La peculiaridad en este caso reside en que cada uno de ellos ejecuta una instancia de la EVM (Ethereum Virtual Machine).

La EVM es una entidad sustentada por miles de computadores conectadas ejecutando un cliente de Ethereum. Además, cabe resaltar que es una de las piezas claves en el funcionamiento de la cadena de bloques de Ethereum. Ésta es una máquina virtual de Turing (modelo matemático de máquina de computación hipotética que puede usar reglas predefinidas para determinar resultados a partir de variables) de 256 bits que permite a cualquiera ejecutar *Bytecode* (código de objeto computacional) de forma arbitraria. EVM es parte del Protocolo Ethereum y juega un rol crucial en el mecanismo de consenso del sistema *Ethereum.*

Las EVMs permiten la ejecución de programas o *smart contracts* con el fin de desplegar sobre dicha cadena de bloques una serie de funcionalidades añadidas para que los usuarios puedan utilizarlas **[7]**. Una vez ejecutan los *smart contracts*, envían el estado a toda la red, de manera que todos los nodos de la red procesarán esa transacción (la ejecución de un *smart contract* no deja de ser una transacción) y almacenará dicho estado de la cadena de bloques. Dicho proceso se puede observar en la siguiente imagen:

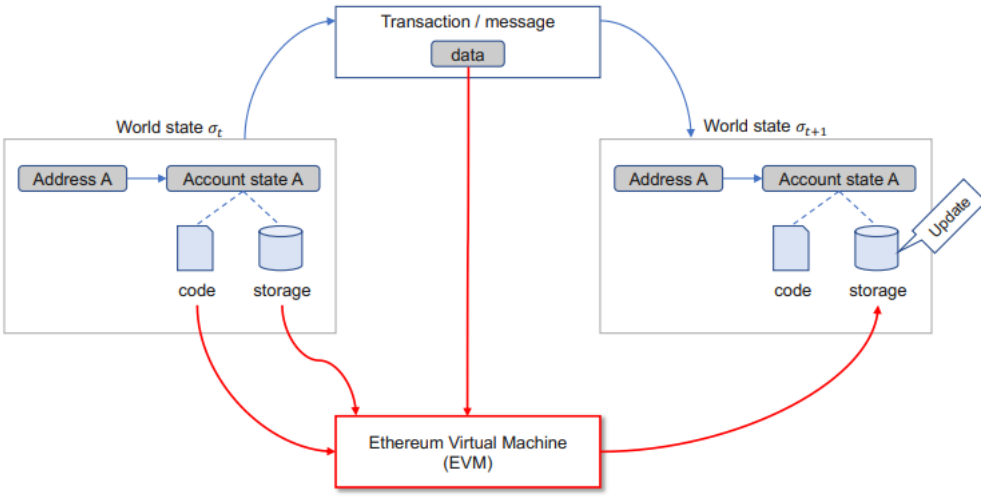

Ilustración 4: Propagación del estado a otros nodos

Dada la característica descentralizada de Ethereum y su capacidad de almacenar *smart contracts* en los nodos de la red, y que con EVM pueden ejecutarse las órdenes programadas en dichos *smart contracts*, Ethereum se convierte en un gran ordenador mundial descentralizado **[8]**.

Algunas de las principales características de las EVMs son:

- Proporciona seguridad y permite ejecutar códigos no confiables en computadoras de todo el mundo, lo que convierte a Ethereum en una red descentralizada.

- Las aplicaciones descentralizadas (DApps) y los contratos inteligentes desarrollados en la EVM son completamente descentralizados y distribuidos. No requiere de la participación de terceros, tampoco pueden ser modificadas ni alteradas.
- La EVM permite el desarrollo de una mayor cantidad de aplicaciones, y que éstas puedan ejecutarse sobre una misma red Blockchain, sin afectar otras operaciones.
- Los contratos inteligentes diseñados en la EVM son invariables y pueden ejecutarse y hacerse cumplir por sí mismos, de una manera autónoma y automática. De esta manera se eliminan los tiempos de espera típicos en los contratos tradicionales.
- La EVM es sustancialmente menos eficiente que muchas otras máquinas virtuales convencionales. Esto se debe a que principalmente su diseño se basó en la utilidad del momento y no en el alto rendimiento **[7]**. Sin embargo, le brinda mayor protección e integridad de los datos y una mayor resistencia a la censura.

Anteriormente se ha comentado que las EVMs ejecutan *smart contracts*, pero para ello se requiere que dichos *smart contracts* se encuentren en lenguaje *bytecode*, y no en lenguaje de alto nivel como puede ser *Solidity*. Es por ello que las EVMs también deben ser capaces de transformar el código que de *Solidity* (o un homólogo suyo) a *bytecode* para posteriormente proceder a ejecutarlo.

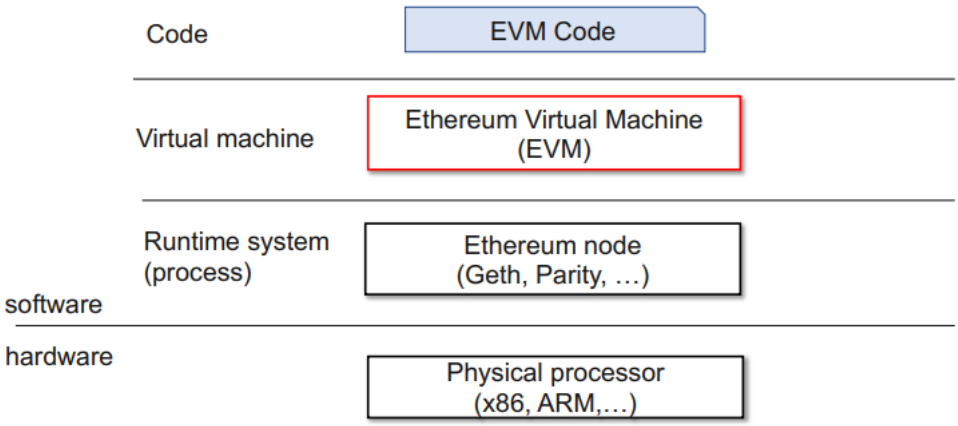

Ilustración 5: Diseño EVM

En la ilustración anterior se muestra el diseño por capas de la EVM. Se puede observar cómo en primer lugar se distingue entre una capa SW y una capa HW.

La capa de software está compuesta por tres componentes principales:

- El código, que es el contrato inteligente que se va a ejecutar.
- La máquina virtual (VM), que es responsable de ejecutar el código del contrato inteligente.
- El sistema de tiempo de ejecución (Runtime System), que proporciona el entorno en el que se ejecuta el contrato inteligente, incluyendo la memoria y el almacenamiento de datos.

Por otro lado, la capa de hardware está compuesta por el procesador físico que ejecuta la máquina virtual.

De esta manera, se puede concluir que la EVM es una plataforma de software-hardware que permite la ejecución segura de contratos inteligentes en la red Ethereum, gracias a su diseño por capas que separa la lógica del contrato inteligente de la infraestructura física que lo ejecuta.

Cabe destacar que las EVMs cuentan con una "memoria de contratos". Dicha memoria sirve para almacenar información a la que la EVM puede acceder rápidamente. Por otra parte, para almacenar datos de manera indefinida y hacerlos accesibles para futuras ejecuciones de contratos, se puede usar el "almacenamiento por contratos". Esta actúa esencialmente como una base de datos pública, desde la cual los valores se pueden leer externamente sin tener que enviar una transacción al contrato, es decir, sin comisiones.

Los *OP_CODES* son una parte muy importante y esencial de la EVM. Estos códigos de operación son los que definen las operaciones válidas que la EVM puede realizar. En las EVMs existe la capacidad de ejecutar hasta 256 *OP_CODES* distintos.

Un aspecto importante de los *OP_CODES* es que es un nivel intermedio de programación para la EVM. El primer nivel vendría dado por *Solidity* y los lenguajes de programación de alto nivel similares a este. Un segundo nivel de profundidad serían los *OP_CODES*. Por último, tendríamos el *Bytecode*, resultado de compilar los *OP_CODES* en el equivalente a lenguaje máquina de la EVM **[8]**.

Sin embargo, al ser Ethereum una Blockchain pública y ser un proyecto que aboga por la apertura y transparencia, el lenguaje *Bytecode* de la EVM se puede descompilar. Es decir, podemos transformar el *Bytecode* a *OP_CODES* y de allí llevarlo a un lenguaje cercano a *Solidity*. Esto es importante puesto que brinda a la EVM la capacidad de mantener de forma abierta y clara el contenido de un *smart contract*. Asimismo, permite reconocer la ABI de la que dispone (se describirá más adelante al hablar del proceso de compilación completo), un dato importante puesto que se trata básicamente de cómo puede codificar llamadas de un contrato para la EVM y, al revés, cómo leer los datos de las transacciones que genera dicho contrato **[8].**

2.3.1.2 Smart contract

La principal diferencia entre Bitcoin (red más conocida y utilizada en el mercado de las criptomonedas) y Ethereum es que esta última se enfoca en ser una plataforma de contratos inteligentes y aplicaciones descentralizadas. Antes de hablar sobre los *smart contracts* hay que ver primero lo que es un contrato tradicional.

Un contrato tradicional se define como un acuerdo verbal o escrito firmado por diferentes partes para llegar a un fin común sujeto a determinadas jurisdicciones. En cambio, un *smart contract* es un trozo de código que no implica la intervención de terceros para la ejecución y verificación del cumplimiento.

De esta manera, los *smart contracts* ejecutan los acuerdos establecidos de manera automática en el momento en que se cumplen las condiciones y acuerdos financieros previamente establecidos, sin la necesidad de contar con agentes externos que verifiquen el cumplimiento.

Los contratos inteligentes son un tipo de cuenta de Ethereum. Esto significa que tienen un saldo y pueden enviar transacciones por la red, es decir, cuentan con una dirección asociada.

Sin embargo, están implementados en la propia red, de manera que no están controlados por ningún usuario y se ejecutan según se hayan programado.

Un ejemplo de uso de contratos inteligentes hoy en día puede ser en el ámbito de las apuestas deportivas. Por ejemplo, si apostamos a la victoria de un equipo de fútbol, la propia casa de apuestas acudirá automáticamente a la base de datos para comprobar el resultado y las condiciones por las que se ha firmado el contrato, y en caso de que la apuesta sea determinada como ganadora, se activará el contrato y los fondos indicados en el mismo serán transferidos al monedero del ganador indicado de manera autónoma, lo que ofrece un sistema descentralizado, sin intermediarios y seguro.

Otra peculiaridad de los contratos inteligentes es que cualquiera es capaz de escribirlos y por tanto implementarlos en la red. El único requisito es que, para escribirlo, el usuario debe ser capaz de manejar uno de los lenguajes utilizados para la creación de este y tener una cantidad suficiente de ETH para implementar el contrato. Implementar un contrato inteligente es técnicamente una transacción, y en Ethereum todas las transacciones requieren de una pequeña comisión llamada gas. De esta manera, se requiere el pago de gas del mismo modo que se necesita pagar gas para realizar una simple transferencia de ETH (token de la red de Ethereum). Pero cabe resaltar que, los costes de gas (comisión) para la implementación de contratos son mucho mayores que en el caso de las transferencias.

Algunos de los lenguajes utilizados para programar un contrato inteligente son Solidity, Vyper o Yul, aunque Solidity es el más extendido. Este lenguaje de programación de alto nivel es muy similar a JavaScript y a C++.

2.3.1.3 DApps

Las DApps (Aplicaciones Descentralizadas) son aplicaciones que se ejecutan en la cadena de bloques y no requieren de una autoridad central para funcionar. Los *smart contracts* son el "backend" o motor que impulsa estas aplicaciones, permitiendo que sean descentralizadas y autónomas, ya que se controlan por la lógica escrita en el contrato, no por una entidad central.

Los contratos inteligentes se encargan de realizar las transacciones en la cadena de bloques, mientras que la DApp es la interfaz de usuario que se comunica con los contratos inteligentes. Al no depender de una entidad central, las DApps pueden ofrecer un mayor nivel de transparencia, seguridad y resistencia a la censura en comparación con las aplicaciones centralizadas tradicionales.

Al correr en una red descentralizada como Ethereum, las operaciones realizadas por las DApps se almacenan en la cadena de bloques, lo que proporciona beneficios de seguridad debido a que las transacciones están distribuidas y cifradas en la cadena de bloques, sin un punto central al que un atacante pueda atacar y obtener acceso. Además, el código de las DApps suele ser *Open Source*, lo que significa que todo el mundo puede acceder a él y mejorarlo, sin que nadie sea propietario de la aplicación.

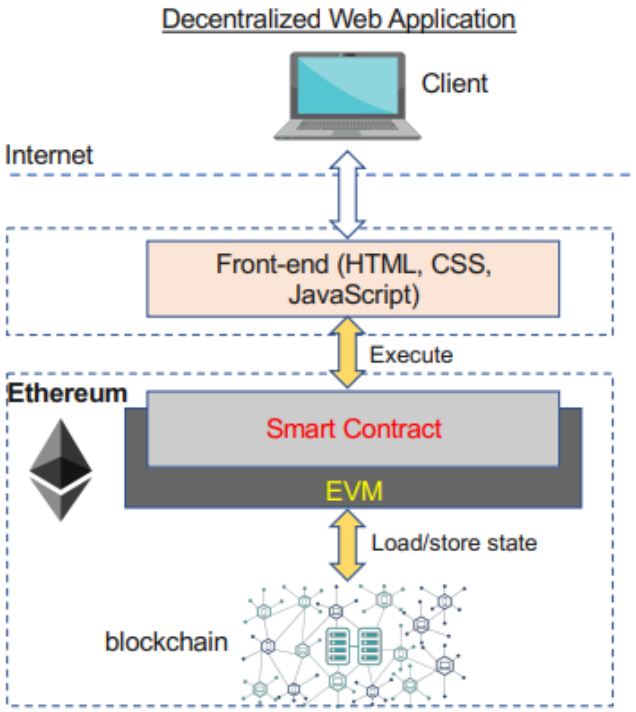

Ilustración 6: Arquitectura DApp

En la ilustración anterior se muestra la estructura de una DApp. En primer lugar, cuenta con un front-end que realiza llamadas a las funciones definidas por el back-end (*smart contract*) que corre en una red descentralizada como Blockchain. Asimismo, el *smart contract* será el encargado de interactuar con la cadena de bloques.

Anteriormente se ha comentado que la EVM realiza un proceso de compilación para pasar el contrato del leguaje de alto nivel (*Solidity*) al lenguaje de bajo nivel (*Bytecode*) y así poder interpretarlo. Sin embargo, el proceso completo es convertir el contrato primero a códigos de operación (*OP_CODES*) y finalmente a *Bytecode*.

De esta manera, si se tuviera el siguiente contrato sencillo:

```
pragma solidity 0.4.24;

contract Greeter {

    function greet() public constant returns (string) {

        return "Hola";

    }
```

La compilación lo convertiría a códigos de operación (*OP_CODES*) y de allí al *Bytecode* siguiente:

```
PUSH1 0x80 PUSH1 0x 40 MSTORE PUSH1 0x4 CALLDATASIZE LT PUSH2 0x41 JUMPI
PUSH1 0x0 CALLDATALOAD PUSH29
0x100000000000000000000000000000000000000000000000000000000 SWAP1 DIV PUSH4
0xFFFFFFFF AND DUP1 PUSH4 0xCFAE3217 EQ PUSH2 0x46 JUMPI JUMPDEST PUSH1 0x0
DUP1 REVERT JUMPDEST CALLVALUE DUP1 ISZERO PUSH2 0x52 JUMPI PUSH1 0x0 DUP1
REVERT JUMPDEST POP PUSH2 0x5B PUSH2 0xD6 JUMP JUMPDEST PUSH1 0x40 MLOAD DUP1
DUP1 PUSH1 0x20 ADD DUP3 DUP2 SUB DUP3 MSTORE DUP4 DUP2 DUP2 MLOAD DUP2
MSTORE PUSH1 0x20 ADD SWAP2 POP DUP1 MLOAD SWAP1 PUSH1 0x20 ADD SWAP1 DUP1
```

```
DUP4 DUP4 PUSH1 0x0 JUMPDEST DUP4 DUP2 LT ISZERO PUSH2 0x9B JUMPI DUP1 DUP3
ADD MLOAD DUP2 DUP5 ADD MSTORE PUSH1 0x20 DUP2 ADD SWAP1 POP PUSH2 0x80 JUMP
JUMPDEST POP POP POP POP SWAP1 POP SWAP1 DUP2 ADD SWAP1 PUSH1 0x1F AND DUP1
ISZERO PUSH2 0xC8 JUMPI DUP1 DUP3 SUB DUP1 MLOAD PUSH1 0x1 DUP4 PUSH1 0x20
SUB PUSH2 0x100 EXP SUB NOT AND DUP2 MSTORE PUSH1 0x20 ADD SWAP2 POP JUMPDEST
POP SWAP3 POP POP POP PUSH1 0x40 MLOAD DUP1 SWAP2 SUB SWAP1 RETURN JUMPDEST
PUSH1 0x60 PUSH1 0x40 DUP1 MLOAD SWAP1 DUP2 ADD PUSH1 0x40 MSTORE DUP1 PUSH1
0x5 DUP2 MSTORE PUSH1 0x20 ADD PUSH32
0x48656C6C6F0000000000000000000000000000000000000000000000000000 DUP2
MSTORE POP SWAP1 POP SWAP1 JUMP STOP LOG1 PUSH6 0x627A7A723058 KECCAK256 SLT
0xec 0xe 0xf5 0xf8 SLT 0xc7 0x2d STATICCALL ADDRESS SHR 0xdb COINBASE 0xb1
BALANCE 0xe8 0xf8 DUP14 0xda 0xad DUP13 LOG1 0x4c 0xb4 0x26 0xc2 DELEGATECALL
PUSH7 0x8994D3E002900
```

El compilador también producirá la interfaz binaria de aplicación (ABI), necesaria para que los nodos Ethereum y los clientes DApp puedan entender y comunicarse con el contrato. La ABI define la estructura de los datos y las funciones que pueden ser llamadas o ejecutadas en el contrato. Además, permite a los desarrolladores escribir aplicaciones que interactúen con los contratos inteligentes sin tener que conocer los detalles técnicos de cómo funciona el contrato. Por todo lo anterior, se puede concluir que la ABI es esencial para la interoperabilidad y la integración de los contratos inteligentes con otras aplicaciones y sistemas en la red Ethereum.

En la siguiente imagen se muestra el proceso de compilación y despliegue de forma gráfica:

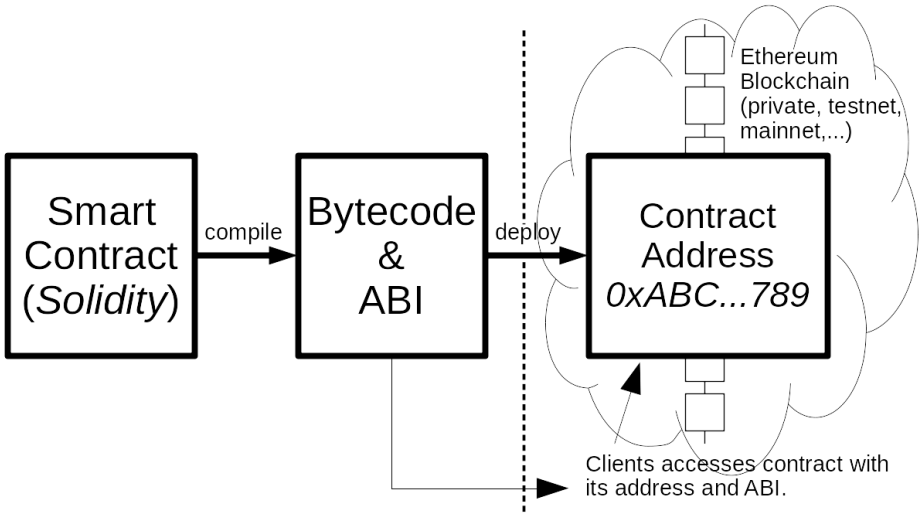

Ilustración 7: Compilación y despliegue smart contracts

Como se puede observar, el proceso de compilación producido en la EVM da como resultado el contenido del contrato en *Bytecode* y una ABI. Una vez se tiene el contrato escrito en este lenguaje de bajo nivel, se puede desplegar en la red de Ethereum.

El despliegue del contrato inteligente da como resultado una dirección única en la red Ethereum. Esta dirección es una dirección pública y se utiliza para acceder al contrato inteligente y para interactuar con él. La dirección puede ser utilizada por cualquier persona o entidad para enviar transacciones a la red Ethereum que interaccionen con el contrato inteligente. Es importante destacar que una vez que se ha asignado una dirección a un contrato, ésta nunca cambiará.

2.3.1.4 Algoritmos de consenso

Una pieza clave de la tecnología Blockchain son los algoritmos de consenso. Éstos tienen la misión de garantizar la integridad, trazabilidad y transparencia de las transacciones que llegan a la red. Además, Permiten a los nodos de la red alcanzar un acuerdo sobre el estado actual de la cadena de bloques y validar las transacciones que se incluyen en ella.

Los algoritmos de consenso también ayudan a prevenir la manipulación y la falsificación de información en la cadena, lo que a su vez garantiza la confianza en la información registrada en ella. Hay diferentes algoritmos de consenso, sin embargo, solo se comentarán el *Proof of Work* (PoW) y *Proof of Stake* (PoS) aunque anteriormente se ha hecho una pequeña introducción de cada uno a la hora de hablar sobre la estructura de los bloques que conforman la cadena.

Hasta hace unos meses, el algoritmo de consenso empleado por la red de Ethereum era *Proof of Work*. Este mecanismo consiste en que cada uno de los usuarios que está corriendo un nodo de la red de Ethereum (mineros) resuelve un problema matemático complejo para crear un bloque y obtener una recompensa en forma de criptomoneda. Anteriormente en la estructura de un bloque en Bitcoin se ha hablado de los Nonces. Los mineros tienen la función de conseguir generar un Nonce que comience por un número determinado ceros. El número de ceros que debe tener dependerá de la dificultad actual que esté fijada en la red. Dicha dificultad variará en función del tiempo que se ha tardado en generar los bloques anteriores. Las redes tienen un tiempo de generación de bloque que quieren cumplir. De esta manera, si los bloques anteriores se han pasado del tiempo objetivo de generación de bloque, la dificultad se verá reducida. Con este cambio, el número de ceros que debe obtener el Nonce al principio será menor, por lo que a priori será más fácil encontrarlo y se reducirá así el tiempo de generación de bloque. De esta manera se conseguirá estar más cerca del tiempo de generación del bloque objetivo.

Cabe resaltar que los mineros compiten entre ellos para resolver el acertijo, ya que el primero que lo resuelva será el que obtenga la recompensa compuesta por cada una de las comisiones que contienen las distintas transacciones que forman el bloque que se ha creado.

Cuando un usuario lleva a cabo una transacción en la cadena de bloques, éste tiene que incluir una pequeña comisión. Pero esta transacción no se materializa al instante, sino que es añadida a un pool de transacciones pendientes de ser incluidas en un bloque. Los mineros cogen las transacciones de dicho pool de transacciones pendientes de materializarse, las agrupan, resuelven el acertijo matemático relacionado con el Nonce mencionado anteriormente y envían el bloque creado al resto de la red para que sea validado. Si dicho bloque es validado de forma exitosa, dicho bloque se añade a la cadena y las transacciones de su interior se habrán hecho efectivas.

Dicho esto, dado que un minero escoge las transacciones que desea del pool de transacciones pendientes, es normal que cada uno quiera agrupar las transacciones que incluyen una mayor comisión, ya que si consiguen crear un bloque que contenga dicha transacción se quedarán la comisión. Es por ello que, las transacciones que contengan una mayor comisión se van a llevar a cabo más rápidamente que las que tienen comisiones menores. De esta manera, la comisión que pagan los usuarios al realizar una transacción para que ésta se realice rápidamente se le denomina comisión por prioridad, ya que proporciona a los mineros un incentivo para incluir la transacción en el bloque. Además de la rapidez con la que se quiere que se procese la transacción, la cantidad de comisión también dependerá del estado de congestión actual de la red.

Anteriormente se ha dicho que una vez un minero ha agrupado las transacciones y ha resuelto el acertijo, el resto de los mineros deberán verificar que efectivamente el bloque generado es válido. De esta manera, es muy complicado crear nuevos bloques que borren las transacciones o creen transacciones falsas. Para que esto se diera, el minero malicioso tendría que resolver el acertijo más rápido que el resto, y para que fuera validado tendría que contar con más del 51% de la potencia de minado de la red para vencer a todos los demás.

Debido a que los mineros trabajan de forma descentralizada, es posible minar dos bloques válidos al mismo tiempo. Esto crea una bifurcación temporal en la cadena, pero solo se considerará como legítima aquella en la que se haya añadido un bloque detrás, lo que la haría más larga que la otra rama.

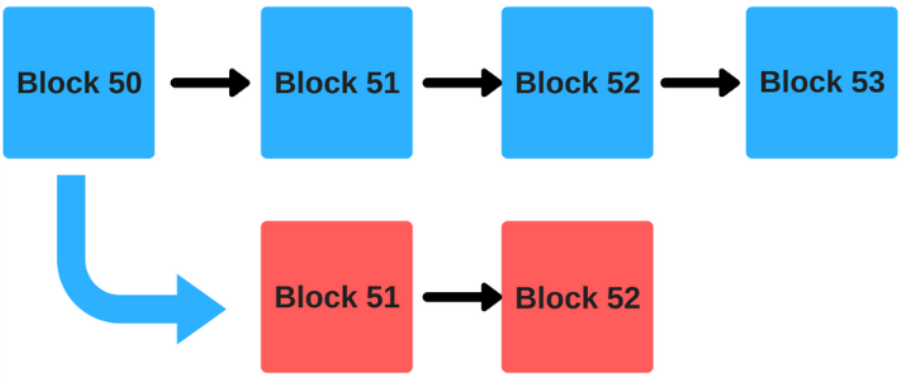

Ilustración 8: Ejemplo de bifurcación de la cadena de bloques

Cabe resaltar que una transacción se da como finalizada cuando ésta es irreversible, y para que esto suceda debe pertenecer a la cadena más larga. De esta manera, en el momento que hay una bifurcación las transacciones que se encuentran en los bloques de las distintas ramas no se consideran aún como finalizadas ya que una de las ramas dejará de existir (la más corta).

La principal limitación del algoritmo de consenso *Proof Of Work* es el gasto energético que requiere. Para mantener la seguridad y la descentralización, Ethereum en la prueba de trabajo (PoW) antes consumía 73,2 TWh al año, el equivalente energético de un país mediano como Austria.

Debido a la desventaja recién comentada, Ethereum ha cambiado a utilizar el algoritmo de consenso llamado *Proof Of Stake*. Este algoritmo trata de superar además las limitaciones de escalabilidad de las redes PoW, además de reducir el alto consumo de energía y la dependencia de hardware especializado. Si bien PoW y PoS comparten el mismo objetivo de llegar a un consenso en la cadena de bloques, PoS tiene una forma diferente de determinar quién valida un bloque de transacciones. Lo primero que llama la atención aquí es que en las redes que implementan este algoritmo no existen los mineros, de manera que, en lugar de depender de computadoras potentes para competir por los derechos de validación de bloques, los validadores de PoS dependen de la cantidad de monedas de la criptodivisa correspondiente que tengan en *staking*.

El concepto de *staking* consiste en que los participantes bloquean una cantidad de monedas en un contrato inteligente específico en la cadena de bloques, y según la cantidad de monedas que tengan los usuarios en *staking* serán elegidos o no para validar el siguiente bloque. Cuantas más monedas se tengan en *stake*, mayor será la probabilidad de que el usuario sea seleccionado para validar el bloque y obtener como recompensa las comisiones de la

transacción del bloque que validó, pero no obtiene una comisión por resolver el problema matemático para generar el bloque como si ocurre en las redes que implementan PoW.

2.3.1.5 Comisión (Gas)

Como se ha comentado anteriormente, en Ethereum esta comisión de denomina gas. Éste indica la cantidad de recursos computacionales que requiere una transacción para ejecutarse. Las comisiones de gas se pagan en la moneda nativa de Ethereum, aunque los precios están indicados en Gwei, que es una denominación de ETH; cada Gwei equivale a 0,000000001 ETH (10^{-9} ETH). Por ejemplo, en lugar de decir que el gas cuesta 0,000000001 Ether, puedes decir que cuesta 1 Gwei. La palabra «gwei» significa «giga-wei» y es que igual a 1.000.000.000 wei. El Wei es la unidad más pequeña de ETH **[9]**.

En Ethereum existe también el concepto de *maxFeePerGas* que puede ser indicado a la hora de realizar una transacción. Este parámetro es opcional e indica la máxima cantidad que el usuario está dispuesto a pagar para que su transacción se lleve a cabo. Para que una transacción se ejecute, la comisión máxima debe ser superior a la suma de la comisión base y la propina. Al usuario que realizó la transacción se le devolverá la cantidad de comisión que no haya sido utilizada.

Las transacciones más complicadas que implican contratos inteligentes requieren mayor trabajo computacional, por tanto, requieren un límite de gas más elevado que un simple pago.

Los bloques generados tienen un tamaño esperado de 15 millones de gas, pero esto puede variar de acuerdo con la demanda de la red. El protocolo alcanza un punto de equilibrio alrededor del tamaño de bloque de 15 millones promedio, a través del proceso de *tâtonnement*. Esto significa que, si el tamaño del bloque es mayor que el tamaño esperado, el protocolo aumentará la comisión base para el siguiente bloque. De manera similar, el protocolo disminuirá la comisión base si el tamaño del bloque es menor que el tamaño esperado. La cantidad con respecto a la que se ajusta la comisión base es proporcional a la diferencia entre el tamaño del bloque actual y el tamaño esperado **[9]**.

2.3.1.6 Wallets

Cuando se habla de Blockchain, es natural introducir el tema de las *wallets*, ya que las *wallets* son una parte fundamental de la experiencia de usuario en el ecosistema de cadena de bloques.

Una *wallet*, o billetera, es una aplicación que almacena un par de claves: pública y privada. Estas claves ayudan a cerciorar que una transacción fue realmente firmada por el remitente y prevenir falsificaciones. La clave privada es lo que se utiliza para firmar mensajes y transacciones, así que garantiza la custodia de los fondos relacionados con tu cuenta. Esto evita que actores maliciosos difundan transacciones falsas a la red, ya que siempre se puede verificar el remitente de una transacción. Dicho de otra manera, una clave privada es como una contraseña que da a su propietario acceso a sus activos digitales o los medios para interactuar de otro modo con las diversas capacidades que ahora soportan las cadenas de bloques.

Cuando se quiere crear una *wallet*, la mayoría de las bibliotecas generan una clave privada aleatoria. Una clave privada consta de 64 caracteres hexadecimales y se puede cifrar con una contraseña.

Ejemplo:

`ffffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd036415f`

Cabe destacar que las claves públicas y privadas se encuentran relacionadas. Es decir, de la clave privada se saca la clave pública mediante el uso del algoritmo de firma digital de curva elíptica.

Es posible obtener nuevas claves públicas a partir de nuestra clave privada, pero no se puede obtener la clave privada a partir de las claves públicas. Esto significa que es vital mantener una clave privada segura. Es muy recomendable mantener la clave privada en forma física en lugar de en cualquier lugar en línea, ya que solo hará que esté más expuesta a riesgos potenciales.

Se necesita una clave privada para firmar mensajes y transacciones, lo que genera una firma. A continuación, otros pueden utilizar la firma para derivar la clave pública y autenticar así al autor del mensaje **[18]**.

Actualmente existen distintos tipos de *wallets*. La decisión sobre cuál es la mejor depende de cada necesidad y uso. De hecho, cada tipo de billetera se adecúa mejor a una finalidad específica y a un perfil de usuario. A continuación, se indican los tipos de *wallets* más populares:

- **Hardware *wallets***: Son pequeños dispositivos físicos parecidos a un *pendrive* que se conectan a través de puertos USB y que se encargan de almacenar las claves privadas de manera offline, lo que limita los vectores de ataque con los que un atacante puede hacerse con las claves.
  Dado que la validación de las transferencias es realizada por la propia *wallet* de hardware de forma externa, su uso es seguro incluso en dispositivos infectados.
- **Paper *wallets***: Son simplemente una clave privada impresa en un pedazo de papel. Aunque parezca anticuado, este método se considera uno de los más seguros, ya que no está conectado a Internet.
- **Software *wallets***: Aquellas que se ejecutan en dispositivos electrónicos, como computadoras o dispositivos móviles, y que no requieren hardware adicional para su uso- Se pueden dividir en distintos tipos:
  - **Desktop *wallets***: Se instalan en un ordenador de escritorio y permiten al usuario tener el control total de sus fondos en una única aplicación. Este tipo de billetera puede ser descargada e instalada en sistemas operativos como Windows, macOS o Linux, y generalmente ofrecen más seguridad y privacidad en comparación con otras opciones de billeteras como las online o móviles
  - **Mobile *wallets***: Son aplicaciones móviles que se utilizan para almacenar y administrar criptomonedas en dispositivos móviles como teléfonos inteligentes y tabletas.
    Las wallets móviles pueden ser de dos tipos: con control total de claves privadas (non-custodial) o con control parcial de claves privadas (custodial). En las wallets non-custodial, el usuario es el único responsable de la gestión de sus claves privadas y, por lo tanto, de la seguridad de sus criptomonedas. En las wallets custodial, las claves privadas son administradas por un tercero, lo

que significa que el usuario cede el control de sus criptomonedas a esa tercera parte

- **Web *wallets***: se ejecutan en un navegador web y se pueden acceder desde cualquier dispositivo con conexión a Internet. Son convenientes porque no requieren la descarga de software adicional. Sin embargo, también pueden ser menos seguras que otras opciones de *wallets* debido a la posibilidad de ataques de phishing y malware.

En este proyecto se va a utilizar la *wallet* MetaMask. MetaMask es una extensión o plugin para navegadores web que permite a los usuarios interactuar fácilmente con las DApps de la cadena de bloques de Ethereum. Asimismo, cuenta con una aplicación que puede ser descargada tanto para Android como para IOS. De esta manera, se la puede clasificar tanto como una *Mobile wallet* como una *Desktop wallet* de tipo software

MetaMask fue creada para ser un monedero para Ethereum y una herramienta para interactuar con DApps. Para lograr ambos cometidos, MetaMask establece un canal de comunicaciones entre la extensión y la DApp en cuestión. Una vez que la aplicación reconoce que MetaMask está presente, se habilita y puede ser utilizada por el usuario **[25].**

En el proceso de creación del monedero, hay que indicar una contraseña. Posteriormente, esa contraseña será combinada con una frase semilla para obtener como resultado la clave privada. La contraseña se utiliza para cifrar la frase semilla, que a su vez se utiliza para generar la clave privada.

La frase semilla, es una serie de 12 o 24 palabras en un orden específico que se utilizan para derivar una clave privada. Esta frase es única para cada usuario y se utiliza para restaurar la cuenta en caso de pérdida de acceso a la cuenta.

La contraseña es una cadena de texto elegida por el usuario que se utiliza para proteger la cuenta de accesos no autorizados. La combinación de la frase semilla y la contraseña se utiliza para desbloquear la clave privada y realizar transacciones. Por lo tanto, es importante elegir una contraseña segura y guardar la frase semilla en un lugar seguro y privado.

2.4 Blockchain aplicado a este caso de uso

Ahora que se ha introducido un poco la tecnología Blockchain y la red de Ethereum, se va a comentar el caso de uso de Blockchain para este proyecto: apuestas deportivas. Actualmente, ya existen múltiples plataformas de apuestas que aceptan como método de pago las criptomonedas sin comisiones adicionales, además de aceptar varios tokens distintos.

Sin embargo, el realizar apuestas con criptomonedas es distinto que aplicar la tecnología Blockchain a dicho sector. Algunas características que podrían hacer de Blockchain una tecnología útil para las apuestas deportivas son que cuenta con un almacenamiento de registros, lo cual otorga transparencia al ser accesible por todo el mundo, y cuenta con seguridad tecnológica. En este caso, Blockchain podría añadir nuevas capas a la protección de datos de clientes y apostantes.

Las casas de apuestas online siempre han dominado el negocio de las apuestas en línea. Ellos determinan las cuotas, recolectan las apuestas de los usuarios y dan la recompensa a los ganadores. Además, son los que fijan las políticas a su gusto, beneficiándose de las mismas.

También pueden imponer tasas elevadas cuando los usuarios quieren retirar grandes cantidades de dinero. Sin embargo, al introducir la característica de la descentralización, lo recién comentado puede cambiar.

Actualmente, existen muchas aplicaciones de apuestas deportivas que están adoptando la tecnología Blockchain para, además de conseguir las ventajas mencionadas anteriormente, automatizar con precisión los datos en tiempo real mediante el uso de contratos inteligentes y oráculos (se hablará de ellos posteriormente), los cuales son capaces de ayudar en una variedad de formas que incluyen pagos, confirmaciones de apuestas, análisis de eventos pasados (debido a la característica de que todo queda registrado en la cadena de bloques) y monitoreo. Asimismo, como se ha comentado, Blockchain ofrece también el anonimato de los usuarios que realizan las apuestas, lo que les permite por ejemplo evitar consideraciones legales de un país específico en el que las apuestas deportivas no estén permitidas. Esto se debe también a la descentralización, la cual permite a los usuarios apostar independientemente de su ubicación.

Al principio se ha comentado que unas de las principales ventajas de la tecnología Blockchain es que elimina la necesidad de que participen terceros en el proceso. En el sector de las apuestas deportivas ocurre lo mismo, esto permite a las plataformas de apuestas deportivas generar mayores ingresos, ofrecer incentivos a los usuarios que apuestan y reducir bastante la cantidad deducida de los usuarios ganadores.

Por ejemplo, unos terceros que participan en el proceso de apostar mediante una casa de apuestas tradicional pueden ser los procesadores de pago. Éstos son empresas que procesan los pagos realizados por los clientes al hacer apuestas deportivas. Estos procesadores de pagos pueden incluir tarjetas de crédito, billeteras digitales, transferencias bancarias, criptomonedas y otros métodos de pago en línea.

Asimismo, la realización de apuestas deportivas mediante Blockchain puede eliminar límites de transacción, cantidades mínimas para realizar un retiro, restricciones en cuanto a la cantidad que se desea depositar, así como una mayor eficiencia en la transacción.

A continuación, se recogen algunas de las ventajas que se han comentado sobre la utilización de Blockchain y *smart contracts* en sistemas de apuestas en línea en comparación con los sistemas tradicionales y se suman algunas otras:

- **Confirmaciones de apuestas**:
    - En el sistema de apuestas tradicional, puede haber numerosos errores a la hora de confirmar las apuestas del apostante. Este puede ser debido a que las bases de datos pueden estar temporalmente apagadas o haber sufrido algún tipo de anomalía que las haga estar inoperativas durante un determinado tiempo. En el caso de la utilización de Blockchain, se puede proporcionar confirmaciones de apuesta, así como precisión para todas las apuestas que se han realizado antes o durante el evento deportivo. Estas apuestas serán confirmadas y permanecerán inmutables.
    - Asimismo, la transparencia puede ayudar a prevenir la corrupción al garantizar que todas las transacciones y resultados sean inmutables e inalterables. Esto significa que ningún actor puede manipular el sistema o los resultados de las apuestas a su favor. Además, el hecho de que los datos estén descentralizados y distribuidos en una red de nodos confiables hace que sea más difícil para un solo individuo o grupo de manipular los datos. Todo esto ayuda a crear un

entorno más justo y confiable para los usuarios y a prevenir la corrupción en el sector de las apuestas en línea.

- **Seguridad**:
  - Blockchain se encarga de securizar el proceso de realización de apuestas a múltiples niveles. En primer lugar, cada bloque generado es ordenado cronológicamente al final de cada cadena. Además, una vez un bloque ha sido añadido, es muy complicado (prácticamente imposible) manipular el contenido del bloque, lo que ofrece transparencia. Esta transparencia permite a cualquier persona verificar la integridad de las apuestas y los resultados, y asegura que los pagos sean realizados de manera justa y segura.

- **Pagos rápidos**:
  - En las casas de apuestas tradicionales, pueden pasar semanas hasta que una persona recibe lo que le corresponde por haber ganado una apuesta ya que si la cantidad es elevada se le realizan transferencias, las cuales no son inmediatas. Con Blockchain este tiempo de espera se verá reducido.

- **Reducción error humano**:
  - En el sistema de apuestas tradicionales hay muchas posibilidades de que errores técnicos y humanos detengan un proceso que, de otro modo, sería fluido. Sin embargo, con la utilización de Blockchain y los *smart contracts* automatizados se reduce la posibilidad de errores humanos y técnicos. La red de nodos y la transparencia de la cadena de bloques ayudan a verificar la legitimidad de las apuestas y a mantener un registro permanente y confiable de las transacciones. Al minimizar la intervención humana y la posibilidad de manipulación, se mejora la confiabilidad y la transparencia del sistema.

Ilustración 9: Ventajas smart contracts

Dadas todas las ventajas que se han comentado, este modelo de realización de apuestas ha llamado la atención de apostantes habituales, haciéndose que se replanteen la utilización de las plataformas que se apoyan en la tecnología Blockchain.

Ahora vamos a ver qué papel juegan los *smart contracts* en un sistema de apuestas que utiliza Blockchain. Para ello hay que recordar las tres funciones que debería tener un contrato inteligente en este caso de uso:

- **Almacenan ciertas reglas**
- **Verifican las condiciones de las reglas**: se verifica si el evento que tendría que pasar realmente ha sucedido o no.
- **Se auto ejecutan sin la intervención de un humano**: una vez el contrato verifica que se han cumplido las condiciones que proclaman al usuario como ganador del juego, se le darán las ganancias correspondientes. No hay humano necesarios para aprobar o declinar el contrato.

Aplicando los *smart contracts* a las apuestas, se puede decir que, si un jugador gana a determinado juego, el *smart contract* dará al usuario los fondos que le corresponden.

Un contrato inteligente no puede ser amañado dado que no está gobernado por una empresa o una persona, sino que está gobernado por la cadena de bloques.

Como bien se ha hecho hincapié anteriormente, una característica de los contratos inteligentes es que éstos son inmutables. Es decir, no puede ser cambiados una vez han sido creados y almacenados en la Blockchain. De esta manera, un casino o una casa de apuestas no pueden cambiar las reglas del juego. Los contratos inteligentes están compuestos por líneas de código que no pueden ser alteradas pase lo que pase.

Los *smart contracts* son distribuidos y descentralizados. De manera que el contenido de uno de ellos y la legitimidad de este puede ser corroborada por cualquiera en la red. En una casa de apuestas en línea convencional, todos los tratos de un jugador son sólo con la casa de apuestas. Nadie sabe lo que ocurre realmente “detrás de las cámaras”. Las casas de apuestas tienen más conocimientos que el jugador. Por lo tanto, la casa de apuestas tiene más control y poder que dicho jugador. Sin embargo, cuando se trata de una casa de apuestas impulsada por Blockchain el jugador está protegido porque todo el mundo en la red vela por él. La casa de apuestas no tiene el control. El jugador y la casa de apuestas tienen la misma cantidad de conocimiento, control y poder. Todo el mundo en la cadena de bloques puede ver y verificar el contrato, no sólo el casino **[14]**.

Una vez expuestos algunos argumentos sobre la utilización de *smart contracts* en cuanto al juego, se pueden resaltar las siguientes ventajas:

- No pueden ser manipulados una vez hayan comenzado.
- La casa de apuestas y el jugador están al mismo nivel.
- Los contratos inteligentes están completamente automatizados y son capaces de retener fondos.
- El jugador tiene la certeza de que no está siendo víctima de un timo, ya que todo es transparente.
- El jugador puede permanecer en el anonimato.
- Todo es descentralizado, nadie puede interferir en la ejecución de un contrato inteligente (gobierno, persona, empresa, etc).

- Se puede cobrar una apuesta automáticamente en cualquier momento, no solo durante horas de trabajo.
- Aumento del negocio (más clientes y volumen de transacciones) gracias a la mejora del rendimiento operativo (transacciones más rápidas y fiables).

Anteriormente se ha hablado de como los *smart contracts* deben ser capaces de recibir los resultados de los eventos deportivos a los que los jugadores han apostado. Pero, los protocolos de las cadenas de bloques y los *smart contracts* están limitados a solo obtener información y acceder a datos que estén dentro de la misma red. Esta limitación ha llevado a los desarrolladores a crear los oráculos. Éstos se pueden describir como un servicio que envía y verifica información del mundo real que sea relevante para una cadena de bloques o para un *smart contract*. Todo esto en forma de datos electrónicos que pueden ser asimilados por las redes que los sostienen.

2.5 Oráculos

Los oráculos son líneas de código que conectan información del mundo real con contratos inteligentes y otros acuerdos en una cadena de bloques. Sirven como un puente entre la cadena y los datos fuera de la cadena.

Los oráculos en sí mismos no son la fuente de información del mundo real, sino que la recopilan de las bases de datos existentes y comunican los datos de manera confiable a la cadena de bloques. La relación entre oráculos y cadenas de bloques es recíproca. Los oráculos pueden recibir datos en cadena para distribuirlos a aplicaciones externas **[20]**.

En este trabajo simplemente se va a utilizar un oráculo para que el smart contract pueda consultar la información de los resultados de los distintos partidos/eventos deportivos.

Cabe destacar que la información que llega a la cadena procedente de un oráculo puede desencadenar una acción específica en caso de que haya un contrato inteligente programado que esté esperando requiere de ese tipo de evento para que se ejecute.

En el caso del oráculo que se utilizará, será necesario que el *smart contract* capture el evento asociado a la recepción de los datos del oráculo para se active y libere los fondos después que se cumplan las condiciones predeterminadas del acuerdo.

Con el uso de los oráculos, el potencial de los *smart contracts* se extiende hasta el mundo real.

Los oráculos recopilan y verifican la información desde varias páginas webs para crear un consenso y dar un resultado preciso que no pueda ser manipulado. Asimismo, no requieren intermediación de una plataforma centralizada que pueda hackearse, engañar a sus clientes o censurarse.

Por todas las razones comentadas, se puede concluir que los oráculos desempeñan un papel fundamental en la ejecución de los contratos inteligentes al aportar datos externos a la ecuación. Sin embargo, también pueden presentar ciertos riesgos, como la posibilidad de manipulación de los datos de entrada o la posibilidad de que el oráculo sea hackeado o comprometido de alguna otra manera.

Para mitigar estos riesgos, se pueden utilizar diversas técnicas, como la elección de oráculos confiables y seguros, la utilización de múltiples oráculos para verificar los datos o la implementación de mecanismos de seguridad adicionales en los contratos inteligentes para

detectar y prevenir posibles manipulaciones de datos. Además, se pueden utilizar técnicas de cifrado y autenticación para garantizar la integridad y autenticidad de los datos transmitidos desde el oráculo al contrato inteligente. Sin embargo, este tema no será tratado en el trabajo.

Ahora, se analizarán los distintos tipos de oráculos existentes para ver cuál de ellos se adaptaría mejor al caso de uso de este proyecto.

La primera clasificación que se puede hacer es si se trata de un oráculo de tipo hardware o tipo software:

- Los oráculos software son los que manejan información en línea, pueden recopilar y ordenar información en línea y luego esos datos son enviados al *smart contract*. De manera que dicho *smart contract* podrá realizar las acciones para las que fue programado
- En cuanto a los oráculos de hardware, éstos son dispositivos físicos que pueden rastrear objetos del mundo real. Esto se debe a que algunos *smart contracts* requieren de información del mundo real.
  Los oráculos de hardware ofrecen un mayor grado de seguridad y privacidad en comparación con los oráculos de software, ya que los datos se almacenan y procesan en dispositivos dedicados y seguros. Además, al ser dispositivos físicos, también se pueden implementar medidas adicionales de seguridad, como cifrado de datos y autenticación de usuarios.

Otra clasificación que se puede hacer en cuanto a los oráculos depende del sentido en el que los datos son enviados. Si los datos son enviados desde el exterior hacia la cadena de bloques o *smart contract* se trata de un oráculo entrante. En caso contrario se tratará de un oráculo saliente.

Finalmente, también existen oráculos de consenso, los cuales se utilizan en sistemas Blockchain para resolver controversias y tomar decisiones en la red. Los oráculos de consenso funcionan como un mecanismo de votación, donde un grupo de nodos seleccionados en la red participan en la votación para decidir sobre una determinada acción o evento.

En nuestro caso de uso es normal pensar que se utilizarán oráculos entrantes para obtener el resultado de los partidos a los que los usuarios han apostado. Asimismo, se tratará de un oráculo de tipo software debido a que este tipo de oráculos se ejecutan en un sistema informático y se comunica con fuentes de datos externas a través de conexiones a internet o a otros sistemas informáticos. Mientras que, como se ha mencionado anteriormente, un oráculo de hardware utiliza dispositivos físicos para recopilar información de fuentes externas, como sensores, cámaras, micrófonos u otros dispositivos electrónicos.

En el ámbito del trabajo, si un usuario desease hacer una apuesta sobre el resultado de un partido de fútbol, el oráculo entrante se encargaría de recopilar y verificar el resultado del partido. Una vez verificada la información, se enviaría dicho resultado a la cadena de bloques y se utilizaría para determinar si coincide con el resultado introducido por el jugador en la apuesta, y en ese caso se liberarían los fondos correspondientes a los ganadores.

Este enfoque garantiza que la información utilizada para determinar el resultado de la apuesta sea precisa y verificable, y que la plataforma de apuestas sea justa y transparente para todos los usuarios. Además, al ser una plataforma basada en Blockchain, los resultados de las apuestas y las transacciones financieras son inmutables y seguros.

Anteriormente se ha visto que existen varios tipos de oráculos y, de la misma manera, también existen diferentes redes descentralizadas de oráculos. En este proyecto se utilizarán los oráculos Chainlink.

Chainlink es una red de oráculos descentralizados que se utiliza para conectar las aplicaciones basadas en Blockchain con los datos y servicios externos. Chainlink se ejecuta en una red de nodos que se encargan de recopilar y verificar la información relevante, y luego la envían a la cadena de bloques. Antes de ser enviada a los *smart contracts*, Chainlink comprueba la calidad de la información proporcionada por cada oráculo; descartando (y penalizando) a los nodos que entregan datos de baja calidad. Chainlink se utiliza para resolver uno de los problemas más grandes de la tecnología Blockchain, que es la falta de acceso a datos externos confiables y verificables. Con Chainlink, los contratos inteligentes pueden acceder a una amplia variedad de fuentes de datos, incluyendo precios de activos financieros, clima, resultados deportivos, entre otros. Cabe destacar que cuando conectas un contrato inteligente a servicios del mundo real o a datos fuera de la cadena, creas un contrato inteligente híbrido.

Según qué información se quiere obtener del exterior, el nodo Chainlink deberá conectarse a distintas fuentes de datos. Por ejemplo, en el caso de querer obtener un resultado de un evento deportivo en tiempo real se puede utilizar el nodo *SportsDataIO*. A continuación, se muestra gráficamente cual sería el proceso que se seguiría cuando el nodo envía la información al contrato:

Ilustración 10: Información del nodo transferida al smart contract

En la imagen anterior se muestra el nodo Chainlink de SportsDataIO, y la alimentación de datos a la red Blockchain.

SportsDataIO proporciona datos sobre decenas de miles de eventos deportivos cada año. Sirviendo a todos los campos de datos deportivos, incluidos los clientes de fantasía, apuestas deportivas, transmisión, medios de comunicación y modelos predictivos en todo el mundo. El compromiso de SportsDataIO con la alimentación de datos de alta calidad en varios formatos les ha valido el premio *FSGA Best Sports Data Providers* de 2018 y 2019, con clientes como *Fox Sports, Sports Illustrated, The Athletic* y *Eurosport* consumiendo sus datos. Asimismo, destinan recursos considerables a verificar y organizar datos deportivos y combinarlos con noticias e imágenes procedentes de sus docenas de socios de datos para ofrecer a las aplicaciones

información relevante y atractiva. De esta manera, se puede concluir que SportsDataIO es una fuente de datos bastante fiable.

Actualmente han decidido dar el salto y apoyar a los mercados emergentes de Blockchain proporcionando sus estadísticas deportivas de primera calidad, resultados de partidos, probabilidades y otros conjuntos de datos directamente a las aplicaciones de Blockchain a través de su nodo Chainlink **[26].**

Como se ha comentado anteriormente, Chainlink es un oráculo altamente escalable y confiable, ya que se basa en una red de nodos descentralizados y un proceso de consenso de varios nodos. Esto garantiza que la información utilizada por los contratos inteligentes sea precisa y verificable.

En la actualidad, Chainlink opera sobre la red de Ethereum, pero está preparada para funcionar en cualquier cadena de bloques. Es por esta razón, entre otras, que se ha optado por esta red de Oráculos.

Un último comentario sobre Chainlink es que el contrato debe pagar a los nodos que recuperan la información del exterior una cantidad de tokens LINK. Estos tokens también son utilizados para los depósitos realizados por los operadores de nodos según lo requerido por los creadores de contratos. La denominación más pequeña de LINK se denomina Juel, y 1.000.000.000.000.000.000 (1e18) Juels equivalen a 1 LINK. Esto es similar a Wei, que es la denominación más pequeña de ETH.

De esta manera, a la hora de desplegar el contrato, éste debe indicar la dirección de token LINK del nodo para que sea posible la recuperación de la información del exterior, así como la dirección del contrato oráculo.

La *wallet* del usuario que desea realizar una apuesta realizando una llamada al *smart contract* deberá contar con dicha cantidad de tokens LINK además de la cantidad de Ether que desea apostar.

Como se ha comentado, en este trabajo se va a utilizar el oráculo para que extraiga la información de las bases de datos de SportsDataIO. Sin embargo, para que esto sea posible hay que llevar a cabo una serie de pasos:

1. Creación de una cuenta en SportsData y suscripción a los distintos deportes. Al subscribirse se recibe una clave API. Esta clave es una forma de autenticación y autorización que permite a los usuarios acceder a los datos de la API de Sportsdata.io de manera segura y controlada.
2. Desarrollo y despliegue del *smart contract* que permita la realización de apuestas por parte de distintos usuarios.
3. Creación y configuración de un nodo Chainlink utilizando Docker en una máquina virtual en la nube de Azure.
4. Configuración de dos servidores intermedios en la misma máquina virtual, los cuales serán encargados de realizar las peticiones a las APIs de SportsDataIO correspondientes. Cada servidor tiene como principal fin almacenar la clave de la API y así evitar que el contrato la tenga que enviar en cada llamada al oráculo, ya que de esta manera quedaría almacenada en la cadena de bloques de manera permanente y podría ser visible por todo el mundo.

5. Crear y configurar unos Jobs en Chainlink que se conecten al servidor correspondiente que está corriendo en la misma máquina.

Una vez se ha creado la cuenta en Sportsdata.io y se cuenta con la clave API, se procede a desarrollar el contrato inteligente.

Capítulo 3

3. Desarrollo Smart Contract

3.1 Introducción Solidity

Ahora que se han introducido todos los conceptos teóricos sobre Blockchain, y más concretamente la red de Ethereum, se procederá a la creación del contrato inteligente que se desplegará en una red de prueba de Ethereum llamada Sepolia. Como bien se ha comentado, el contrato será el encargado de realizar una apuesta, de manera que los pagos y el dinero será gestionado por él mismo.

Sin embargo, para poder desarrollar el contrato, ha sido necesario aprender el lenguaje con el que se escriben. Dicho lenguaje es *Solidity*.

Un contrato escrito en *Solidity* debe estar compuesta de lo siguiente:

- **Versión pragma** → Declaración de la versión de compilador de *Solidity* que debe usar el código. Evita problemas con futuras versiones del compilador que podrían introducir cambios que romperían el código. Dado que en este proyecto se va a utilizar Remix para compilar y desplegar el contrato, hay que asegurarse que el compilador y la versión pragma del contrato coinciden.
- **Bloque de tipo *contract*** → Es el bloque de construcción fundamental de las aplicaciones de Ethereum. Todas las variables y funciones definidas deben encontrarse dentro de dicho bloque.

Solidity es un lenguaje estáticamente tipado. Lo que significa que hay que conocer el tipo de nuestras variables en tiempo de compilación, en contraposición a lenguajes dinámicamente tipados que usan algún tipo de inferencia para deducir el tipo de las variables (por ejemplo, JavaScript). Esto implica que el programador debe indicar el tipo de las variables en los contratos al escribir el código.

Dentro de los diferentes tipos de datos en *Solidity*, podemos agruparlos en dos grupos, tipos por valor y tipos por referencia.

- **Tipos por valor (o básicos)**: son pasados por valor y son copiados cuando son usados como argumentos en funciones y asignaciones. Dentro de esta categoría se tienen **[16]**:
    - Valores lógicos (Boolean)
    - Números Enteros (Integer):
        - Defininidos como:
            - Int → con signo

        - Uint → sin signo
        - Se pueden definir numéricos de entre 8 y 256 bits (**int8** – **int256** y **uint8** – **uint256**). Debido que el almacenamiento de Ethereum es caro (por el gas) siempre se debería usar variables de menor tamaño cuando sea posible.
    - Números reales en punto flotantes (Fixed Point)
    - Direcciones (Address)
    - Arrays de bytes de tamaño fijo (bytesN)
- **Tipos por referencia (o complejos):** son tipos complejos y que en ocasiones puede no ser suficiente con los 256 bits de nuestra máquina virtual. Deben ser tratados con más cuidado que los tipos básicos ya que se podría incurrir en un uso de gas excesivo. En esta categoría se encuentran los siguientes tipos:
    - Arrays
    - Estructuras
    - Mappings: almacenamiento clave-valor para la consulta de datos. La sintaxis para almacenar datos en mappings es la misma que para arrays.

  Los *Strings* en Solidity son arrays dinámicos. Éstos son un tipo de referencia de tipo de datos que almacena la ubicación de los datos en lugar de almacenar directamente los datos en la variable.

En los contratos también aparecen los eventos. Éstos son una forma de registrar y emitir información relevante en la cadena de bloques. Estos eventos pueden ser escuchados por otros contratos o aplicaciones externas. Los eventos son utilizados para notificar a los interesados acerca de la ejecución de una transacción o la ocurrencia de un evento en el contrato.

A la hora de crear el contrato inteligente hay que tener claro que la cadena de bloques de Ethereum está compuesta por cuentas. Estas cuentas tienen un balance de Ether (moneda utilizada en la cadena de bloques de Ethereum), y se pueden enviar dichos Ethers entre distintas cuentas, como si se tratara de una transferencia normal y corriente. Ahora bien, cada cuenta tiene una dirección, que sería el equivalente al número de cuenta de una cuenta bancaria tradicional. Se trata de un identificador único que se asemeja al siguiente:

0x0cE446255506E92DF41614C46F1d6df9Cc969184

Nuevamente, hay que destacar también los propios contratos también tienen asociada una dirección. De manera que cuando un jugador quiera realizar una apuesta deberá enviar la cantidad que quiere apostar más la cantidad requerida de gas al contrato.

En Solidity, hay ciertas variables globales que están disponibles para todas las funciones. Una de ellas es la variable *msg.sender*, que devuelve la dirección de la persona (o *smart contract*) que ha llamado a la función actual (todas las funciones han de ser llamadas por un agente externo). Usar *msg.sender* da la seguridad en la blockchain de Ethereum. La única manera de que alguien pueda modificar los datos de otra persona sería robando la clave privada asociada a su dirección de Ethereum **[15]**.

En *Solidity*, las variables pueden ser almacenadas en dos sitios: almacenamiento y memoria.

El almacenamiento se refiere a las variables almacenadas permanentemente en la cadena de bloques. Las variables de memoria son temporales, y se borran entre llamadas a funciones externas a tu contrato. Para definir una variable en memoria basta con introducir el identificador “memory” después del tipo en la declaración de una variable. De esta manera, se pueden definir variables temporales dentro de una función sin necesidad de escribir dicha variable en la cadena de bloques, lo que provoca una reducción del gas requerido. Una peculiaridad de las variables de memoria es que su tamaño no puede ser modificado.

El hecho de que los contratos sean inmutables una vez pasan a formar parte de la blockchain hace que se tenga que tener en cuenta el aspecto de la seguridad desde el principio, ya que ésta no se va a poder añadir más adelante. En caso de que un contrato cuente con defectos, se tendrá que avisar a los usuarios de alguna manera para que no utilicen dicho contrato, y se les entregará la dirección del nuevo contrato desplegado que sí que cumple con las especificaciones de seguridad. Sin embargo, puede tener sentido utilizar funciones que permitan actualizar partes clave de la DApp.

Como se ha mencionado anteriormente, los usuarios tienen que pagar cada vez que ejecutan una función en una DApp. Para realizar dicho pago se utiliza gas. El gas puede ser comprado con la moneda nativa de Ethereum (Ether). De esta manera, los usuarios han de gastar Ethers para poder ejecutar las funciones en una DApp. La cantidad de gas necesaria para ejecutar una función viene determinada por la complejidad lógica de la misma. Es decir, dependiendo de cuantos recursos se necesitan para realizar las operaciones que se encuentran en la función, el gas requerido variará. Por todo la anterior, en *Solidity* es fundamental la optimización de código para que se realicen el menor número de operaciones necesarias.

Ahora vamos a hablar de las funciones de tipo *Payable* de *Solidity*. Éstas permiten que el contrato pueda recibir *Ethers*, lo que proporciona una funcionalidad realmente interesante, como exigir un determinado pago al contrato para ejecutar una función. Dado que el trabajo tratará sobre la realización de apuestas, este tipo de funciones serán utilizadas para enviar el dinero correspondiente de la apuesta que se quiere realizar, así como realizar el pago de las ganancias a los usuarios que hayan acertado su apuesta.

¿Pero que pasa exactamente cuándo se envía dinero a un contrato? Un dinero se almacena en la cuenta Ethereum del contrato, y se requiere una función que se encargue de retirar ese dinero para que sacar el dinero del contrato. Cabe destacar que no se puede transferir Ether a una dirección a no ser que dicha dirección sea de tipo *address payable*.

Remix es un IDE (Integrated Development Environment) web para Ethereum que permite escribir, compilar, desplegar y probar contratos inteligentes. Es una herramienta muy útil para desarrolladores que quieren trabajar con contratos inteligentes en la plataforma Ethereum, ya que ofrece una interfaz gráfica de usuario para interactuar con los contratos, y proporciona un entorno de prueba integrado para depurar y validar el código del contrato antes de desplegarlo en la red principal.

Ahora, se puede desarrollar el contrato teniendo en cuenta todas las consideraciones a cerca de los oráculos y la sintaxis de *Solidity*. Como herramienta para desarrollar el contrato de utilizará Remix.

3.2 Explicación código smart contract

El código del contrato se ha incluido en el anexo del documento. Sin embargo, se procede a explicar su contenido para facilitar la comprensión al lector a la hora de encontrarse con él sin tener conocimientos a cerca del lenguaje Solidity. Se recuerda que dicho contrato se utiliza para que interactúen los usuarios que deseen realizar una apuesta de manera descentralizada utilizando la tecnología Blockchain. Tal y como se ha desarrollado el contrato, los usuarios solo podrán realizar apuestas de fútbol a partidos de la liga española y a ganadores de torneos de golf.

En primer lugar, se importan los siguientes contratos:

- **ChainlinkClient.sol**: Este contrato es importado de la biblioteca Chainlink. Permite al contrato que se ha desarrollado para este proyecto interactuar de manera eficiente y segura con los servicios de oráculo de Chainlink. Alguna de las funciones clave de este contrato pueden ser la creación y envío de solicitudes de datos a los oráculos de Chainlink. Muchas de estas funciones se describirán próximamente cuando se explique el código del contrato.
- **Ownable.sol**: Este contrato es importado de la biblioteca OpenZeppelin, la cual es utilizada principalmente para proporcionar seguridad a los contratos. *Ownable* es un contrato que implementa un modelo de propiedad donde solo el propietario del contrato puede realizar ciertas funciones. A destacar, el modificador onlyOwner(), el cual comprueba que el usuario que llama a la función del contrato es el dueño del mismo.
- **SafeMath.sol**: Este contrato también es importado de la biblioteca OpenZeppelin. SafeMath permite realizar operaciones matemáticas de números enteros sin signo de forma segura, previniendo desbordamientos y subdesbordamientos que podrían llevar a resultados incorrectos o vulnerabilidades de seguridad. En el contexto del contrato desarrollado en este trabajo, se utilizará para comprobar que al incrementar el valor de una variable de tipo uint256 no se producen *overflows*.
- **Strings.sol**: Este contrato nuevamente es importado de la biblioteca OpenZeppelin. Strings permite manipular y trabajar con cadenas de caracteres en Solidity. En el contrato desarrollado se utilizará para convertir un valor numérico a cadena de caracteres.

Posteriormente, se define el contrato llamado *SportsDataOracle* que heredará del contrato *ChainlinkClient* y *Ownable* que se han importado anteriormente. Esto significa que el contrato utilizará las funciones y características proporcionadas por estos contratos base para interactuar con los oráculos de Chainlink y establecer propiedades de propiedad y control de acceso. También se están utilizando las bibliotecas *SafeMath* y *Chainlink* a través de las directivas *using* para proporcionar funciones adicionales al contrato.

Ya dentro del cuerpo del contrato se definen una serie de variables de estado:

- balances: Ss una variable de tipo “mapping”, los cuales no son más que una colección de pares clave-valor donde una clave se asigna a un valor. En este caso se utilizan direcciones de Ethereum como claves Y números enteros sin signo (uint) como valores. Esta variable se utiliza entonces para almacenar los saldos de ETH asociados a direcciones de usuarios (*wallets*).

- *Bet_soccer* y *Bet_golf:* Son las variables más importantes. Son unas estructuras que contienen cada uno los atributos necesarios para realizar una apuesta. La primera contiene distintos atributos como *player, amount, winner, jornada, equipo_local, equipo_visitante,* y *completed*, y hace referencia a todas las apuestas de fútbol que se van a realizar. Por otro lado, la estructura Bet_golf engloba los atributos *player, amount, winner, id_torneo,* y *completed.* Cada apuesta que realice un usuario contendrá los atributos anteriormente mencionados según el deporte al que apueste a excepción del atributo *completed*. Cabe destacar que el atributo *completed* no lo debe inicializar el usuario, ya que el contrato se encargará de iniciarlo a "False" (al tratarse de una variable de tipo bool). Esto se debe a que este atributo indica que la apuesta se ha completado. Es por ello por lo que, al crear una apuesta se debe indicar que no se ha completado porque aún no se sabrá el resultado de la misma.
- *bets_soccer* y *bets_golf:* Estas variables son arrays. La primera es un array de estructuras *Bet_soccer* y la segunda de *Bet_golf*.
- betToOwner: Otra variable de tipo "mapping" que relaciona una apuesta con la *wallet* del usuario que ha realizado la apuesta.
- ownerBetsCount: Nuevamente una variable de tipo "mapping" que con relaciona el número de apuestas que ha realizado un usuario. Se entiende a cada usuario como una dirección de Ethereum.
- oracle: Dirección de Ethereum del contrato operador.
- *jobId_resultados_soccer* y *jobId_resultados_golf*: Variables de tipo *Bytes32*. La primera sirve para almacenar el *jobID* del Job que será el encargado de obtener los resultados de fútbol, mientras que la segunda variable indica el jobID del Job que hará lo propio con los resultados de los torneos de golf. Cada Job incluye cierta lógica que tiene como objetivo recuperar unos datos en concreto. Sin embargo, las funciones de cada uno de los Jobs se describen más adelante.
- fee: Variable que contiene la tarifa en tokens LINK que debe pagar el contrato al nodo Chainlink para que ejecute los Jobs correspondientes especificados en las llamadas. Este valor será inicializado en el constructor.
- OracleResponse: Se trata de un evento. Los eventos se suelen utilizar para que los usuarios y otros contratos escuchen y reaccionen a eventos que ocurren en un contrato. Este evento se utiliza para registrar y notificar las respuestas obtenidas de los oráculos de Chainlink.

Posteriormente, el contrato cuenta con un conjunto de funciones. En primer lugar, se encuentra el constructor. Al igual que en muchos lenguajes de programación, el constructor es una función especial que se ejecuta en el momento de despliegue del contrato. Este constructor se utiliza para inicializar valores de determinadas variables como pueden ser la tarifa que debe pagar cada usuario que desea enviar una solicitud al oráculo, la dirección del contrato oráculo (Operator.sol) que se ha tenido que desplegar y que es necesario para gestionar la operación de los nodos, y la cuenta Ethereum Sepolia a la que se realizan pagos LINK para compensar al nodo por sus servicios.

En cuanto a la función *placeBet()*, se trata de una función que el usuario llamará cuando desee realizar una apuesta, tanto de fútbol como de golf. Dicha función se ha definido como publica debido a que se espera que dicha función sea llamada por todo el mundo que quiera realizar una apuesta, por lo que se quiere que sea accesible desde cualquier lugar. Además, también se

ha definido como tipo *payable*. Esto provoca que la función pueda recibir cierta cantidad de Ether junto con la llamada a la función. Es decir, los usuarios que interactúan con el contrato pueden enviar Ether al llamar a esa función. Sin embargo, al marcar una función como *payable*, también se está permitiendo que el contrato pueda recibir Ether a través de otras operaciones, como el envío directo de Ether a la dirección del contrato. En este caso, interesa que la función esté definida como payable ya que el usuario deberá enviar una cantidad de ETH al contrato para poder realizar una apuesta.

El valor de Ether enviado en la propia llamada a la función estará disponible dentro de la función a través de la variable *msg.value.*

El usuario que llame a esta función deberá pasar como argumento todos los atributos necesarios para realizar una apuesta, además de indicar el deporte para el cual está realizando una apuesta. En caso de que desee realizar una apuesta de futbol, se verán pasar como argumento todos los atributos de la estructura *Bet_soccer* (salvo el atributo *completed* por las razones especificadas anteriormente). En la definición de la función nótese como hay varias variables definidas con el modificador *memory.* Éste se utiliza para indicar que aquella variable debe almacenarse en la memoria temporal durante la ejecución de la función. Es decir, su valor no persistirá en el estado del contrato una vez que la función haya finalizado su ejecución. Esto tiene la gran ventaja de que la ejecución de la función puede ser más eficiente en términos de consumo de gas y costo computacional.

Se puede observar cómo al principio de la función se declara la variable *player* y se le asigna el valor de la dirección del usuario que ha llamado a dicha función (*msg*.*sender*). Posteriormente, se realizan numerosas comprobaciones utilizando la función *require()*. Además, todas las comprobaciones conviene realizarlas al principio de la función antes de cambiar el estado de alguna de las variables, lo que supone gasto de gas. Al colocar las validaciones al principio de la función, se tiene un mayor control sobre el flujo de ejecución. Esto permite que las condiciones requeridas se verifiquen antes de realizar cualquier acción o cálculo en el contrato, evitando posibles problemas de seguridad o resultados no deseados. Un fallo común que sucede en estos casos son los llamados *reentrancy attacks*. Esta vulnerabilidad en los contratos permite a un actor malicioso llama a una función de un contrato que interactúa con otro contrato y luego llama inmediatamente a la misma función de nuevo antes de que la primera llamada a la función se complete, lo que puede provocar una retirada de dinero en bucle del contrato. *Require()* es una función de *Solidity* que se utiliza para verificar ciertas condiciones antes de ejecutar el resto de la función.

Las comprobaciones que se realizan son:

1. Comprobar que la cantidad apostada es mayor que cero.
2. Comprobar que el usuario que realiza la apuesta cuenta con más ETH de los que está apostando.
3. Comprobar que el valor enviado de ETH enviado al contrato en la llamada a la función y el valor que el usuario desea apostar (especificado como argumento “amount” en la llamada a la función) coinciden.

Entre alguna de las comprobaciones se ha incluido un nuevo elemento al “mapping” balances. Se ha añadido la dirección del usuario que llama a la función como clave y su balance de ETH como valor. Asimismo, después de las validaciones se reduce el balance en el valore de la clave anterior dentro de la variable balances.

Después, se comprueba si la apuesta que se desea realizar es de fútbol o golf mediante la comprobación de uno de los parámetros que se deben pasar a la función (aquel que indica el deporte). En función del tipo de apuesta que sea se crea una variable de tipo estructura (*Bet_soccer* o *Bet_golf*) inicializando los atributos de la estructura correspondiente con los valores que se han pasado a la función en la llamada. A continuación, se añade dicha estructura al array de estructuras *bets_soccer* o b*ets_golf* en función del tipo de apuesta que sea. Posteriormente, se inicializa una variable llamada *betId*, la cual tendrá el valor correspondiente a la longitud del array de estructuras *bets_soccer* o b*ets_golf.* Mas tarde, se añade al "mapping" *ownerBetsCount* un registro en el que la clave es la dirección del usuario que ha llamado a la función y el valor se incrementa en uno utilizando la función segura *add()* del contrato SafeMath.sol.

Finalmente, se llama a la función *requestOracleDataResultado()* en la que se le pasa como argumentos el deporte al que se está apostando y el ID de la apuesta definido anteriormente (*betId*).

En cuanto a la función *requestOracleDataResultado()*, ésta se llamará una vez la apuesta se haya dado de alta en el array de apuestas definido al principio. En este caso, la función puede ser declarada como privada, ya que ésta solo será llamada por la función creada anteriormente. Esta función se encargará de realizar la solicitud al nodo Chainlink, y se le pasarán como argumentos el *betId* (id de apuesta), y el deporte al que se apuesta.

Lo primero que hace esta función es comprobar si se trata de una apuesta a un partido de fútbol o a un torneo de golf. Para realizar dicha comprobación se utiliza el primer argumento pasado a la función que indica el deporte al que se está apostando. Posteriormente, independientemente del deporte al que se esté apostando hay que crear una solicitud al oráculo utilizando la función *buildChainlinkRequest()* del contrato ChainlinkClient.sol. En dicha solicitud se incluyen ciertos parámetros como *jobId_resultados_soccer* (en el caso de que se trate de una apuesta de fútbol), que representa el identificador único del trabajo específico que se debe ejecutar en nodo Chainlink), *address(this)* (índica la dirección del contrato actual ya que es donde se espera que el nodo Chainlink devuelva la respuesta), y *this.fulfillOracleRequest.selector* (selector de la función de *callback* en el contrato actual que se invocará cuando el nodo Chainlink complete la solicitud y proporcione una respuesta).

A continuación, se añaden los parámetros de la solicitud utilizando el método *add* de la solicitud. Estos parámetros incluyen información específica sobre la apuesta. En caso de que se trate de una apuesta a un partido de fútbol, se incluyen parámetros como el deporte, el identificador de apuesta (*betId*), el equipo local, el equipo visitante y la jornada. En el caso de que se trate de una apuesta de golf, los parámetros que se incluyen en la solicitud al oráculo son el deporte, el id de la apuesta y el torneo al que se está apostando. Esos datos que se pasan en la solicitud al oráculo son los justos y necesarios que necesita el oráculo para obtener el ganador del evento.

Finalmente, en ambos casos se llama a la función *sendChainlinkRequestTo()* para enviar la solicitud al oráculo especificado por la dirección *oracle*. También se incluye el parámetro *fee*, que representa la cantidad de tokens de pago que se enviarán al contrato Oracle como compensación por el servicio proporcionado.

De esta manera, el oráculo cogerá la información de la estructura recibida y la pasará al servidor intermedio correspondiente en función del tipo de apuesta que se quiera realizar. El

cual la utilizará para poder recuperar el resultado de dicho partido utilizando la API de *SportsData*.

Como bien se ha comentado anteriormente, se cuenta con una función llamada *fulfillOracleRequest()* que se ejecutará cuando el nodo Chainlink le envíe la respuesta al contrato. El nodo Chainlink devolverá tanto el resultado del evento deportivo al que el usuario ha apostado, el *betId* mandado previamente por el propio contrato y el deporte al que le corresponde ese resultado. A continuación, se realiza nuevamente una comprobación del deporte se está tratando. En el caso de que se trate de una apuesta de fútbol, se crea una variable llamada apuesta que representa la apuesta de fútbol que realizó el usuario. Para ello se utiliza la variable *betId* recibida como argumento en la función. Asimismo, se pone modifica el atributo "completed" de esa apuesta, el cual indica que la apuesta ha sido completada. Posteriormente, se realiza una comprobación de que el ganador del partido devuelto por el nodo Chainlink es igual al atributo "winner" de la apuesta realizada. En caso de que se cumpla esta condición se llama a la función *calcular_cuotas()* que será la encargada de realizar el cálculo de las cuotas para estimar posteriormente la cantidad que hay que pagar al usuario. Por otro lado, en caso de que la condición anterior no se cumpla, es decir, de que el ganador del partido no concuerde con lo introducido en el parámetro "winner" al crear la apuesta terminará la ejecución del contrato.

Para el caso de que se trate de una apuesta de golf la lógica seguida es equivalente, salvo que esta vez se comprueba que el ganador del torneo devuelto por el oráculo concuerde con el atributo "winner" de la apuesta creada.

La función *calcular_coutas()* nuevamente ha sido definida como privada, ya que únicamente debería ser llamada por una función dentro del contrato. Esta función procederá a comprobar nuevamente si se trata de una apuesta de fútbol o de golf. En el caso de que sea una apuesta de fútbol, se realiza el cálculo de las cuotas en función de la cantidad de apostantes que hay a cada equipo de dicho evento. Es decir, primero se cuentan las apuestas que se han realizado al mismo partido (el valor se almacena en la variable "apuestas_al_partido" definida al principio de la función). Para ello, se comprueba que los partidos tienen el atributo "jornada" igual y luego se comprueba que tanto el equipo local como el equipo visitante coinciden. Mas tarde, se comprueba cuantas de estas apuestas han elegido al mismo equipo que la apuesta ganadora como vencedor, y dicho valor se almacena en la variable *mismas_apuestas*.

Finalmente, para calcular la cuota se divide la variable *apuestas_al_partido* entre *mismas_apuestas*. Una vez calculada la cuota se calcula la cantidad que hay que pagar al usuario ganador. Para ello, se multiplica la cuota calculada por la cantidad apostada del jugador (accesible a través del atributo "amount" de la estructura que representa la apuesta.

Por otro lado, en el caso de que se trate de una apuesta de golf, para calcular la cuota se comprueban cuantas apuestas se han realizado al mismo torneo (valor almacenado en la variable *apuestas_al_torneo*) y cuantas de ellas son también al mismo ganador (valor almacenado en la variable *misma_apuesta)*. El cálculo de la cuota y de la cantidad que se debe enviar al ganador es igual que en el caso de fútbol.

Finalmente, en ambos casos se llamará a la función *realizar_pago()* que se encargará de realizar el pago al usuario que se ha determinado como ganador.

La función *realizar_pago()* recibe como argumentos la dirección de la cartera de ETH del ganador y la cantidad que se le debe abonar. Dicha función hace uso de la función *transfer()*

para el envío del dinero (tokens Ether). Sin embargo, primero se comprueba que el balance del contrato es superior a la cantidad de dinero que debe enviar al jugador. En caso de que esta condición no se cumpla no se podrá proceder al envío de los tokens ETH correspondientes.

El contrato también cuenta con dos funciones *setters.* Una de ellas se encarga de definir los *jobIDs* a los que realizará la solicitud el *smart contract*. Cabe destacar que estas funciones reciben los *jobIDs* de los Jobs que se van a encargar de recuperar los datos de los eventos deportivos en formato *String* y esta función se encarga de convertirlos a tipo *Bytes32* porque es el tipo que soporta la función *buildChainlinkRequest()* que se ha comentado anteriormente. A continuación se muestra la función *buildChainlinkRequest()* del contrato ChainlinkClient:

```
function buildChainlinkRequest(
  bytes32 specId,
  address callbackAddr,
  bytes4 callbackFunctionSignature
) internal pure returns (Chainlink.Request memory) {
  Chainlink.Request memory req;
  return req.initialize(specId, callbackAddr, callbackFunctionSignature);
}
```

Ilustración 11: Función buildChainlinkRequest()

Por otro lado, la otra función *setter* se encargará de cambiar la tarifa que cobrará el nodo Chainlink por utilizar sus servicios en caso de que en futuro se quiera cambiar el valor. Ambas funciones cuentan con el modificador *onlyOwner* que heredan del contrato *Ownable.sol*. Dicho modificador hace que la función solo pueda ser ejecutada cuando quien la llama es el dueño del contrato.

Además de las funciones *setters* recién mencionadas, el contrato cuenta con cuatro funciones *getters*. Dichas funciones se utilizan para recuperar los valores de las variables *oracle*, *fee*, *jobId_resultados_golf* y *jobId_resultados_soccer*. Estas funciones son necesarias para obtener los valores de las variables ya que se han definido como privadas.

Finalmente, el contrato cuenta con la función mencionada anteriormente llamada *stringToBytes32()*, la cual se encarga de convertir una variable *String* a *Bytes32*.

3.3 Pruebas unitarias smart contract

Una vez se tiene el contrato preparado, se ha optado por realizar una serie de tests unitarios sobre el mismo para comprobar que se comporta como se espera. Precisamente para ello, se ha utilizado la herramienta **Truffle**.

Truffle es un framework Opensource el cual entre muchas otras funciones permite probar el funcionamiento del código escrito. Actualmente soporta tanto código JavaScript como Solidity. En este caso, se va a utilizar para probar que contrato inteligente hace lo que se espera que haga. Algunas de las características más interesantes de Truffle se detallan a continuación:

- Truffle proporciona un entorno de prueba preconfigurado y listo para usar. Este entorno está diseñado para simular una cadena de bloques local y permite a los desarrolladores ejecutar y depurar sus contratos inteligentes en un entorno controlado antes de implementarlos en una red de producción.

- Los scripts de prueba se pueden personalizar para adaptarse a las necesidades específicas del contrato. Puedes definir tus propias pruebas y asertos, lo que te brinda un alto grado de flexibilidad para verificar diferentes aspectos del contrato.
- Truffle simplifica el proceso de despliegue de contratos inteligentes para pruebas. Se pueden definir diferentes redes (como desarrollo, prueba y producción) en el archivo ***truffle-config.js*** y utilizar dichas configuraciones para implementar automáticamente los contratos en la red de pruebas
- Truffle habilita la utilización de cuentas de prueba financiadas con Ether en el entorno de prueba de manera que los tests que requieran de transacciones y/o saldos de cuenta se puedan implementar.
- El framework registra las transacciones que se llevan a cabo, lo cual facilita la trazabilidad de las acciones realizadas.
- Se permite la inclusión de bibliotecas.

Los pasos que se han llevado en los que la herramienta Truffle se ha visto involucrada son:

1. **Creación de un proyecto Truffle**. Una vez se ha instalado Truffle en el sistema, hay que crear un proyecto Truffle. Para ello, hay que dirigirse con el terminal a una ubicación en la que se desee que se ubique el proyecto y ejecutar el comando "***truffle init***". Al ejecutar el comando, se crea una estructura de directorios y archivos.
   En primer lugar, se crea un directorio llamado "test" donde se incluyen los archivos que contienen los tests unitarios. Dichos archivos son documentos JS, cuyo nombre debe acabar con el sufijo ".test.js".
   En segundo lugar, se crea un directorio llamado "migrations". En este directorio se almacenan los archivos de migración, los cuales definen como se desplegarán los contratos en las diferentes redes.
   Adicionalmente, se crea también una carpeta llamada "contracts" en la que se deben ubicar los contratos sobre los cuales se van a realizar pruebas.
   Finalmente, en la ubicación donde se ha ejecutado anteriormente el comando "***truffle init***" se han creado dos archivos:
   - *truffle-config.js*: Contiene información sobre la configuración de Truffle en el proyecto.
   - *package.json*: Archivo de configuración para las dependencias de Node.js.
2. **Creación de tests**. Tras la iniciación del proyecto Truffle. Se procede a crear el archivo que contendrá los pruebas que se quieren realizar para comprobar que la lógica del contrato inteligente es la correcta.
   En este proyecto se han creado distintos tests, pero todos ellos se encuentran en el mismo archivo. En primer lugar, se han creado dos tests tenían como motivo la comprobación de que las estructuras correspondientes a las apuestas de fútbol y golf se creaban correctamente. En segundo lugar, se ha creado un test que se encarga de comprobar que las variables inicializadas en el constructor obtienen el valor esperado. Y finalmente, se han creado tests para comprobar que las funciones que se encargan de cambiar el valor del id del Job de Chainlink que es el encargado de obtener los resultados tanto de golf como de fútbol funcionan correctamente.

3. **Ejecución de tests**. El paso final es correr los tests hallados en el archivo con sufijo ".test.js". Para ello se puede ejecutar el comando "**truffle test**". Este comando ejecuta todas las pruebas que se hayan definido en los archivos de la carpeta "test"

(mencionada anteriormente). Para la ejecución de las pruebas es necesario la compilación de los contratos, por lo que si el contrato cuenta con errores se mostrarán por pantalla.

Cabe mencionar que se ha tenido que modificar ligeramente el contrato original para la realización de las pruebas. Esto se debe a que truffle no entiende de los contratos previamente desplegados en la red Ethereum, de manera que también hay que incluir los contratos que son importados en nuestro contrato en el proyecto truffle en la carpeta "contracts".

Como resultado, se compilarán todos los contratos para que las pruebas se puedan llevar a cabo.

Asimismo, la ejecución del comando mostrará los resultados de las pruebas e informará si todas las afirmaciones pasan (o no).

El código del documento que se ha escrito que contiene los tests unitarios para probar el funcionamiento del contrato desarrollado en este proyecto se incluye también al final del documento en el Anexo. Sin embargo, se procede a comentar un poco su contenido.

En primer lugar, el archivo de test verifica el funcionamiento del contrato inteligente llamado "SportsDataOracle". En la primera línea se importa la línea utilizando la función *artifacts.require()* proporcionada por Truffle. Esto permite interactuar con el contrato en las pruebas.

La línea "contract('MyContract', (accounts) => {" indica la definición de las pruebas. Posteriormente, se inicializa una instancia del contrato inteligente previa a la ejecución de las pruebas. La función *before()* se ejecuta antes de cualquier prueba. Y en este caso se utiliza para crear la nueva instancia del contrato. Finalmente, se encuentran las pruebas unitarias como tal. Actualmente hay cinco pruebas, se procede a comentar cada una de ellas de manera detallada:

1. Prueba para comprobar que los valores de las variables inicializadas en el constructor contienen el valor deseado. El test consiste en definir una serie de variables que se obtienen como resultado de la ejecución de las funciones *getOracle()* y *getFee()*, las cuales devuelven los valores actuales de las variables *oracle* y *fee* respectivamente. Luego se hace una comprobación para ver si los valores de las variables coinciden.
2. Prueba para comprobar que la estructura (apuesta) de fútbol se ha creado correctamente. En esta caso el test consiste en llamar a la función *placeBet()* del contrato y se le pasan los argumentos necesarios para la creación de la estructura que contiene la apuesta. El último paso consiste en verificar si los valores almacenados en la variable *bet* coinciden con lo que se espera utilizando afirmaciones *assert.equal()*.
3. Prueba para comprobar que la estructura (apuesta) de golf se ha creado correctamente. Igual que la prueba anterior, pero con golf.
4. Prueba para comprobar el funcionamiento de la función *setter* y *getter* de la variable *jobId_resultados_golf*. El test consiste en llamar a la función setter (*setJobId_resultados_golf*) pasándole como argumento un valor determinado. Luego se llama a la función getter (*getJobId_resultados_golf*) y se almacena su valor en una variable. Finalmente se comparan ambos valores, si son iguales la prueba pasa.
5. Prueba para comprobar el funcionamiento de la función *setter* y *getter* de la variable *jobId_resultados_soccer*. Igual que el test anterior, pero con fútbol en vez de con golf.

En la salida del comando “truffle test” se puede comprobar como los tests se han pasado exitosamente:

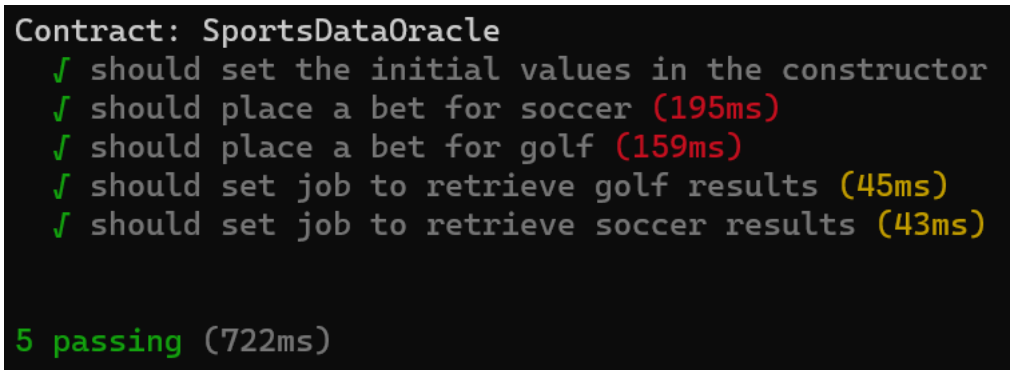

Ilustración 12: Salida comando "truffle test"

3.4 Despliegue smart contract

Como se ha comentado anteriormente, para poder desplegar el contrato, este primero debe estar en Bytecode. Remix nos permite tanto compilar el contrato (convertirlo a Bytecode), como desplegarlo.

Los pasos a seguir para desplegar el contrato son los siguientes:

1. **Acceder a Remix**. La herramienta Remix es accesible a través de https://remix.ethereum.org/
2. **Cargar archivo Solidity**. Hay que cargar el archivo “.sol” que contiene el contrato. Una vez cargado, la interfaz se ve así:

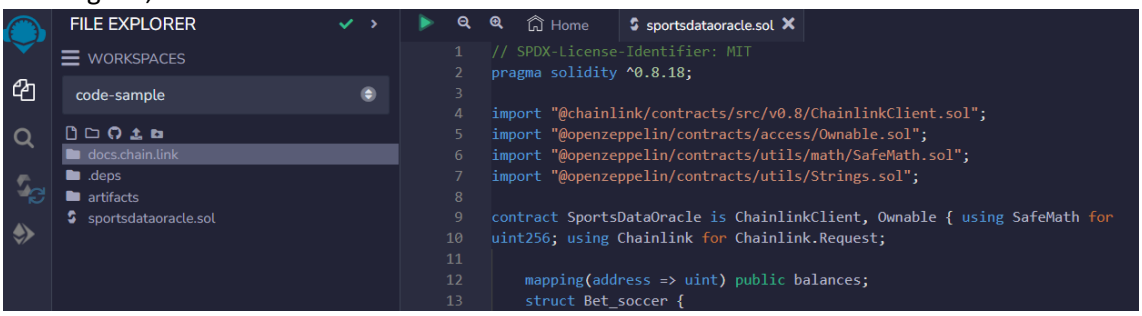

Ilustración 13: Contrato cargado en Remix

3. **Compilar el contrato**. Para compilar el contrato, basta con hacer clic en el triángulo verde situado en la parte superior de la pantalla.
4. **Desplegar contrato**. Una vez se tiene el contrato compilado, se puede proceder a su despliegue. Para ello, hay que irse a la última de las ventanas del menú lateral ubicado a la izquierda. Es ahí cuando hay que modificar algunas opciones para desplegar el contrato de manera exitosa.

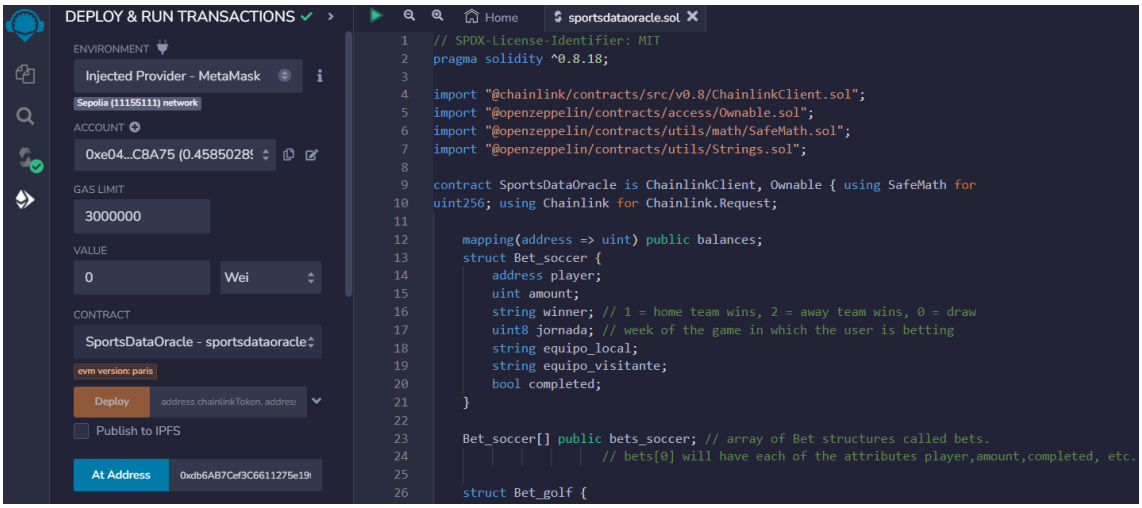

Ilustración 14: Ventana de despliegue Remix

En primer lugar, donde pone la opción “Environment” (primera de las opciones) hay que seleccionar MetaMask, ya que se desea que utilizar la *wallet* de MetaMask para el despliegue. Posteriormente, hay que indicar el límite de Gas a utilizar para el despliegue del mismo. Asimismo, en la opción “Value” se puede indicar una cantidad de ETH o de Wei para enviar al contrato en el momento del despliegue. Para el caso de uso de este proyecto no se requiere que el contrato cuente con ETH desde el principio. Finalmente, en la opción “Deploy” hay que indicar el valor de los argumentos que hay que pasar al constructor que se ha comentado anteriormente. Una vez se hayan introducido hay que hacer clic en “Deploy”.

De esta manera, ya se tendía el contrato desplegado y se podría interactuar con él. Sin embargo, como se ha comentado anteriormente, el contrato debe pagar en tokens LINK al oráculo Chainlink para que éste lleve a cabo la recopilación de datos que el contrato le ordena. Entonces, es necesario enviar al contrato ciertos LINKs.

Cabe mencionar que otro contrato debe haber sido desplegado para la realización de este proyecto. Este contrato es el Operator.sol. Este contrato se utiliza para gestionar la operación y administración de los nodos de Chainlink. El contrato Operator debe haber sido desplegado con anterioridad al contrato desarrollado en este proyecto, ya que su dirección se debe haber pasado como argumento a la hora de desplegarlo en el constructor.

El contrato Operator cuenta con un función llamada *SetAuthorizedSenders()* que hay que llamar pasando como argumento la dirección del nodo Chainlink que se ha desplegado. Esto permitirá que el nodo Chainlink pueda llamar al contrato Operator.

Capítulo 4

4. Nodo Chainlink y Servidor intermedio

4.1 Levantar nodo Chainlink

La máquina en la que se va a levantar un nodo Chainlink debe contar con un mínimo de dos núcleos de CPU y 4 GB de RAM para que el nodo se ejecute y se puedan realizar pruebas. En mi caso, el nodo Chainlink solo contará con la ejecución de dos *jobs*, por lo que una máquina con las características recién mencionadas debería ser suficiente **[31]**.

Además, se recomienda que la máquina que se elija cuente con un sistema operativo Linux, aunque se puede levantar un nodo con una imagen de Docker. Las características de la máquina virtual que se va a utilizar en la nube de Azure son las que se encuentran en la posterior imagen:

Tamaño de VM ↑↓	Tipo ↑↓	vCPU ↑↓	RAM (GiB) ↑↓	Discos de datos ↑↓	E/S máxima por s... ↑↓	Almacenamiento t... ↑↓	Disco prémium ↑↓	Costo/mes ↑↓
Más usados por los usuarios de Azure	Los tamaños más usados por los usuarios en Azure.							
B2s	Uso general	2	4	4	1280	8	Se admite	30,37 US$

Ilustración 15: Especificaciones máquina virtual

Una vez se tiene la máquina virtual corriendo, hay que conectarse a ella mediante SSH. Cuando se esté dentro de la máquina, lo primero que hay que hacer es instalar Docker para correr un nodo Chainlink localmente. Dicho nodo será configurado para conectarse a la red de Ethereum Sepolia. Sepolia es una red para pruebas de PoS para Ethereum, lo que lo convierte en un entorno adecuado para realizar todo tipo de pruebas antes de desplegar las DApps en la red principal. Sepolia es muy similar a Ethereum, y esto permite a los desarrolladores diseñar, crear, probar y monitorear el desempeño público de su proyecto antes de lanzarlo en la red principal Ethereum **[1]**.

El primer paso para levantar el nodo Chainlink es levantar una base de datos relacional PostgreSQL, para ello se puede emplear una imagen de *Docker*. El comando que hay que introducir para correr la base de datos PostgreSQL en un contenedor Docker es:

```
docker run --name cl-postgres -e POSTGRES_PASSWORD=mysecretpassword -p 5432:5432 -d postgres
```

mysecretpassword se debe reemplazar con la contraseña que se quiera utilizar.

Una vez, creada la base de datos PostgreSQL, se puede crear el nodo Chainlink. Sin embargo, para poder levantarlo, es necesario contar con un archivo *secrets.toml* y *config.toml*. El archivo *secrets.toml* se encarga de almacenar la clave de la base de datos PostgreSQL inicializada anteriormente y la URL a dicha base de datos. Por otro lado, el archivo *config.toml* cuenta con las variables de configuración del nodo. Estos ficheros de configuración serán incluidos en el anexo.

En el fichero de configuración *config.toml* se configuran una serie de parámetros. Cada uno de esos parámetros se encuentran en secciones:

- Sección “Log”:
    - Parámetro “Level”: Establece el nivel de registro de los mensajes del nodo. En mi caso, se ha configurado como “warn”, de manera que solo se registrarán logs cuyo nivel de registro sea de advertencia o superior.
- Sección “WebServer”:
    - Parámetro “AllowOrigins”: Define los dominios que deberían poder acceder a la interfaz web del nodo. En este caso se ha permitido el acceso a cualquier dominio (“*”).
    - Parámetro “SecureCookies”: Indica si las cookies deben ser seguras (HTTPS) o no (HTTP). En este caso está definido a “false”, por lo que no deben ser seguras.
- Sección “WebServer.TLS”:
    - Parámetro “HTTPSPort”: Define el puerto para el servidor HTTPS. Dado que en este caso no se está empleando HTTPS, el puerto definido es 0.
- Sección ”EVM”:
    - Parámetro ChainID: Especifica el ID de la cadena de bloques Ethereum con la que el nodo se conectará. En este caso, el ID de la cadena es '11155111'.
- Sección “EVM.Nodes”:
    - Parámetro “Name”: Especifica el nombre que se le debe asignar al nodo Chainlink en la red de Ethereum.
    - Parámetro “WSURL”: La URL WebSocket a la que se conectará el nodo para interactuar con la red Ethereum.
    - Parámetro “HTTPURL”: La URL HTTP a la que el nodo se conectará para realizar llamadas a la red Ethereum.

Como se ha podido comprobar, hay que incluir un par de URLs que se obtienen tras crear un proyecto en una cuenta de un proveedor externo Ethereum. En este proyecto se ha creado una cuenta en Infura. Infura permite a los desarrolladores y aplicaciones interactuar con la red Ethereum sin necesidad de configurar y mantener su propio nodo.

En cuanto al fichero *secrets.toml,* este también cuenta con una serie de secciones. Dentro de las secciones se encuentran unos parámetros:

- Sección “Password”:
    - Parámetro “KeyStore”: Aquí se proporciona una contraseña para el *keystore*. Ésta es una parte importante de la seguridad del nodo, ya que almacena las claves privadas necesarias para interactuar con la cadena de bloques
- Sección “Database”:
    - Parámetro “URL”: se especifica la URL de la base de datos que el nodo Chainlink utilizará para almacenar y gestionar datos.

Una vez se tiene los ficheros de configuración se puede proceder al levantamiento del nodo. En mi caso, se ha levantado el nodo en el puerto 6688 TCP. Sin embargo, no es posible conectarse a la máquina virtual por RDP, por lo que puedo realizar una regla de *port forwarding* para traerme el puerto 6688 de la máquina virtual que he levantado en la nube al mismo puerto en mi propia máquina (localhost). El comando que se ha ejecutado desde mi propia máquina para conseguir el reenvío de puertos es el siguiente:

```
ssh -i nodochainlink.pem user@20.23.13.24 -L 6688:localhost:6688 -N
```

A continuación, se explicará el comando detenidamente:

- "ssh": Es el comando para iniciar una sesión SSH.
- "-i nodochainlink.pem": Especifica la ruta y el nombre del archivo de clave privada que se utilizará para la autenticación de SSH.
- "user@20.23.13.24": Es el nombre de usuario y la dirección IP de la máquina en la nube a la que me deseo conectar.
- "-L 6688:localhost:6688": Establece un reenvío de puertos local. En este caso, se está redirigiendo el tráfico desde el puerto 6688 de la máquina en la nube hacia el puerto 6688 del equipo local (localhost).
- "-N": Esta opción indica que no se ejecute ningún comando remoto después de establecer la conexión SSH. Se utiliza cuando solo se necesita el reenvío de puertos y no se requiere una sesión interactiva.

Tras realizar el reenvío de puertos, puedo conectarme al nodo Chainlink a través de la siguiente URL en el navegador: http://localhost:6688/

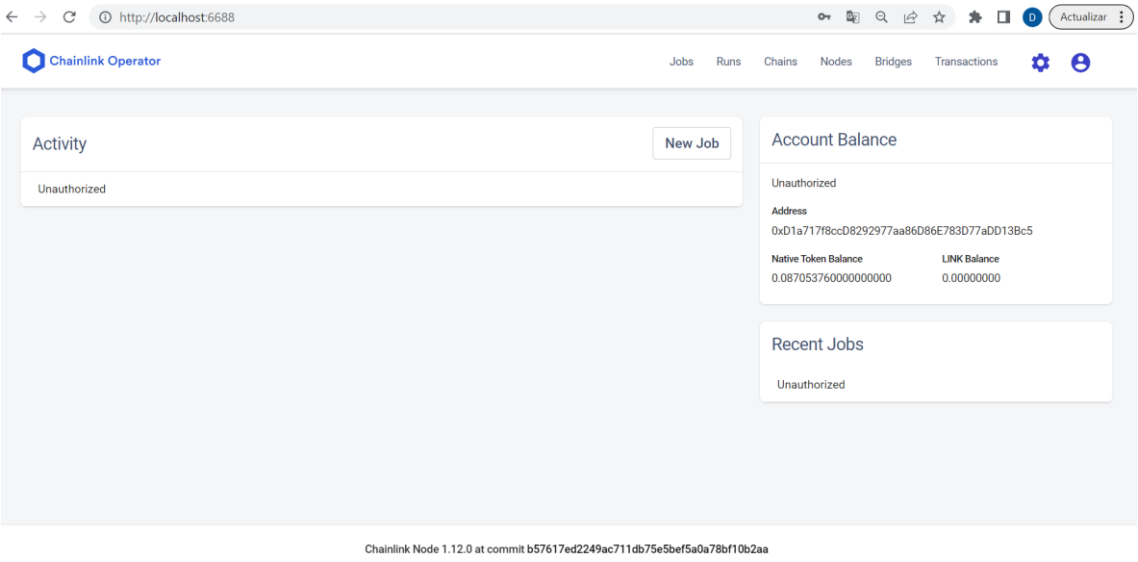

Ilustración 16: Interfaz nodo Chainlink

4.2 Creación de Jobs

Una vez se tiene el nodo creado, se tendrán que crear los Jobs, los cuales se ejecutarán cuando el contrato envíe la solicitud de un resultado al nodo Chainlink. Ambos Jobs deberán realizar una solicitud *HTTP GET* a su respectivo servidor (realmente el servidor está ubicado en la misma máquina, localhost, pero en distintos y endpoints) con los parámetros necesarios para que éste sea capaz de recuperar el resultado del evento deportivo de la API de SportsDataIO.

Como bien se ha comentado anteriormente, uno de ellos se ejecutará cuando el nodo reciba una llamada para una solicitud del resultado de un partido de fútbol, y el otro se ejecutará cuando el nodo reciba una solicitud de resultado de un torneo de golf.

La creación del Job consiste en un documento llamado ”especificación del Job” que contiene variables y las distintas tareas que deberá realizar el Job. Una de las variables que hay que indicar es la dirección del contrato oráculo que se ha tenido que desplegar con anterioridad. De esta manera, el Job no se puede crear hasta que el contrato Operator no haya sido desplegado.

Los documentos que definen la creación de cada Job se pueden observar en el anexo situado al final de la memoria.

A continuación, se procede a explicar un poco el documento que describe la función del Job que se encarga de recuperar el resultado de un partido de fútbol (en el caso del golf será muy parecido). En primer lugar, se definen una serie de variables como son el nombre del Job, la versión del esquema utilizado en la especificación del Job, el tipo del Job, la dirección del contrato *Operator* desplegado al principio, máxima duración durante la cual las tareas dentro de la especificación del Job pueden ejecutarse, y pago mínimo requerido en Juels (unidad de pago de Chainlink) para que el Job realice sus tareas.

Asimismo, la sección “observationSource” describe el flujo de trabajo que seguirá el Job. Las fases por las que está compuesta se enumeran a continuación:

1. **decode_log:** Se encarga de decodificar los registros de eventos (logs) almacenados en la cadena de bloques. Utiliza un ABI (Application Binary Interface) específico para decodificar los datos relevantes del evento.
2. **decode_cbor:** En esta fase, se realiza el análisis y decodificación del formato de datos CBOR (Concise Binary Object Representation). Se extraen los valores necesarios del objeto CBOR decodificado.
3. **Fetch:** Realiza una solicitud HTTP GET a la URL especificada (http://127.0.0.1:8080/endpoint) en la que se mandan en el cuerpo algunos parámetros necesarios para poder recuperar la información del evento deportivo. La respuesta de la solicitud se almacena para su posterior procesamiento.
4. **Parse:** Se realiza el análisis del contenido de la respuesta HTTP obtenida en la fase anterior. Se utiliza un analizador JSON para extraer los datos relevantes de la respuesta según una ruta o estructura específica.
5. **encode_data: C**odifica los datos recopilados y los prepara para su uso en la transacción de vuelta al contrato.
6. **encode_tx:** Codifica los datos necesarios para llamar a la función *fulfillOracleRequest* en el contrato del operador con los parámetros correctos. Los datos se codifican en el formato requerido por la función del contrato.
7. **submit_tx:** Envía la transacción al contrato del operador para ejecutar la función *fulfillOracleRequest* con los datos codificados. Esto permite que el contrato ejecute la lógica asociada y realice las operaciones necesarias.

En cuanto a la especificación del Job encargado de comunicarse con el servidor intermedio que obtiene resultados de los torneos de golf, los parámetros iniciales son iguales salvo por el nombre y el tiempo máximo que puede ejecutarse una tarea dentro del Job. En este caso, dicho valor máximo se ha fijado en 120 horas (cinco días) debido a que un torneo de golf tiene una duración de cuatro días y se da un día de margen para que los usuarios realicen las apuestas. Una vez dentro de la sección “observationSource” también se tiene todo bastante parecido al caso anterior salvo por la siguiente diferencia:

- En la fase “fetch”, lógicamente la URL a la que envía una solicitud GET es distinta. Esto se debe a que en este caso se está comunicando el servidor intermedio que recupera los resultados de los torneos de golf. Asimismo, los parámetros enviados en la solicitud también son distintos que en el caso anterior.

4.3 Configuración servidores intermedios

Cuando se tiene el nodo Chainlink levantado y con los Jobs configurados, se procede a la creación de dos servidores intermedios. Como se ha comentado anteriormente, cada uno de estos servidores tiene como función principal el almacenamiento de la clave API necesaria para realizar la consulta de los resultados a la API SportsDataIO. Con esto, lo que se consigue es que dicha clave no se envíe desde el *smart contract* hasta el nodo Chainlink, ya que dicha transacción quedaría reflejada permanentemente en la cadena de bloques y podría ser vista (y utilizada) por cualquiera, lo que supondría que gente puede hacerse pasar por el usuario de la clave.

Para levantar los servidores se ha utilizado el framework Flask de Python. Este framework permite crear aplicaciones web rápidas y sencillas, con un enfoque en la modularidad y simplicidad. El código Python que se ha utilizado para la creación de dichos servidores también se encuentra reflejado en el anexo y se recomienda su visualización para comprender su lógica.

El código y la lógica que realizan ambos servidores se pueden encontrar en el anexo. Sin embargo, se va a proceder a explicar el código de cada uno:

- En el caso del servidor intermedio de fútbol se importan los módulos flask y json necesarios para levantar el servidor flask y parsear los datos obtenidos de la API respectivamente. Además, se importa el módulo requests, el cual será útil para poder acceder a los parámetros que se le envían al servidor en la solicitud.
  El código cuenta con una función llamada *endpoint()* que únicamente se activa cuando recibe una solicitud HTTP GET, en la dirección en la que está levantado el servidor. Esto se puede comprobar porque se tiene la siguiente línea: `@app.route('/endpoint', methods=['GET'])`. En este caso, el servidor se encuentra levantado en la dirección localhost en el puerto 8080 en el enpoint “/endpoint”. Se sabe que el servidor se encuentra levantado en el puerto 8080 por la siguiente línea en el main: `app.run(host='0.0.0.0', port=8080)`. Además, el parámetro host=”0.0.0.0” indica que el servidor pueda ser accedido por todo el mundo.
  Volviendo a la función *endpoint().* Dentro de la dicha función se empiezan definiendo una serie de variables como pueden ser el array “data” y el booleano “final” inicializado a “False”.
  Posteriormente, la función se encarga de recuperar los parámetros enviados en la solicitud GET al servidor utilizando el módulo *request* y almacenando el valor en la variable *form_data*. A continuación, se agregan dichos parámetros al array “data” y se vuelca su contenido en un fichero llamado “parámetros.txt”.
  El siguiente paso es obtener los datos de los distintos parámetros que envía el usuario en la solicitud. Los parámetros que se deben enviar son “EquipoLocal”, “Jornada”, y “EquipoVisitante”. Los valores de dichos parámetros se almacenan en las variables “local”, “jornada”, y “visitante”. Los datos pasados al servidor deben estar en el orden

que se ha indicado, que es el orden que se ha especificado en la configuración del Job anteriormente.

Después hay un bucle que se continuará ejecutando hasta que la variable “final” tome el valor de “True”. En dicho bucle se inicializa la variable “api_key” con el valor de la clave API necesaria para acceder a API de SportsDataIO. Asimismo, se inicializa la variable “url” con la url de la API a la que hay que acceder para recuperar los resultados de los partidos de fútbol. La respuesta obtenida a la solicitud se almacena en la variable “response”. Más adelante, se parsea la información obtenida utilizado la función *loads()* del módulo json y se almacena la respuesta obtenida parseada en la variable “parsed_data”.

A continuación, se procede a filtrar la respuesta obtenida de SportsDataIO hasta obtener el número de goles del equipo local y del equipo visitante y así poder comprobar quien ha ganado el partido. Eso sí, antes de eso hay que comprobar que el partido ha terminado. Para ello se comprueba que el parámetro *["Games"][x]["Status"] == "Final"*.

Una vez se obtiene los goles marcados por ambos equipos se cambia el valor de la variable “final” a “True”, de manera que el bucle dejará de ejecutarse.

Cabe destacar que, en caso de que el partido no se haya terminado, el servidor seguirá haciendo llamadas a la API, hasta que el parámetro recientemente mencionado indique que efectivamente el partido ha terminado.

Después, se realiza una comparación de los goles que ha marcado cada equipo y se establece un ganador. Es decir, en caso de que el equipo local haya marcado más goles que el equipo visitante, significa que el equipo local ha ganado, por lo que se inicializa la variable resultado a 1. En caso de que el número de goles marcados por ambos equipos sea el mismo, el partido ha terminado en empate y la variable resultado se inicializa a 0. Finalmente, en caso de que el equipo visitante haya marcado más goles que el equipo local, el equipo visitante se ha proclamado vencedor y la variable resultado se inicializa a 2.

Finalmente, el valor de la variable resultado se devuelve a quien ha llamado la función, que en este caso será el Job de fútbol del nodo Chainlink que se ha desplegado.

- En cuanto al servidor intermedio que se encarga de obtener los resultados de los torneos de golf de la API de SportsDataIO hay que comentar que esta vez el servidor está levantado en el puerto 8081 y ubicado en el endpoint “/golf”. En este caso, se realiza la misma importación de módulos que en el caso del código del servidor de fútbol y también se cuenta con la función *endpoint()* que será ejecutada cuando el servidor reciba una solicitud GET.

  La obtención de los valores de los parámetros dentro de la solicitud GET se obtienen de la misma manera que en el caso anterior. Salvo que, en este caso, solo se envía el parámetro “id_torneo”, el cual indica el torneo del cual se quiere obtener el ganador. El valor del parámetro recibido se almacena en la variable “id_torneo”.

  Posteriormente, el código cuenta también con un bucle muy parecido al del caso de obtención de resultados de fútbol, salvo por el caso de que la variable “url” en este caso especifica la URL a la que hay que acceder para recuperar los datos del torneo específico. Además, la clave API es distinta, ya que se ha tenido que crear una cuenta nueva para acceder a los resultados de los torneos de golf.

  Dentro del bucle, también se comprueba que el torneo ha terminado viendo el valor de *parsed_data["Tournament"]["IsOver"]*. Una vez se comprueba que el torneo ha

terminado, se extrae el nombre ganador del torneo y se envía de vuelta dicho resultado.

4.3.1 Pruebas Servidores Intermedios

En esta sección se van a realizar una serie de pruebas para comprobar que los servidores intermedios funcionan correctamente.

En primer lugar, se va a probar el servidor intermedio que recupera los resultados de los partidos de fútbol. Para ello, primero hay que levantar el servidor intermedio y, una vez se tiene levantado, se va a lanzar otro script que realiza una solicitud GET a dicho servidor intermedio. Recordemos que los Jobs del nodo Chainlink se han configurado para realizar solicitudes GET al servidor intermedio, de manera que con esta prueba se está probando como se comportará el servidor intermedio cuando reciba las solicitudes:

```python
import requests

# URL del servidor
url = 'http://127.0.0.1:8080/endpoint'

# Parámetros de la solicitud GET
params = {
    'EquipoLocal': 'Celta',
    'Jornada': '3',
    'EquipoVisitante':'Madrid'
}

# Realizar la solicitud GET y obtener la respuesta
response = requests.get(url, params=params)

# Obtener el contenido de la respuesta
content = response.text

# Imprimir el contenido de la respuesta
print(content)
```

```
2
```

Ilustración 17: Prueba funcionamiento servidor intermedio fútbol

En la imagen anterior se puede ver el script que se ha utilizado para realizar la solicitud al servidor intermedio. En este caso el servidor intermedio se tiene levantado en la propia máquina. De esta manera, la solicitud se debe hacer a la dirección localhost (127.0.0.1). En el caso de verdad la solicitud GET que envía el Job también debe ser a la dirección de localhost, ya que tanto el nodo Chainlink, como el servidor intermedio están levantados en la misma máquina virtual de Azure.

En la imagen también se puede observar como el servidor intermedio devuelve el resultado “2”, indicando que el Real Madrid ha ganado el partido que el enfrentaba al Celta de Vigo en la tercera jornada de la liga española.

Si vemos los logs que ha dejado el servidor intermedio encontramos lo siguiente:

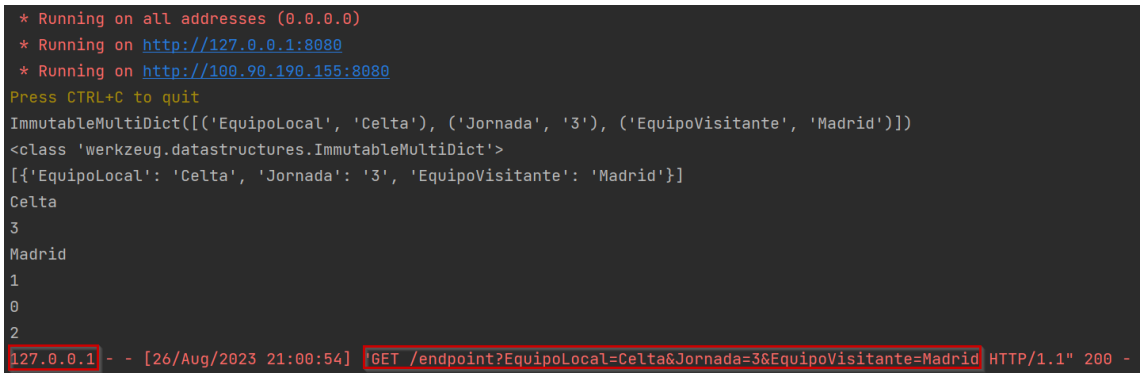

Ilustración 18: Logs servidor intermedio fútbol

Se puede contemplar como el servidor intermedio se encuentra levantado en la IP 127.0.0.1 en el puerto 8080. Asimismo, la IP de la máquina desde la cual se está realizando la solicitud al servidor intermedio también es la de localhost. Finalmente, se puede ver la URL completa que se ha construido y se puede comprobar como ésta es la misma que utiliza el Job (se puede ver en el código de la especificación de Job situado en el Anexo).

A continuación, se va a realizar la misma prueba para comprobar el correcto funcionamiento del servidor intermedio que recopila los resultados de los torneos de golf.

En este caso, también se levanta el servidor intermedio, y posteriormente se utilizará otro script, que será lanzado desde la misma máquina, que será el encargado de realizar una solicitud GET con el envío de parámetros oportunos.

Una vez se tiene el servidor levantado, se procede a lanzar el siguiente script:

```python
import requests

# URL del servidor
url = 'http://127.0.0.1:8081/golf'

# Parámetros de la solicitud GET
params = {
    'id_torneo': '553',
    'jugador': 'Rickie Fowler'
}

# Realizar la solicitud GET y obtener la respuesta
response = requests.get(url, params=params)

# Obtener el contenido de la respuesta
content = response.text

# Imprimir el contenido de la respuesta
print(content)
```

```
1
```

Ilustración 19: Prueba funcionamiento servidor intermedio golf

El script es muy similar al del caso anterior, salvo por la diferencia de que el servidor espera solicitudes en un endpoint y puerto distinto. Además, los parámetros que han de ser enviados en la solicitud GET cambian también, como es lógico.

Se puede observar como el servidor intermedio devuelve un “1”, lo que indica que el jugador que se le ha pasado como argumento en la solicitud GET sí que ha sido el ganador del torneo que se la especificado mediante el parámetro “id_torneo”.

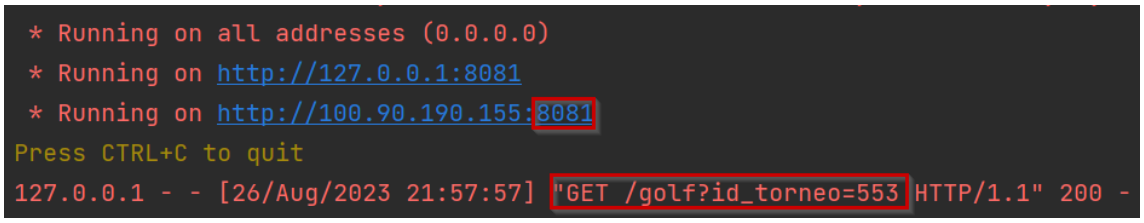

Ilustración 20: Logs servidor intermedio golf

Viendo los logs del servidor intermedio se comprueba como el servidor se encuentra levantado en el puerto 8081 y está ubicado en el endpoint “golf”. Asimismo, se puede ver como la URL utilizada corresponde nuevamente con la utilizada en el caso del Job de golf.

4.3.2 Uso TMUX

El nodo Chainlink y los servidores intermedios están corriendo en una máquina virtual en la nube. Para realizar funciones en esa máquina hay que conectarse a ella a través de SSH. Para que los procesos arrancados, como pueden ser dichos servidores intermedios, no mueran al cerrar la conexión SSH se ha utilizado la herramienta TMUX.

TMUX es un multiplexor de terminal de Linux que permite ejecutar varios programas Linux en una sola conexión, como cuando uno se conecta de forma remota a una máquina a través de SSH.

Una de las muchas ventajas de TMUX es que desacopla los programas de la terminal principal, protegiéndolos de una desconexión accidental. De esta manera, se puede desconectar TMUX de la terminal actual y todos sus programas continuarán ejecutándose de forma segura en segundo plano.

Además de sus beneficios con conexiones remotas, la velocidad y flexibilidad de TMUX lo convierten en una herramienta fantástica para administrar múltiples terminales en su máquina local, similar a un administrador de ventanas.

TMUX solo se puede utilizar en Windows a través de Windows Subsystem for Linux (WSL), por ejemplo, WSL2 (Windows Subsystem for Linux 2). Con WSL2, tienes acceso a las aplicaciones y herramientas de Linux en Windows.

Los aspectos más importantes del funcionamiento de TMUX son las sesiones de terminal, las ventanas y los paneles:

- Sesiones de terminal: en las sesiones de terminal se ejecutan tareas y comandos, y se maneja la distribución de Linux.
- Ventanas: si deseas realizar tareas específicas por separado dentro de una sesión, puedes utilizar tmux para trabajar en paralelo con varias ventanas en una sesión al mismo tiempo.
- Paneles: también puedes dividir estas ventanas en diferentes áreas (paneles o “Panels”), para ejecutar, por ejemplo, logs de errores mientras realizas otras tareas en la misma ventana.

4.3.2.1 Demostración TMUX

Cuando se ha establecido una conexión con la máquina virtual de Azure, se puede ejecutar el comando "`tmux new-session -s golf`". Tras ejecutar este comando, saltará otra ventana distinta en la que se podrá ejecutar el servidor intermedio de golf.

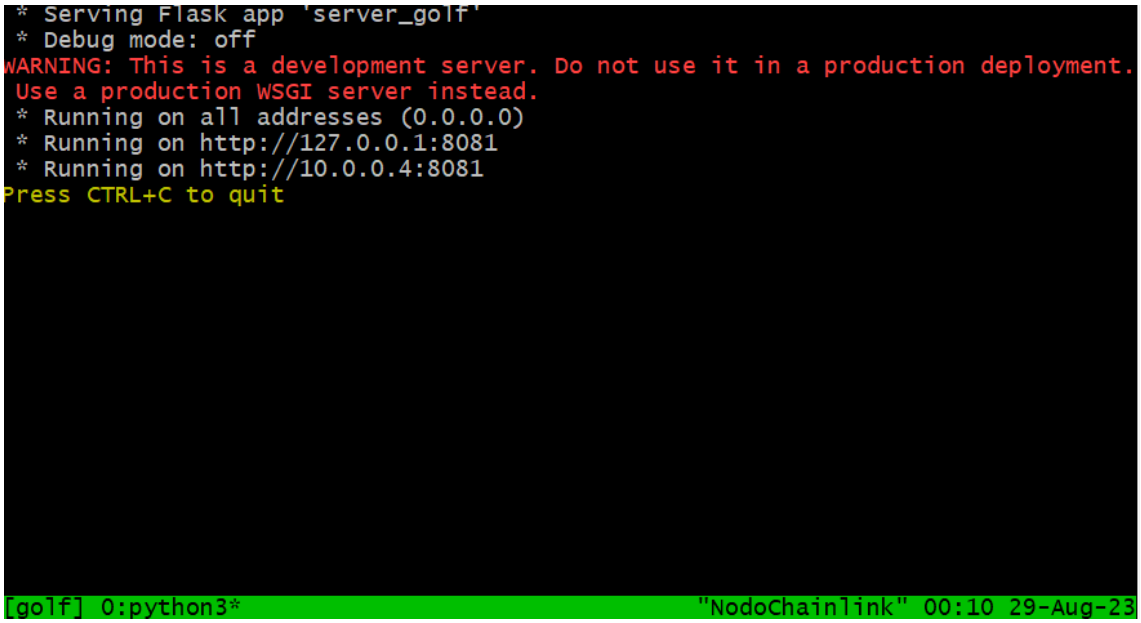

Ilustración 21: Sesión TMUX

Para salir de la sesión TMUX sin cerrarla basta con ejecutar Ctrl + 'b' seguido de 'd'.

Posteriormente, hay que hacer lo propio con una sesión que se encargue de levantar el servidor intermedio que recopila los resultados de fútbol.

De esta manera, si se sale de la conexión SSH y se vuelve a entrar y se ejecuta el comando "`ps -aux | grep serv`" para comprobar si los programas que se encargan de levantar los servidores se siguen ejecutando, se podrá comprobar que si:

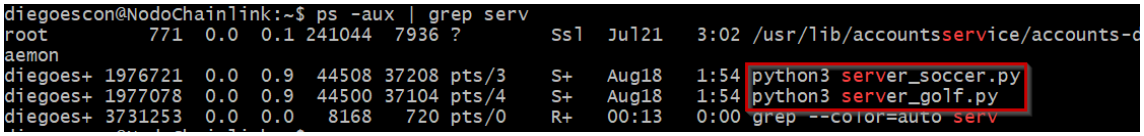

Ilustración 22: Comprobación ejecución servidores intermedios

Otra manera de comprobar si los servidores siguen levantados consiste en meterse de nuevo en la sesión TMUX creada anteriormente. Para entrar de nuevo en la sesión "golf" creada anteriormente se puede ejecutar el comando "`tmux attach-session -t golf`".

Ahora ya se tienen todas las piezas del puzle, por lo que únicamente falta ponerlas en práctica. Recordemos el funcionamiento:

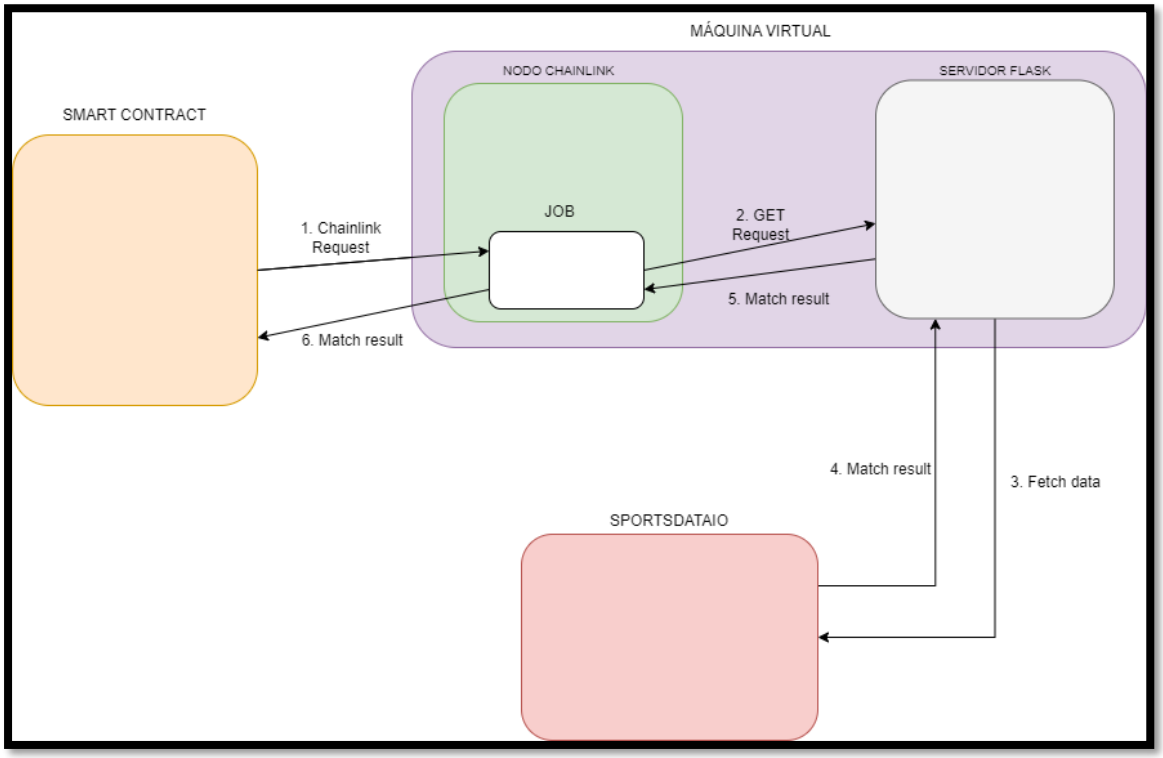

Ilustración 23: Funcionamiento proyecto

En la imagen anterior se indican todas las partes involucradas en el proceso tanto de la realización de una apuesta, obtención de resultados y entrega de estos al *smart contract*.

Recordemos que en el paso número tres, el cual consiste en la llamada a la API para la obtención de resultados, se envía la clave API que se almacena en el propio servidor intermedio.

Ahora, será necesario realizar el despliegue del contrato. Se recuerda que al desplegar un contrato en la red Ethereum se está ejecutando una transacción, y como todas las transacciones en Ethereum, se requiere gas para pagar a los validadores de la red por procesar y confirmar la transacción.

Como se ha comentado durante el desarrollo del contrato, la cantidad de gas requerida para desplegar un contrato dependerá de varios factores:

1. Tamaño del contrato: Cuanto más grande es el contrato, más gas se requiere para desplegarlo.
2. Complejidad del contrato: Cuanto más complejas son las funciones y las operaciones realizadas en el contrato, más gas se requiere para su despliegue.
3. Uso de bibliotecas externas: Si el contrato utiliza bibliotecas externas, se requiere gas adicional para su enlace.
4. Demanda de la red: Si la red está congestionada y hay muchas transacciones en cola, el costo del gas puede aumentar debido a la competencia por los recursos de la red.

Capítulo 5

5. Complicaciones encontradas a lo largo del proyecto

5.1 Limitaciones para probar el funcionamiento

Durante la realización del proyecto se han encontrado una serie de obstáculos que han impedido lograr los objetivos del proyecto.

Una razón que ha impedido lograr el objetivo final ha sido que este proyecto solo podía ser probado los fines de semana, ya que es durante ese tiempo en el que hay partidos de futbol disponibles para apostar (lo mismo se aplica a los torneos de golf). Esto ha ralentizado bastante el proceso, y se debería de haber planeado con la antelación suficiente para evitar lo que ha sucedido.

5.2 Añadir ETH al nodo Chainlink

La siguiente complicación que ha surgido y que también ha impedido lograr los objetivos ha sido que no se ha podido dotar a la cuenta a la que está asociada el nodo Chainlink de tokens ETH.

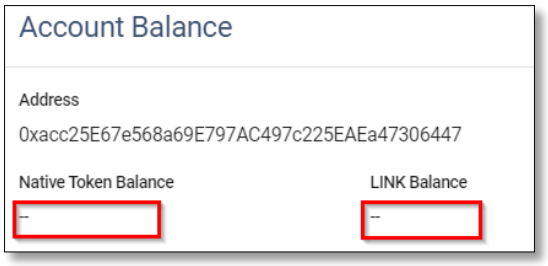

Ilustración 24: Balances Nodo Chainlink

Como se puede ver en la imagen anterior, el balance del token nativo es inexistente. Sin embargo, la transacción que se envía a la dirección que se muestra en la imagen sí que se ha efectuado correctamente. La transacción efectuada se puede ver a continuación:

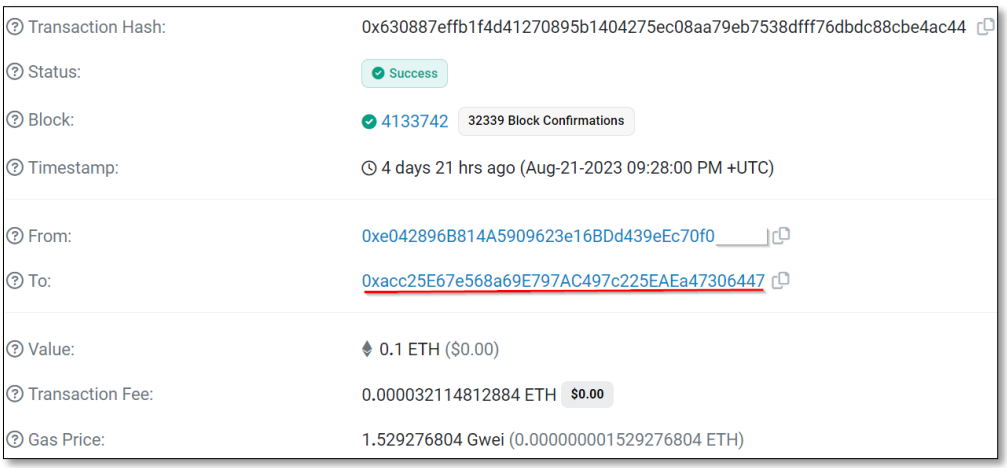

Ilustración 25: Confirmación transaccion nodo Chainlink

Como se puede ver, la transacción se efectúa satisfactoriamente a la dirección del nodo Chainlink, pero en ningún momento parece que la recibe. Recordemos que el nodo Chainlink necesita tener cierto balance de ETH positivo para poder realizar la llamada de vuelta al contrato en la que le comunica al oráculo los resultados del evento deportivo. Esto se debe a que como se ha comentado anteriormente, la realización de una transacción en la cadena de bloques (en este caso una llamada a una función de un contrato) requiere el pago de cierta comisión llamada gas.

En la primera imagen, también se ha podido ver como el balance de tokens LINK es inexistente. Esto nos dice que la comunicación entre el contrato y nodo Chainlink no se está realizando correctamente. Anteriormente se ha comentado que el contrato debe incluir ciertos tokens LINK en la llamada al oráculo (valor definido en la variable *fee* del contrato) para recompensar al nodo Chainlink por sus servicios. El hecho de que el nodo no cuente con tokens LINK parece indicar que la comunicación entre el contrato y el nodo Chainlink no se está realizando de manera correcta, lo que lleva al siguiente punto.

5.3 Conexión contrato – oráculo

La conexión entre el contrato y el nodo Chainlink no está funcionando. La razón exacta por la cual la comunicación no se está estableciendo no se sabe. Sin embargo, se plantea una posibilidad.

El contrato hace uso de una función llamada *buildChianlinkRequest()* del contrato *ChainlinkClient* que es importado al principio del documento Solidity. Dicha función se muestra a continuación:

```
function buildChainlinkRequest(
  bytes32 specId,
  address callbackAddr,
  bytes4 callbackFunctionSignature
) internal pure returns (Chainlink.Request memory) {
  Chainlink.Request memory req;
  return req.initialize(specId, callbackAddr, callbackFunctionSignature);
}
```

Ilustración 26: Función buidChainlinkRequest()

Esta función se encarga de crear la solicitud que posteriormente será enviada al oráculo tras añadir los distintos argumentos que permitirán al Job obtener los resultados correspondientes. Como se puede observar el primer argumento que se le pasa a la función debe ser *jobID* con el que se va a contactar. Esta variable debe ser de tipo *Bytes32*. Sin embargo, como se puede observar en la siguiente imagen, el *jobID* al que se puede acceder desde la interfaz del nodo Chainlink se encuentra en formato *String*.

Ilustración 27: JobID en formato String

De esta manera, hay que realizar una conversión de *String* a *Bytes32* para que se pueda llamar a la función *buidChainlinkRequest()* sin errores.

Para la conversión de *String* a *Bytes32* se ha utilizado una función llamada *stringToBytes32()* que se ha obtenido de la página web de Chainlink. Dicha función es la siguiente:

```
function stringToBytes32(
    string memory source
) private pure returns (bytes32 result) {
    bytes memory tempEmptyStringTest = bytes(source);
    if (tempEmptyStringTest.length == 0) {
        return 0x0;
    }

    assembly {
        // solhint-disable-line no-inline-assembly
        result := mload(add(source, 32))
    }
}
```

Ilustración 28: Función stringToBytes32()

Cada vez que se llama a la función *setJobId_resultados_soccer* o *setJobId_resultados_golf* se realiza esta conversión de *String* a *Bytes32* utilizando esta función.

La posibilidad que se plantea es que la función *stringToBytes32()* no convierta a *Bytes32* de manera correcta. De todas formas, más información acerca de la utilización de esta función se detalla en la siguiente sección.

Otra razón por la cual se cree que la conexión del contrato con el oráculo es que la máquina virtual que se ha levantado y que se utiliza para hospedar el nodo Chainlinlink y los servidores intermedios no cuenta con los recursos suficientes.

Esta suposición se ha sacado tras ver que máquina virtual cuenta con los recursos mínimos para hospedar un nodo Chainlink de lo más sencillo. Además, hay que tener en cuenta que esta máquina también está continuamente ejecutando unos programas que consumen recursos.

Capítulo 6

6. Conclusiones y líneas Futuras de Investigación

6.1 Conclusiones

Nuevamente, hay que recalcar que los objetivos fijados al principio del proyecto no se han obtenido. Se ha sido demasiado ambicioso al querer implementar una solución en un campo en el que no hay mucha información y, en ocasiones, puede ser hasta errónea.

La confusión entre los distintos métodos para realizar la integración con los oráculos ha provocado no aclararse sobre cómo hacer que finalmente el nodo Chainlink recibe la solicitud enviada por el contrato. Es más, en la página web oficial de Chainlink, se ha encontrado información contradictoria. De esta manera, la primera conclusión directa que se extrae es que la información en Chainlink no está adecuadamente revisada y resulta confusa.

Un ejemplo de información contradictoria que se ha encontrado está en el siguiente contrato de ejemplo:

```
bytes32 private jobId;
uint256 private fee;

event RequestGasPrice(
    bytes32 indexed requestId,
    uint256 gasPriceFast,
    uint256 gasPriceAverage,
    uint256 gasPriceSafe
);

/**
* @notice Initialize the link token and target oracle
*
* Sepolia Testnet details:
* Link Token: 0x779877A7B0D9E8603169DdbD7836e478b4624789
* Oracle: 0x6090149792dAAeE9D1D568c9f9a6F6B46AA29eFD (Chainlink DevRel)
* jobId: 7223acbd01654282865b678924126013
*
*/
constructor() ConfirmedOwner(msg.sender) {
    setChainlinkToken(0x779877A7B0D9E8603169DdbD7836e478b4624789);
    setChainlinkOracle(0x6090149792dAAeE9D1D568c9f9a6F6B46AA29eFD);
    jobId = "7223acbd01654282865b678924126013";
    fee = (1 * LINK_DIVISIBILITY) / 10; // 0,1 * 10**18 (Varies by network and job)
}

/**
* Create a Chainlink request the gas price from Etherscan
*/
function requestGasPrice() public returns (bytes32 requestId) {
    Chainlink.Request memory req = buildChainlinkRequest(
        jobId,
        address(this),
        this.fulfill.selector
    );
```

Ilustración 29: Error en contrato Chainlink

En dicho ejemplo se puede comprobar como en el constructor se inicializa la variable *jobID*, la cual es de tipo bytes32 con una cadena alfanumérica y, posteriormente, se pasa dicho valor directamente como argumento en la función *buildChainlinkRequest().* En este ejemplo no se hace uso de la función stringToBytes32() que se ha comentado anteriormente, sino que se incluye directamente el *jobID* sacado de la interfaz del nodo Chainlink sin guiones.

Sin embargo, si nos vamos a otro ejemplo de la página web de Chainlink que muestra un ejemplo completo sobre como realizar llamadas a un oráculo se puede observar que sí que se hace uso de una función que convierta de *String* a *Bytes32* (https://docs.chain.link/chainlink-nodes/v1/fulfilling-requests).

```
function requestEthereumPrice(
    address _oracle,
    string memory _jobId
) public onlyOwner {
    Chainlink.Request memory req = buildChainlinkRequest(
        stringToBytes32(_jobId),
        address(this),
        this.fulfillEthereumPrice.selector
    );
    req.add(
        "get",
        "https://min-api.cryptocompare.com/data/price?fsym=ETH&tsyms=USD"
    );
    req.add("path", "USD");
    req.addInt("times", 100);
    sendChainlinkRequestTo(_oracle, req, ORACLE_PAYMENT);
}
```

Ilustración 30: Ejemplo utilización función stringToBytes32().

Ha sido precisamente de este ejemplo de donde se ha sacado la función *stringToBytes32()* que se ha utilizado en el contrato desarrollado en este proyecto. Se debe recalcar que se han probado ambas alternativas en el proyecto. Es decir, se ha probado a hacer uso de la función que convierte una variable de tipo *String* a otra de *Bytes32*, y también se ha probado a no usar dicha función y pasar el *jobID* directamente como argumento a la función *buildChainlinkRequest()*, pero ninguna de las opciones ha funcionado.

A pesar de no haber conseguido unir las partes del oráculo y el contrato inteligente, se ha aprendido bastante a cerca de este mundo tan apasionante como es el de Blockchain. De esta manera, se pueden seguir extrayendo ciertas conclusiones:

1. **Ventajas de la tecnología Blockchain en apuestas online.**
   La transparencia es una de las cualidades más distintivas de la tecnología Blockchain. En el contexto de las apuestas en línea, esta característica adquiere un valor excepcional, ya que elimina gran parte de la desconfianza que los usuarios pueden tener hacia las casas de apuestas tradicionales. Al registrar todas las transacciones y eventos en la cadena de bloques de manera inmutable y accesible públicamente, se crea un registro transparente y verificable de todas las apuestas y resultados.
   Esta transparencia no solo protege a los usuarios al prevenir la manipulación de resultados y el fraude interno, sino que también fomenta un ambiente de confianza mutua entre los participantes. Los usuarios son capaces de verificar de manera independiente los resultados y las transacciones, lo que reduce significativamente la necesidad de confiar en intermediarios y terceros. Además, el registro público de eventos y transacciones permite una auditoría precisa y efectiva de las operaciones, lo que puede ser crucial en caso de disputas.
2. **Eficiencia y Automatización**.
   La automatización se convierte en un activo fundamental en este caso de uso. Los contratos inteligentes están diseñados para ejecutarse automáticamente cuando se cumplen condiciones específicas predefinidas. En el contexto de este proyecto, esto significa que una vez que se alcanza el fin de un evento deportivo, el contrato inteligente se activará de manera automática para determinar el resultado de la apuesta, calcular la cuota que debe ser multiplicada por la cantidad invertida por el usuario para calcular así las ganancias que se le deben, y la distribución de las mismas en caso de que el usuario haya resultado ganador.

Esta automatización ofrece varios beneficios. En primer lugar, elimina la necesidad de intervención humana en la mayoría de las etapas del proceso, lo cual disminuye la probabilidad de errores humanos y agiliza el proceso general. Esto a su vez puede mejorar la satisfacción del usuario al recibir resultados rápidos y precisos (dadas la característica de Chainlink de consenso y rapidez).

En segundo lugar, la automatización también puede disminuir los costes operativos al reducir la cantidad de recursos humanos necesarios para administrar y supervisar el proceso. Los contratos inteligentes se ejecutan en la cadena de bloques, lo que significa que la lógica y la ejecución de las operaciones están aseguradas de manera descentralizada y transparente, sin requerir intermediarios costosos.

Por último, esta transformación hacia la automatización también puede impactar positivamente en la escalabilidad del sistema, ya que las operaciones pueden realizarse de manera simultánea y eficiente sin necesidad de intervención manual.

3. **Interacción con el contrato inteligente**.

Pese a que estos contratos brindan numerosas ventajas en términos de seguridad, confiabilidad y transparencia, la forma en que los usuarios interactúan con ellos puede diferir significativamente de las interfaces de usuario tradicionales que se encuentran en las casas de apuestas convencionales. A continuación, se muestra una interfaz de una casa de apuestas online:

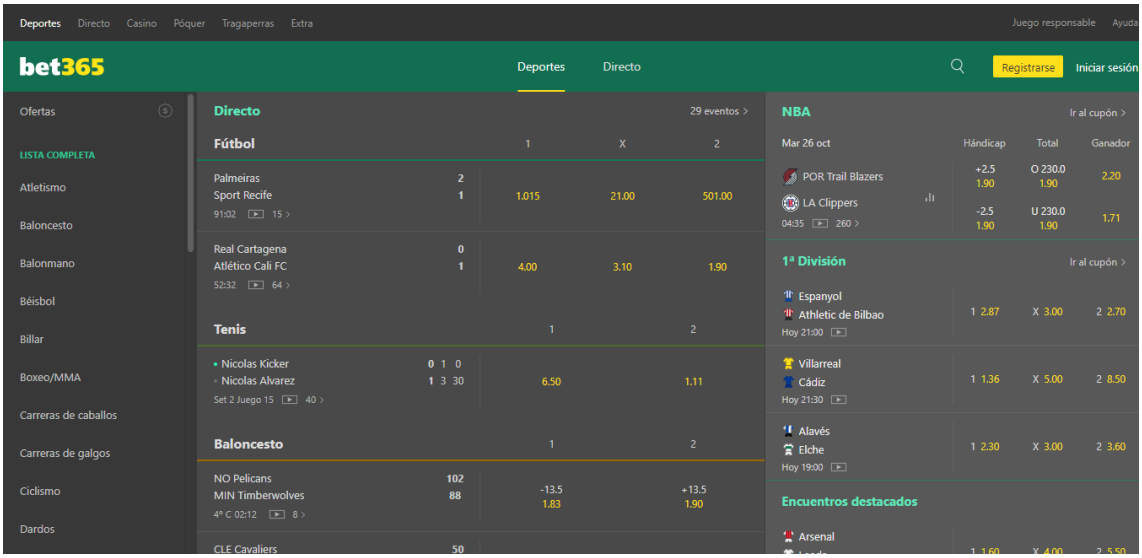

Ilustración 31: Interfaz casa de apuestas online

Como se puede observar, la interfaz es bastante intuitiva y amigable, lo cual puede mejorar la experiencia de usuario. Estas interfaces les permiten explorar eventos deportivos, elegir entre diferentes tipos de apuestas, ingresar la cantidad que desean apostar y confirmar sus apuestas con facilidad.

Por otro lado, al cambiar a un sistema basado en contratos inteligentes, la interacción puede volverse más técnica y menos intuitiva para los jugadores. A continuación, se muestra la interfaz a través de la cual hay que interactuar con el contrato para realizar una apuesta.

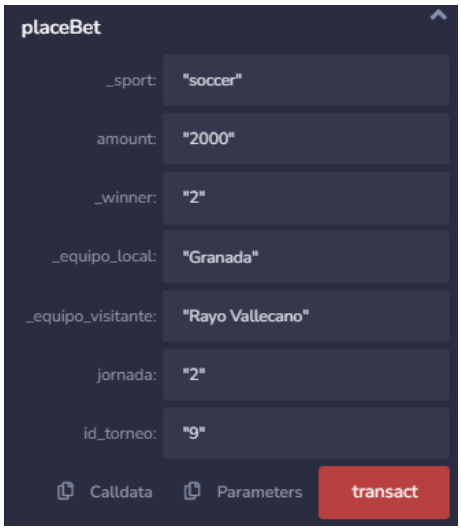

Ilustración 32: Interfaz interacción con contrato inteligente

Esta interfaz no es más que la propia de Remix y, como se puede observar, los usuarios deben realizar llamadas directas a las funciones específicas de los contratos inteligentes.

Esta transición puede ser desafiante para algunos usuarios, especialmente para aquellos que no están familiarizados con la programación o el funcionamiento interno de los contratos inteligentes. La falta de una interfaz gráfica puede resultar en una experiencia de usuario menos cómoda y más propensa a errores al interactuar con los contratos inteligentes. Esto podría afectar la adopción general de la plataforma, ya que algunos usuarios pueden preferir la comodidad y la familiaridad de las interfaces tradicionales antes que cualquiera de los beneficios que aporta la tecnología Blockchain en este ámbito.

4. **Falta de contenido para apostar**.
   El enfoque inicial del trabajo fue la creación de un sistema de apuestas para un deporte específico (el fútbol). Sin embargo, el hecho de que durante los meses de mayo, junio y julio no hubiera partidos de fútbol debido al parón de verano provocaron que se tuviera que buscar otro deporte al que apostar durante ese tiempo para probar el funcionamiento. Fue ahí cuando entró en juego el deporte de golf en el proyecto.
   Pese a que finalmente se cuentan con dos deportes a los que poder apostar en vez de uno, no termina de ser suficiente. Esto se debe a que normalmente los usuarios quieren realizar las apuestas todas a través de un único sitio. Sin embargo, la poca oferta de deportes disponibles a los que poder apostar pueden echar al usuario para atrás a la hora de utilizar la tecnología Blockchain para realizar una apuesta.
5. **Desafíos regulatorios**.
   Uno de los principales desafíos que enfrentan las plataformas de apuestas en línea basadas en Blockchain es la verificación de la identidad de los usuarios. En muchos países, las regulaciones exigen que las plataformas de apuestas realicen un proceso de verificación riguroso para prevenir el lavado de dinero, el fraude y el acceso de menores de edad. Sin embargo, en un entorno descentralizado donde los usuarios interactúan directamente con contratos inteligentes, el proceso de verificación puede ser más complejo.

La tecnología Blockchain, al ser transparente y pública, presenta un conflicto con la privacidad de los usuarios. Mientras que la identidad de las partes involucradas en las transacciones puede permanecer en gran medida anónima, las regulaciones pueden requerir que las plataformas cumplan con ciertos estándares de verificación y seguimiento. En este sentido, encontrar el equilibrio adecuado entre la privacidad de los usuarios y los requisitos regulatorios es un desafío crítico.

6. **Impacto en la industria**.
   Algunos de los aspectos clave a considerar en este contexto son los siguientes:
   - Internacionalización sin Fronteras: La característica de descentralización de la tecnología Blockchain va más allá de las fronteras geográficas y las limitaciones regulatorias tradicionales. Esto provoca que las plataformas de apuestas en línea basadas en Blockchain operen a nivel global sin estar restringidas por las criterios legales y regulatorias de diferentes países. Los jugadores de todo el mundo podrían participar en las mismas plataformas sin obstáculos geográficos.
   - Eliminación de Intermediarios: Como bien se ha comentado a lo largo del proyecto, los contratos inteligentes permiten la automatización y ejecución directa de acuerdos entre las partes involucradas. Al eliminar intermediarios, se pueden reducir los costos y agilizar los procesos. Los usuarios podrían interactuar directamente con los contratos inteligentes, lo que potencialmente podría conducir a cuotas más favorables y mayores ganancias para los jugadores.
   - Cambio de Paradigma en la Experiencia del Usuario: Aunque la interacción con contratos inteligentes puede presentar desafíos en términos de accesibilidad para algunos usuarios, podría también cambiar la forma en que los jugadores interactúan con las apuestas en línea. La adopción generalizada podría impulsar la creación de interfaces de usuario más intuitivas y atractivas, facilitando la transición hacia la nueva forma de apostar.

6.2 Líneas futuras de investigación

En los últimos años, la tecnología Blockchain ha revolucionado diversos sectores, brindando oportunidades innovadoras y disruptivas en múltiples áreas. Uno de estos sectores es el de las apuestas online, donde la aplicación de la tecnología Blockchain ha abierto nuevas posibilidades para mejorar la transparencia, la seguridad y la confiabilidad de los sistemas existentes. Este trabajo se centra en el desarrollo de un sistema de apuestas online basado en Blockchain, que busca aprovechar las ventajas inherentes de esta tecnología descentralizada para crear una plataforma más justa y confiable para los usuarios.

En este trabajo tenemos al contrato, que es el responsable de realizar el cálculo de cuotas. Éste itera por todas las apuestas que se han realizado a los distintos partidos y luego en función de las veces que se ha apostado a cada equipo del partido calcula una cuota asociada a cada equipo.

Esto puede suponer dos problemas:

1. El hecho de iterar sobre un array de estructuras y comparar si un atributo de cada estructura es igual a otro atributo supone un gran gasto computacional, lo que lleva también a un consumo de gas elevado. De esta manera, el usuario que desea realizar una apuesta deberá dedicar una cantidad considerable a las comisiones de gas.
2. Como bien se ha comentado, el cálculo de cuotas se realiza una vez el usuario ha realizado su apuesta, por lo que a la hora de apostar no sabe la posible cantidad de dinero que puede ganar. Esto se podría arreglar empleando otro oráculo que se conectase también a la SportsDataIO y obtuviera de ahí las cuotas asignadas a cada partido. Con este enfoque, el usuario podría saber la cuota del partido al que desea apostar al principio y luego decidir si finalmente apuesta a ese partido o no.

Asimismo, podría ser interesante desplegar un front-end con el que los jugadores pudieran interactuar directamente con el contrato a través de él y poder realizar apuestas. Algunas de las ventajas que aportaría este enfoque se detallan a continuación:

1. **Nueva experiencia de usuario mejorada**: Un front-end proporcionaría una interfaz amigable y fácil de usar para los jugadores. Esto permitiría una adopción más amplia y una participación más activa de los usuarios.
2. **Interacción intuitiva**: El front-end podría ofrecer una forma intuitiva de visualizar y seleccionar las opciones de apuestas, realizar transacciones y realizar seguimiento de los resultados
3. **Integración de servicios adicionales**: Sería posible integrar servicios adicionales como sistemas de autenticación de usuarios, métodos de pago, visualización de estadísticas y resultados en tiempo real, entre otros. Esto enriquecería la funcionalidad del sistema y brindaría características adicionales para los jugadores.
4. **Adaptabilidad y escalabilidad**: Al separar la interfaz de usuario del contrato inteligente, se facilita la evolución y actualización del sistema en el futuro. Se pueden realizar mejoras en el front-end sin afectar directamente al contrato inteligente subyacente, lo que brinda flexibilidad y escalabilidad al sistema.

Anexo

Ficheros de Configuración Nodo Chainlink

Archivo config.toml

```
[Log]
Level = 'warn'

[WebServer]
AllowOrigins = '\*'
SecureCookies = false

[WebServer.TLS]
HTTPSPort = 0

[[EVM]]
ChainID = '11155111'

[[EVM.Nodes]]
Name = 'Sepolia'
WSURL =
'wss://mainnet.infura.io/ws/v3/baace7c11a584b2da90fd72dfd297014'
HTTPURL =
'https://mainnet.infura.io/v3/baace7c11a584b2da90fd72dfd297014'
```

Archivo secrets.toml

```
[Password]
Keystore = 'mysecretkeystorepassword'
[Database]
URL =
'postgresql://postgres:mysecretpassword@host.docker.internal:5432/post
gres?sslmode=disable'
```

Código Smart Contract

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;

import "@chainlink/contracts/src/v0.8/ChainlinkClient.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/utils/math/SafeMath.sol";
import "@openzeppelin/contracts/utils/Strings.sol";

contract SportsDataOracle is ChainlinkClient, Ownable { using SafeMath
for
uint256; using Chainlink for Chainlink.Request;
```

```
    mapping(address => uint) public balances;
    struct Bet_soccer {
        address player;
        uint amount;
        string winner; // 1 = home team wins, 2 = away team wins, 0 =
draw
        uint8 jornada; // week of the game in which the user is
betting
        string equipo_local;
        string equipo_visitante;
        bool completed;
    }

    Bet_soccer[] public bets_soccer; // array of Bet structures called
bets.
                        // bets[0] will have each of the attributes
player,amount,completed, etc.

    struct Bet_golf {
        address player;
        uint amount;
        string winner; // name of the golfer that will win the
tournament
        uint256 id_torneo;
        bool completed;
    }

    Bet_golf[] public bets_golf;

    mapping (uint256 => address) public betToOwner;
    mapping (address => uint256) ownerBetsCount;
    // Variables de estado
    address private oracle;
    bytes32 private jobId_resultados_soccer;
    bytes32 private jobId_resultados_golf;
    uint256 private fee; //me tengo que enterar de cual es la tarifa

    // Event to record the oracle's answer (I won't need it, it
seems).
    event OracleResponse(bytes32 indexed requestId, string indexed
winner);

    constructor(address chainlinkToken, address chainlinkOracle,
uint256 tarifa) {
        setChainlinkToken(chainlinkToken);
        setChainlinkOracle(chainlinkOracle);
        oracle = chainlinkOracle;
        fee = tarifa; //mirar si definir fee en Link o en Juels -->
Juels
        // uint256 public fee = 100000000000000000; // tarifa de 0.1
LINK
    }

    //function that the user calls to create the bet
    //amount and balance must be in the same unit (balance returns the
result in Wei, so it must be divided by 10^18 to obtain it in Ethers).
```

```
    function placeBet(string memory _sport, uint amount, string memory
_winner, string memory _equipo_local, string memory _equipo_visitante,
uint8 jornada, uint256 id_torneo) public payable {
        address player = msg.sender;
        // All the requires at the beginning to make sure the state of
the variebles doesn´t change until all the conditions are met.
        require(amount > 0, "The bet amount must be greater than 0.");
        require(address(msg.sender).balance >= amount, "The player
does not have enough funds.");
        balances[msg.sender] = address(msg.sender).balance; //cojo el
balance de aquel que llama a la funcion (en wei)
        // first I have to take the money from the bettor before it is
realized.
        require(msg.value == amount, "The value sent does not match
the bet amount."); // I check that the money has been sent to the
contract.
        balances[player] -= amount;
        if (keccak256(abi.encode(_sport)) ==
keccak256(abi.encode("soccer")))  {
            Bet_soccer memory bet = Bet_soccer(msg.sender, amount,
_winner, jornada, _equipo_local, _equipo_visitante,false);
            bets_soccer.push(bet);
            uint betId = bets_soccer.length - 1; // returns the length
of bets -1 --> It increases
            betToOwner[betId] = msg.sender; // address that has called
the function
            ownerBetsCount[msg.sender] =
ownerBetsCount[msg.sender].add(1); // I use safemath's add function to
check for overflows
            requestOracleDataResultado(_sport, betId);
        }
        if (keccak256(abi.encode(_sport)) ==
keccak256(abi.encode("golf")))  {
            Bet_golf memory bet = Bet_golf(player, amount, _winner,
id_torneo, false);
            bets_golf.push(bet);
            uint betId = bets_golf.length - 1; // returns the length
of bets -1 --> It increases
            betToOwner[betId] = msg.sender; // address that has called
the function
            ownerBetsCount[msg.sender] =
ownerBetsCount[msg.sender].add(1); // I use safemath's add function to
check for overflows
            requestOracleDataResultado(_sport, betId);
        }


    }

    // function that is called when the bet has been created
    function requestOracleDataResultado(string memory _sport, uint
betId) private {
        if (keccak256(abi.encode(_sport)) ==
keccak256(abi.encode("soccer")))  {
          Bet_soccer memory apuesta = bets_soccer[betId];
```

```
            Chainlink.Request memory request =
buildChainlinkRequest(jobId_resultados_soccer, address(this),
this.fulfillOracleRequest.selector);
            request.add("sport", "soccer");
            request.add("betId", Strings.toString(betId));
            request.add("Equipo Local", apuesta.equipo_local);
            request.add("Equipo Visitante", apuesta.equipo_visitante);
            request.add("Jornada", Strings.toString(apuesta.jornada));
// Enviar la solicitud al oráculo
            sendChainlinkRequestTo(oracle, request, fee);
        }

        if (keccak256(abi.encode(_sport)) ==
keccak256(abi.encode("golf")))  {
            Bet_golf memory apuesta = bets_golf[betId];
            Chainlink.Request memory request =
buildChainlinkRequest(jobId_resultados_golf, address(this),
this.fulfillOracleRequest.selector);
            request.add("sport", "golf");
            request.add("betId", Strings.toString(betId));
            request.add("id_torneo",
Strings.toString(apuesta.id_torneo));
            sendChainlinkRequestTo(oracle, request, fee);
        }
    }

    // Function for processing the oracle response
    //receives the response from the Chainlink oracle network and
updates the state variables with the retrieved results
    // the answer of the oracle must include the winner and the betId.
    function fulfillOracleRequest(bytes32 _requestId, string memory
_winner, string memory _sport, uint betId) public
recordChainlinkFulfillment(_requestId){
        // Actualizar la variable de estado con el resultado recibido
        if (keccak256(abi.encode(_sport)) ==
keccak256(abi.encode("soccer")))  {
            Bet_soccer memory apuesta = bets_soccer[betId];
            apuesta.completed = true;
            if (keccak256(abi.encode(apuesta.winner)) ==
keccak256(abi.encode(_winner))) {
                // call the function that performs the calculation of
the odds
                calcular_coutas(_sport, betId);
            }
            emit OracleResponse(_requestId, _winner);

        }

        if (keccak256(abi.encode(_sport)) ==
keccak256(abi.encode("golf")))  {
            Bet_golf memory apuesta = bets_golf[betId];
            apuesta.completed = true;
            //the oracle returns the winner of the tournament
            if (keccak256(abi.encode(apuesta.winner)) ==
keccak256(abi.encode(_winner))) {
                //call the function that performs the calculation of
the odds
```

```
                calcular_coutas(_sport, betId);
            }
            emit OracleResponse(_requestId, _winner);

        }

    }


    function calcular_coutas(string memory _sport, uint256 betId)
private {
        if (keccak256(abi.encode(_sport)) ==
keccak256(abi.encode("soccer")))  {
            uint256 apuestas_al_partido = 0;
            uint256 mismas_apuestas = 0;
            uint256 cuota = 0;
            uint256 cantidad_a_pagar=0;

            for (uint i = 1; i <= bets_soccer.length; i++) { // to see
if it is the same match I have to see if the teams and the date
coincide
                if (bets_soccer[i].jornada ==
bets_soccer[betId].jornada) {
                    if
(keccak256(abi.encode(bets_soccer[i].equipo_local)) ==
keccak256(abi.encode(bets_soccer[betId].equipo_local))) {
                        apuestas_al_partido++; //so I can see how many
of the same games I have. Now I have to see how many of those have
been bet on the same thing.
                        if
(keccak256(abi.encode(bets_soccer[i].winner)) ==
keccak256(abi.encode(bets_soccer[betId].winner))) {
                            mismas_apuestas++;
                        }
                    }
                }
            }
            //odds calculation
            cuota = apuestas_al_partido/mismas_apuestas;
            cantidad_a_pagar = cuota*bets_soccer[betId].amount;
            // call the function that makes the payment
            realizar_pago(payable(bets_soccer[betId].player),
cantidad_a_pagar);

        }
        if (keccak256(abi.encode(_sport)) ==
keccak256(abi.encode("golf")))  {
            uint256 apuestas_al_torneo = 0;
            uint256 mismas_apuestas = 0;
            uint256 cuota = 0;
            uint256 cantidad_a_pagar=0;

            for (uint i = 1; i <= bets_golf.length; i++) {
                if (bets_golf[i].id_torneo ==
bets_golf[betId].id_torneo) {
                    apuestas_al_torneo++;
```

```
                    if (keccak256(abi.encode(bets_golf[i].winner)) ==
keccak256(abi.encode(bets_golf[betId].winner))) {
                        mismas_apuestas++;
                        }
                    }
                }

            cuota = apuestas_al_torneo/mismas_apuestas;
            cantidad_a_pagar = cuota*bets_golf[betId].amount;

            realizar_pago(payable(bets_golf[betId].player),
cantidad_a_pagar);
        }
    }

    function realizar_pago(address payable destino, uint256 cantidad)
private {
        require(address(this).balance >= cantidad, "Insufficient
balance");
        destino.transfer(cantidad);
    }

    //only the owner should be able to set this
    //There will be a JOB to achieve the results.
    function setJobId_resultados_soccer(string memory
_jobId_resultados) public onlyOwner {
        jobId_resultados_soccer = stringToBytes32(_jobId_resultados);
    }

    function setJobId_resultados_golf(string memory _jobId_resultados)
public onlyOwner {
        jobId_resultados_golf = stringToBytes32(_jobId_resultados);
    }

    // Comments "onlyOwner":
    // With this, the setJobId_results function may only be called by
the.
    // account that deployed the contract, since that account will be
the owner
    // of the contract due to inheriting Ownable.

    //I dont think it will be needed
    function setFee(uint256 _fee) public onlyOwner {
        fee = _fee;
    }

    function getOracle() public view returns (address) {
        return oracle;
    }

    function getFee() public view returns (uint256) {
        return fee;
    }

    function getJobId_resultados_golf() public view returns (bytes32)
{
        return jobId_resultados_golf;
```

```
    }

    function getJobId_resultados_soccer() public view returns
(bytes32) {
        return jobId_resultados_soccer;
    }

    function stringToBytes32(
        string memory source
    ) private pure returns (bytes32 result) {
        bytes memory tempEmptyStringTest = bytes(source);
        if (tempEmptyStringTest.length == 0) {
            return 0x0;
        }

        assembly {
            // solhint-disable-line no-inline-assembly
            result := mload(add(source, 32))
        }
    }
}
```

Especificación del Job fútbol nodo Chainlink

```
type = "directrequest"
schemaVersion = 1
name = "Soccer"
externalJobID = "55df1321-0531-432c-883e-bbc890ca924c"
forwardingAllowed = false
maxTaskDuration = "5h0m0s"
contractAddress = "0xcd8632E8eE6414e317F8b602Da974622439c23ce"
minContractPaymentLinkJuels = "100000000000000000"
observationSource = """
    decode_log   [type="ethabidecodelog"
                  abi="OracleRequest(bytes32 indexed specId, address
requester, bytes32 requestId, uint256 payment, address callbackAddr,
bytes4 callbackFunctionId, uint256 cancelExpiration, uint256
dataVersion, bytes data)"
                  data="$(jobRun.logData)"
                  topics="$(jobRun.logTopics)"]

    decode_cbor  [type="cborparse" data="$(decode_log.data)"]
    fetch        [type="http" method=GET
url="http://127.0.0.1:8080/endpoint?EquipoLocal=$(decode_cbor.equipo_l
ocal)&EquipoVisitante=$(decode_cbor.equipo_visitante)&Jornada=$(decode
_cbor.jornada)" allowUnrestrictedNetworkAccess="true"]
    parse        [type="jsonparse" path="$(decode_cbor.path)"
data="$(fetch)"]


    encode_data  [type="ethabiencode" abi="(bytes32 _requestId, string
_winner, string _sport, uint256 betId)" data="{ \\"_requestId\\":
```

```
$(decode_log.requestId), \\"_winner\\": $(fetch), \\"_sport\\":
$(decode_cbor.sport), \\"betId\\": $(decode_cbor.betId) }"]
    encode_tx    [type="ethabiencode"
                  abi="fulfillOracleRequest(bytes32 requestId, uint256
payment, address callbackAddress, bytes4 callbackFunctionId, uint256
expiration, bytes calldata data)"
                  data="{\\"requestId\\": $(decode_log.requestId),
\\"payment\\":   $(decode_log.payment), \\"callbackAddress\\":
$(decode_log.callbackAddr), \\"callbackFunctionId\\":
$(decode_log.callbackFunctionId), \\"expiration\\":
$(decode_log.cancelExpiration), \\"data\\": $(encode_data)}"
                  ]
    submit_tx    [type="ethtx"
to="0xcd8632E8eE6414e317F8b602Da974622439c23ce" data="$(encode_tx)"]

    decode_log -> decode_cbor -> fetch -> parse  -> encode_data ->
encode_tx -> submit_tx
"""
```

Especificación del Job golf nodo Chainlink

```
type = "directrequest"
schemaVersion = 1
name = "Golf"
externalJobID = "68764f40-cba2-44fc-b507-b1883bd76050"
forwardingAllowed = false
contractAddress = "0xcd8632E8eE6414e317F8b602Da974622439c23ce"
maxTaskDuration = "120h0m0s"
minContractPaymentLinkJuels = "100000000000000000"
observationSource = """
    decode_log   [type="ethabidecodelog"
                  abi="OracleRequest(bytes32 indexed specId, address
requester, bytes32 requestId, uint256 payment, address callbackAddr,
bytes4 callbackFunctionId, uint256 cancelExpiration, uint256
dataVersion, bytes data)"
                  data="$(jobRun.logData)"
                  topics="$(jobRun.logTopics)"]

    decode_cbor  [type="cborparse" data="$(decode_log.data)"]
    fetch        [type="http" method=GET
url="http://127.0.0.1:8081/golf?id_torneo=$(decode_cbor.id_torneo)"
allowUnrestrictedNetworkAccess="true"]
    parse        [type="jsonparse" path="$(decode_cbor.path)"
data="$(fetch)"]


    encode_data  [type="ethabiencode" abi="(bytes32 requestId, string
_winner, string _sport, uint256 betId)" data="{ \\"requestId\\":
$(decode_log.requestId), \\"_winner\\": $(fetch), \\"_sport\\":
$(decode_log.sport), \\"betId\\": $(decode_cbor.betId) }"]
    encode_tx    [type="ethabiencode"
                  abi="fulfillOracleRequest(bytes32 requestId, uint256
payment, address callbackAddress, bytes4 callbackFunctionId, uint256
expiration, bytes calldata data)"
```

```
                    data="{\\"requestId\\": $(decode_log.requestId),
\\"payment\\":   $(decode_log.payment), \\"callbackAddress\\":
$(decode_log.callbackAddr), \\"callbackFunctionId\\":
$(decode_log.callbackFunctionId), \\"expiration\\":
$(decode_log.cancelExpiration), \\"data\\": $(encode_data)}"
                    ]
    submit_tx    [type="ethtx"
to="0xcd8632E8eE6414e317F8b602Da974622439c23ce " data="$(encode_tx)"]

    decode_log -> decode_cbor -> fetch -> parse  -> encode_data ->
encode_tx -> submit_tx
"""
```

Código Servidor Intermedio fútbol

```
import json

from flask import Flask, jsonify, request
import requests

"""
Tras ejecutar el servidor ahora está corriendo y se puede acceder al
endpoint
haciendo una petición POST a la dirección
http://localhost:8080/endpoint
"""
# Inicializa una lista vacía para almacenar los datos recibidos


app = Flask(__name__)

@app.route('/endpoint', methods=['GET'])

def endpoint(): #solo se activa cuando recibe una solicitud HTTP GET
    data = []
    final = False
    # Obtiene los datos del formulario enviado en la solicitud GET

    """
    EJEMPLO DE PAYLOAD:
    payload = [('EquipoLocal')'CA Osasuna', (Jornada) '33',
(EquipoVisitante) 'Real Sociedad de Fútbol']
    """
    form_data = request.args

    # Agrega los datos a la lista
    data.append(dict(form_data))
    file = open("parametros.txt", "w")
    file.write(str(data))

    #Obtengo los datos de los parámetros enviados en el GET:
    local = data[0]['EquipoLocal']
    jornada = data[0]['Jornada']
    visitante = data[0]['EquipoVisitante']
```

```
    # Obtener la apikey del header de la petición

    while not final:
        api_key = "35f1f991f8a047a5b2197eb2af6fc"
        url =
"https://api.sportsdata.io/v4/soccer/scores/json/CompetitionDetails/4?
key=" + api_key
        # Hacer la petición al servicio externo
        response = requests.get(url)

        # Obtener la información de la respuesta y parsearla
        parsed_data = json.loads(response.text)

        for x in range(len(parsed_data["Games"])):

            if parsed_data["Games"][x]["Week"] == int(jornada):
                if local in parsed_data["Games"][x]["HomeTeamName"]
and visitante in parsed_data["Games"][x]["AwayTeamName"]:
                    if parsed_data["Games"][x]["Status"] == "Final":
                        goles_local =
parsed_data["Games"][x]["HomeTeamScorePeriod1"] +
parsed_data["Games"][x]["HomeTeamScorePeriod2"]
                        goles_visitante =
parsed_data["Games"][x]["AwayTeamScorePeriod1"] +
parsed_data["Games"][x]["AwayTeamScorePeriod2"]
                        final = True

    if goles_local == goles_visitante:
        resultado = 0
    elif goles_local > goles_visitante:
        resultado = 1
    else:
        resultado = 2

    # Devolver la respuesta al oráculo
    return str(resultado)

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8080)
```

Código Servidor Intermedio golf

```
import json
import time
from flask import Flask, jsonify, request
import requests


app = Flask(__name__)

@app.route('/golf', methods=['GET'])
```

```python
def endpoint(): #solo se activa cuando recibe una solicitud HTTP GET
    data = []
    final = False
    # Obtiene los datos del formulario enviado en la solicitud GET

    """
    EJEMPLO DE PAYLOAD:
    payload = [(id_torneo)'553', (Jugador) 'Jon Rahm']
    """
    form_data = request.args

    # Agrega los datos a la lista
    data.append(dict(form_data))
    file = open("parametros_golf.txt", "w")
    file.write(str(data))

    #Obtengo los datos de los parámetros enviados en el POST:
    id_torneo = data[0]['id_torneo']
    jugador = data[0]['jugador']

    # Obtener la apikey del header de la petición

    while not final:
        api_key = "626d289c5f5e4cd18c2daee46eaf418c"
        url = "https://api.sportsdata.io/golf/v2/json/Leaderboard/" +
id_torneo + "?key=" + api_key
        # Hacer la petición al servicio externo
        response = requests.get(url)

        # Obtener la información de la respuesta y parsearla
        parsed_data = json.loads(response.text)
        if parsed_data["Tournament"]["IsOver"]:
            if parsed_data["Players"][0]["Name"] == jugador:
                resultado = 1
            else:
                resultado = 0
            final = True
        time.sleep(60)

    # Devolver la respuesta al oráculo
    return str(resultado)

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8081)
```

Código Tests unitarios

```javascript
const MyContract = artifacts.require('SportsDataOracle.sol');

contract('SportsDataOracle', (accounts) => {
  let myContractInstance;
```

```
  before(async () => {
    const chainlinkToken = accounts[2]; // Address to be used as
chainlinkToken
    const chainlinkOracle = accounts[3]; // Address to be used as
chainlinkOracle
    const tarifa = 10000000000; // Convert ether to wei
    myContractInstance = await
MyContract.new(chainlinkToken,chainlinkOracle,tarifa);
  });

  it('should set the initial values in the constructor', async () => {
    // Retrieve the values set in the constructor
    const oracle = await myContractInstance.getOracle();
    const fee = await myContractInstance.getFee();

    // Assert that the retrieved values match the expected values
    assert.equal(oracle, accounts[3], 'The oracle address is
incorrect');
    assert.equal(fee, 10000000000, 'The fee is incorrect');
  });

  it('should place a bet for soccer', async () => {
    const sport = 'soccer';
    const amount = 100;
    const winner = '1';
    const equipoLocal = 'Real Madrid';
    const equipoVisitante = 'Athletic Club';
    const jornada = 38;

    // Call the placeBet function
    await myContractInstance.placeBet(sport, amount, winner,
equipoLocal, equipoVisitante, jornada,0,  {from: accounts[0], value:
100000000000000 });

    // Assert the state of the contract
    // Check if the bet was stored correctly in the bets_soccer array
    const bet = await myContractInstance.bets_soccer(0);
    assert.equal(bet.amount, amount, 'The bet amount is incorrect');
    assert.equal(bet.winner, winner, 'The bet winner is incorrect');
    assert.equal(bet.equipo_local, equipoLocal, 'The local team is
incorrect');
    assert.equal(bet.equipo_visitante, equipoVisitante, 'The visitor
team is incorrect');
    assert.equal(bet.jornada, jornada, 'The gameweek is incorrect');
  });

  it('should place a bet for golf', async () => {
    const sport = 'golf';
    const amount = 200;
    const winner = 'Rickie Fowler';
    const id_torneo = 53;

    // Call the placeBet function
    await myContractInstance.placeBet(sport, amount, winner, '', '',
10, id_torneo, { from: accounts[1], value: amount });
```

```
    // Assert the state of the contract
    // Check if the bet was stored correctly in the bets_golf array
    const bet = await myContractInstance.bets_golf(0);
    assert.equal(bet.amount, amount, 'The bet amount is incorrect');
    assert.equal(bet.winner, winner, 'The bet winner is incorrect');
    assert.equal(bet.id_torneo, id_torneo, 'The id_torneo is
incorrect');
  });

  it('should set job to retrieve golf results', async () => {
    const newJobId =
web3.utils.utf8ToHex('7223acbd01654282865b678924126013');
    // Call the setJobId_resultados_golf function
    await myContractInstance.setJobId_resultados_golf(newJobId, {
from: accounts[0] });

    // Retrieve the stored jobId_resultados_golf from the contract
    const storedJobId = await
myContractInstance.getJobId_resultados_golf();

    // Assert that the stored jobId_resultados_golf matches the
expected value
    assert.equal(storedJobId, newJobId, 'The stored
jobId_resultados_golf is incorrect');
  });

  it('should set job to retrieve soccer results', async () => {
    const newJobId =
web3.utils.utf8ToHex('7223acbd01654282865b678924126099');
    // Call the setJobId_resultados_golf function
    await myContractInstance.setJobId_resultados_soccer(newJobId, {
from: accounts[0] });

    // Retrieve the stored jobId_resultados_golf from the contract
    const storedJobId = await
myContractInstance.getJobId_resultados_soccer();

    // Assert that the stored jobId_resultados_golf matches the
expected value
    assert.equal(storedJobId, newJobId, 'The stored
jobId_resultados_golf is incorrect');
  });

  it('should check if the bettor wins', async () => {
    const sport = 'golf';
    const amount = 200;
    const winner = 'Rickie Fowler';
    const id_torneo = 53
    const bet = await myContractInstance.bets_golf(10);

    const requestId =
web3.utils.utf8ToHex('7223acbd01654282865b678924126013');
    // Call the setJobId_resultados_golf function
    const winner2 = await
myContractInstance.responseOracleRequest(requestId, 'Rickie Fowler',
'golf', 10, { from: accounts[4] });
```

```
    // Assert that the stored jobId_resultados_golf matches the
expected value
    assert.equal(winner, winner2, 'The winner retrieved by the oracle
does not match the one indicated in the bet. WE HAVE A LOSER!');
  });
});
```

Bibliografía

Referencias Bibliográficas

[1] *¿Qué es una cadena de bloques?* (s.d.). Recuperado del sitio web de Coinbase: https://www.coinbase.com/es-LA/learn/crypto-basics/what-is-a-blockchain#:~:text=La%20cadena%20de%20bloques%20de%20Ethereum%20es%20un%20paso%20m%C3%A1s,solo%20para%20administrar%20dinero%20digital.

[2] *¿Qué es una cadena de bloques?* (s.d.). Recuperado del sitio web de Criptonoticias: https://www.criptonoticias.com/criptopedia-old/blockchain-bloques-transacciones-firmas-digitales-hashes/

[3] Wackerow, P. (2022). *Bloques*. Recuperado de: https://ethereum.org/es/developers/docs/blocks/

[4] Smith, C. (2023). *Proof of Work*. Recuperado de: https://ethereum.org/es/developers/docs/consensus-mechanisms/pow/

[5] Wackerow, P. (2022). *Introduction to Smart Contracts*. Recuperado de: https://ethereum.org/es/developers/docs/smart-contracts/

[6] Wackerow, P. (2022). *Ethereum Virtual Machine*. Recuperado de: https://ethereum.org/es/developers/docs/evm/

[7] *EVM – Ethereum Virtual Machine.* (s.d). Recuperado del sitio web de diariobitcoin: https://www.diariobitcoin.com/glossary/evm-ethereum-virtual-machine-maquina-virtual-de-ethereum/#:~:text=EVM%20(Ethereum%20Virtual%20Machine)%20es,objeto%20computacional)%20de%20forma%20arbitraria.

[8] *¿Qué es la Ethereum Virtual Machine (EVM)?* (2023). Recuperado de: https://academy.bit2me.com/que-es-ethereum-virtual-machine-evm/

[9] Smith, C. (2023). *Gas and Fees*. Recuperado de: https://ethereum.org/es/developers/docs/gas/#:~:text=%C2%BFQu%C3%A9%20es%20el%20gas%3F,cada%20transacci%C3%B3n%20requiere%20una%20comisi%C3%B3n

[10] *¿Podría utilizarse la tecnología Blockchain en las apuestas deportivas?* (2022). Recuperado de:

https://www.digitalgamingpe.com/2022/08/05/podria-utilizarse-la-tecnologia-blockchain-en-las-apuestas-deportivas/

[11] *Why sports betting is shifting to blockchain.* (s.d.). Recuperado del sitio web de Goracle: https://www.goracle.io/post/why-sports-betting-is-shifting-to-blockchain

[12] Jones, Z. (2021). *How Blockchain Betting Technology Is Bringing The Power Back To The Bettors. Forbes.*

[13] *Algorand, the future of sports betting.* (s.d.). Recuperado del sitio web de Goracle: https://www.goracle.io/post/algorand-the-future-of-sports-betting

[14] *A beginner's guide to the benefits of decentralized gambling using smart contracts*. Recuperado de: https://www.bsc.news/post/a-beginners-guide-to-the-benefits-of-decentralized-gambling-using-smart-contracts

[15] *Tipos de datos 1*. (s.d.). Recuperado de: https://aprendeblockchain.wordpress.com/desarrollo-en-ethereum/tipos-de-datos-i/

[16] Wackerow, P. (2022). *Ethereum Accounts.* Recuperado de: https://ethereum.org/es/developers/docs/accounts/#:~:text=Una%20clave%20privada%20consta%20de,puede%20cifrar%20con%20una%20contrase%C3%B1a.&text=La%20clave%20p%C3%BAblica%20se%20genera,firma%20digital%20de%20curva%20el%C3%ADptica.

[17] Sant, H. (2023). *¿Qué es MetaMask y cómo usarlo? Recuperado del sitio web de GeekFlare:* https://geekflare.com/es/finance/beginners-guide-to-metamask/

[18] Mastando, M. (2022). *¿Qué son los oráculos de blockchain?* Forbes.

[19] Díaz, N. (2023). ¿Qué son los oráculos blockchain?. Recuperado de: https://academy.bit2me.com/que-es-oraculos-blockchain/

[20] López, A. (2022). ¿Qué es Chainlink? Recuperado de: https://cryptoconexion.com/chainlink/

[21] Smith, C. (2023). *Deploying Smart Contracts*. Recuperado de: https://ethereum.org/es/developers/docs/smart-contracts/deploying/#:~:text=Necesitas%20implementar%20tu%20contrato%20inteligente,recopilado%20sin%20especificar%20ning%C3%BAn%20destinatario

[22] Smith, C. (2023). Nodes and clients. Recuperado de: https://ethereum.org/en/developers/docs/nodes-and-clients/

[23] Pascual, J. L. (2023). *¿Qué es MetaMask? La forma más fácil de usar dApps*. Recuperado de: https://academy.bit2me.com/que-es-metamask-la-forma-mas-facil-de-usar-dapps/

[24] *SportsData launches live Chainlink node giving smart contracts access to premium sports data. (s.d.).* Recuperado del sitio web de SportsData: https://sportsdata.io/sportsdataio-launches-live-chainlink-node-giving-smart-contracts-access-to-premium-sports-data

[25] Smith, C. (2023). *Introduction to dApps.* Recuperado de: https://ethereum.org/en/developers/docs/dapps/

[26] *What Are Decentralized Apps?* (2021). Recuperado del sitio web de Cryptopedia: https://www.gemini.com/cryptopedia/decentralized-applications-defi-dapps#section-decentralized-app-criteria

[27] *Existing Job Request. (s.d.).* Recuperado del sitio web de Chainlink: https://docs.chain.link/any-api/get-request/examples/existing-job-request/

[28] *Running a Chainlink node. (s.d.).* Recuperado del sitio web de Chainlink: https://docs.chain.link/chainlink-nodes/v1/running-a-chainlink-node

[29] Abiodun, M. (2023). *¿Cómo pueden los desarrolladores recuperar ETH de la red de pruebas Sepolia de Ethereum?* Recuperado de: https://www.cryptopolitan.com/es/como-los-desarrolladores-recuperan-eth-sepolia-testnet/#What_is_the_Sepolia_Testnet

[30] Gerardi, R. (2022). *A beginner´s guide to tmux.* Recuperado de: https://www.redhat.com/sysadmin/introduction-tmux-linux

[31] *Tmux: como funciona el multiplexor de terminal*. (2023). Recuperado del sitio web de IONOS: https://www.ionos.es/digitalguide/servidores/know-how/tmux-multiplexor-de-terminal/

[32] *Write Solidity tests*. Recuperado del sitio web de Truffle Suite: https://trufflesuite.com/docs/truffle/how-to/debug-test/write-tests-in-solidity/