ORÁCULOS EN BLOCKCHAIN

IGNACIO SILVA CARRASCO

GRADO EN INGENIERÍA INFORMÁTICA. FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin de Grado en Ingeniería Informática

Curso

2022/2023

Director:

Jesús Correas Fernández

Autorización de difusión

Ignacio Silva Carrasco

Curso 2022/2023

El/la abajo firmante, matriculado/a en el Grado en Ingeniería Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Grado: "Oráculos en blockchain", realizado durante el curso académico 2022-2023 bajo la dirección deJesús Correas Fernández en el Departamento de Sistemas Informáticos y Computación, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Índice general

Capítulo 1

Introducción

Debido a la naturaleza de los sitemas blockchain, en la que toda la información ha de ser verificable y reproducible para mantener una red basada en la confianza entre nodos, introducir información proveniente de sistemas externos no es trivial. Esta información puede ser falsa ya sea porque sea extraída de una fuente comprometida o bien con fines maliciosos, es decir, que el dueño quiera realizar una manipulación dentro de la blockchain. Además, como esta información es externa, no es verificable dentro de la propia red, por lo tanto es información en la que no se puede confiar. Para resolver esta problemática se presenta la idea de los "Oráculos", piezas de código diseñadas para la interconexión de redes blockchain con el mundo exterior con: aplicaciones, scripts, bases de datos o incluso otras blockchain.

Este trabajo de fin de grado tiene como objetivo pruncipal de realizar un estudio en profundidad de estos Oráculos y su funcionamiento teniendo en mente unos objetivos específicos:

1. Estado del arte: averiguar el estado actual del desarrollo de oráculos por diferentes empresas dedicadas al mundo de las redes blockchain, su desarrollo y, en el caso de poder ser capaces de utilizar esta tecnología, realizar un breve análisis del funcionamiento de estos. Además de empresas, también se pretende contemplar las propuestas académicas existentes.

2. Taxonomía: entender la composición de los oráculos a través de diferentes artículos y fuentes de información disponibles, de tal forma que se puedan clasificar los oráculos dependiendo de varias características como:

    - La fuente los datos: si los datos son procedentes de fuentes software, hardware o humanas.
    - El modelo de confianza propuesto: un modelo centralizado en el que sólo existe un oráculo o un modelo descentralizado en el que existe una red de oráculos que han de llegar a un consenso.
    - El patrón de diseño: se refiere a el modelo elegido a la hora de que exista una interacción entre los clientes y contratos con el oráculo, dando lugar principalmente a tres modelos: *Immediate-Read, Publish-Subscribe* o *Request-response*.

- La interacción entre los sistemas: se refiere al flujo de los datos entre el sistema blockchain y el sistema externo, es decir, si los datos son envíados desde la blockchain hacia el exterior o de forma contraria.

3. Desarrollar prototipos: realizar un pequeño desarrollo de algunos modelos de oráculos en una red Ethereum con el fin de entender: su funcionamiento, la seguridad necesitada por el código de estos contratos frente a diversos ataques existentes en blockchain, medidas de optimización y diferentes modelos económicos que se pueden llevar a cabo para rentabilizar los oráculos en sus diferentes diseños.

El plan de trabajo acordado con el tutor consiste en:

1. Reuniones semanales de una hora.

2. Primera fase de estudio en la que se pretende recopilar los diferentes artículos, libros, etc. Para poder realizar un análisis posterior acerca de los contenidos y desarrollar tanto la memoria como los prototipos.

3. Preparación del entorno de trabajo, tanto respecto al software: editor de código, entornos de desarrollo de Solidity, red blockchain local, entorno de pruebas, etc. Como preparación del equipo a utilizar: conexión remota del equipo, limpieza e instalación limpia de sistema operativo, etc.

4. Desarrollo de los prototipos con la información obtenida.

5. Estudio de los resultados hallados durante la ejecución y análisis de los prototipos creados.

6. Conclusiones finales del trabajo.

Capítulo 2

Introducción a la Blockchain y Oráculos

2.1. Contexto general

En las últimas décadas, la tecnología blockchain ha sido reconocida como una innovación de gran alcance con el potencial de transformar la forma en que se llevan a cabo las transacciones digitales y se gestionan los activos en línea.

En su núcleo, la blockchain es una estructura de datos descentralizada y segura, donde las transacciones que se realizan en su interior se agrupan en bloques verificables por todos los nodos que la forman. Estos bloques se encadenan mediante el uso de un hash criptográfico propio y el hash generado en el bloque anterior, lo que garantiza la integridad y la inmutabilidad de la información, ya que si se intenta modificar un dato de un bloque anterior se produciría un cambio drástico en el hash de todos los bloques posteriores de la cadena. Además del hash, también se guardan(*Figura 2.1*):

- El instante: en el que ese bloque ha sido"creado (*minado* en la terminología utilizada en los sistemas de blockchain)". Es decir, una marca del orden cronológico en el que la cadena crece despuéws de que un Nodo "minero" haya resuelto el problema estipulado.
- El nonce: usado en las blockchain de "prueba de trabajo". La idea es que el bloque genere un hash con unas características concretas, lo más común por ejemplo es un número X de ceros en la parte inicial o final del hash. El nonce es el número que junto con toda la demás información en el bloque hace que el hash de ese bloque cumpla las condiciones establecidas. El trabajo que se realiza para encontrar este nonce, que es un trabajo computacional iterativo es lo que le da nombre a esta prueba. Este problema escala a nivel de la potencia de computación, que mejora con los años, ya que exigir más ceros seguidos aumenta el problema de manera exponencial.
- La raíz del Merkle tree[1]: para poder localizar las transacciones contenidas en el bloque.

[1] https://en.wikipedia.org/wiki/Merkle_tree

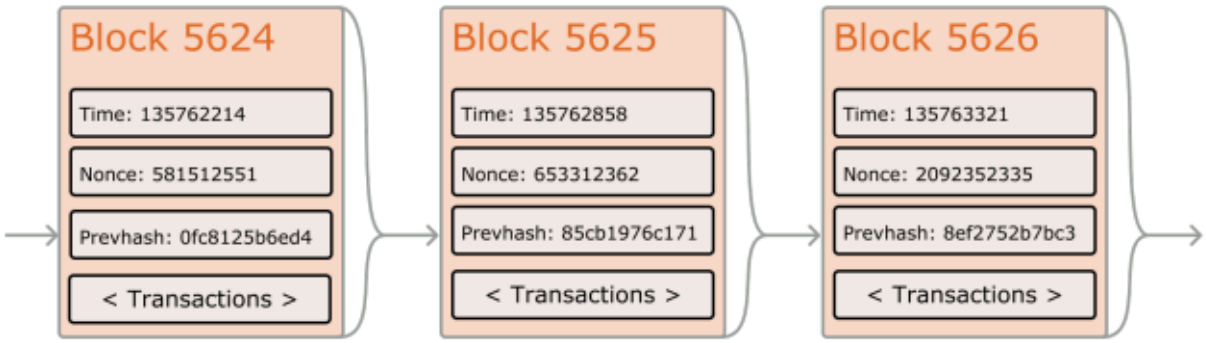

Figura 2.1: *Cadena de bloques [14]*

Además, con la llegada de Ethereum se creó un nuevo concepto: "Blockchain programable" [14]. La blockchain, aparte de contener información de transacciones, contiene un objeto llamado "contrato inteligente (*smart contract*)". Estos contratos pueden ser entendidos, al igual que en el mundo real, como un acuerdo entre dos partes. Pero a diferencia del contrato físico que no te asegura la ejecución de ninguna acción en particular, estos contratos están formados por código de un programa cuya ejecución asegura la aplicación en términos del contrato. Esto ha dado lugar a la posibilidad de creación de aplicaciones contenidas dentro de la blockchain en un verdadero entorno peer-to-peer evitando así intermediarios y con total transparencia.

Gracias a estas características, las blockchains han encontrado aplicaciones en campos tan diversos como las finanzas, la logística, la atención médica etc.

Sin embargo, a pesar de los beneficios que ofrece esta tecnología, una de sus principales limitaciones radica en su aislamiento del mundo real. La mayoría de los sistemas existentes son incapaces de acceder a datos y eventos externos sin comprometer su integridad y descentralización. Esto representa un obstáculo significativo para su adopción masiva y para la expansión de su potencial.

Para entender mejor el problema pongamos un ejemplo:

Alice y Bob hacen una apuesta de 10€ cada uno a un partido de fútbol. Alice apuesta al equipo A y Bob al equipo B. El dinero queda retenido en un smart contract. ¿Cómo sabría el smart contract cuando termina el partido a quién de los dos ha de pagarse el dinero?

Para poder tener esta información, el smart contract debe poder acceder a datos externos para obtener la información relacionada con la finalización y el resultado del partido y así poder recompensar al ganador de la apuesta.

Como solución al problema se presenta la idea de los "Oráculos [9]", pequeños sitemas tanto on-chain como off-chain que sirven como nexo de unión entre el mundo real y la blockchain o incluso como intermediarios entre blockchains diferentes y que proveen información a los smart contracts. Gracias a ellos el resto de contratos pueden utilizar información del mundo exterior para la toma de decisiones y desencadenar diferentes acciones.

2.2. El problema de los oráculos

Pese a dar una solución a la conexión con el exterior, los Oráculos causan que uno de los principios fundamentales de la blockchain se vea comprometido: la integridad de los datos. Estos han de ser inmutables, pero también comprobables y reproducibles. Esto es debido a que en una red distribuida todos los nodos han de llegar a un consenso sobre el estado de la cadena.

La reproducibilidad garantiza que todos los nodos puedan verificar que los datos en un bloque sean correctos para llegar a un acuerdo sobre su validez. Dicho de otra manera, si los datos no fueran reproducibles, los nodos no podrían estar seguros de si están llegando a un consenso sobre la misma información. Para asegurarse de que esto es así los nodos ejecutan de nuevo las transacciones de un mismo bloque y comprueban que el resultado sea el mismo, garantizando que no se producen manipulaciones por parte de ningún nodo malicioso. Si durante estas pruebas se realiza una llamada a un sistema externo que provee datos diferentes cada vez, una transacción podría dar resultados distintos y por tanto un nodo validador consideraría la transacción como incorrecta. Es por esto que los sistemas basados en blockchain la mayoría de veces prohíben las llamadas a sistemas externos. Esto es debido a que los sistemas blockchain se basan en el consenso de los nodos de la red, es decir, la mayoría de los nodos pertenecientes a la red tienen que llegar a un acuerdo para decidir cuál será el siguiente bloque que se añadrá al final de la cadena. Esto se realiza a través de un sistema de votación con unas normas comunes. Una llamada a un sistema exterior que cada vez da diferentes resultados puede afectar a este consenso. Algunos de los ataques que se podrían producir son:

- Ataques Sybil [3]: un usuario crea varias cuentas que intentan alterar el resultado del consenso mediante la validación de bloque que a ellos les interese.
- Ataques del 51 % [1]: El ataque se basa en conseguir que el 51 % de los nodos vote por el bloque que el nodo malicioso quiera. Si muchos nodos se pusieran de acuerdo para validar la misma transacción en un momento en el que saben que todos van a dar el mismo resultado, podrían alterar el resultado de la votación.

Aún así los contratos muchas veces necesitan esta información del mundo exterior para su funcionamiento y, al introducir datos del mundo real: datos de sensores, valor de monedas para sistemas financieros, controles meteorológicos para sistemas de seguros..., se crea el problema conocido como "garbage in garbage out". Este problema, aunque en parte relacionado, es diferente al de la falta de reproducibilidad y se basa en que no se puede confiar

directamente en la información de un medio ajeno a la blockchain ya que esta puede ser falsa, ya sea porque al proveedor de esta información le beneficie de alguna manera o porque ataques externos al proveedor de la información hayan comprometido la integridad de los datos. Por tanto los oráculos se transforman en contratos en los que no se puede confiar plenamente y, en un sistema basado en la confianza, se convierte en una de las mayores vulnerabilidades en la seguridad de ese sistema.

Los principales problemas que generan los oráculos en los sistemas blockchain se pueden resumir en:

- Al ser puntos centralizados de provisión de información no se puede confiar en ellos, o dicho de otra manera, como no funcionan por consenso, su nivel de confianza dentro de la red es menor. Esto es debido a que el oráculo funciona como único proveedor, la información que él provee no es algo que se pueda votar ya que, la mayoría de veces, no provee información que exista o se pueda generar dentro del sistema y por tanto es única, lo que la convierte en algo difícil de validar.
- En el caso en el que se confíe en ellos, nada asegura que la información que provean, aunque sea de manera involuntaria, sea falsa. Como se ha mencionado antes, la información que provee el oráculo no se puede asegurar que sea verdadera ya que, dependiendo de la fuente que utilice, por ejemplo una API pública, esta puede ser susceptible a ataques o bien el dueño de la fuente puede tener intereses maliciosos, es decir, querer manipular el resutlado de la votación.

Capítulo 3

Oráculos

En este capítulo se va a tratar tanto la taxonomía y diseño de oráculos, como el estado actual del desarrollo de oráculos por diferentes empresas e investigaciones o artículos académicos.

3.1. Taxonomía de Oráculos

En esta sección se describen los tipos de oráculo que existen en función de diferentes categorías de clasificación: según la fuente de datos, el modelo de confianza utilizado, el patrón de diseño del oráculo, y la interacción del oráculo con los sistemas que lo utilizan [12, 16, 20].

3.1.1. Fuente de datos

Según la fuente de datos podemos distinguir principalmente tres tipos de Oráculos:

- Software: en este tipo de oráculo se incluyen aquellos que acceden a fuentes de datos relacionadas con Internet: bases de datos, servidores, etc. Algunos ejemplos de este tipo de información podría ser: el precio de la moneda para contratos financieros recogido mediante uso de APIs de la bolsa o liquidaciones automatizadas dependiendo del precio...
- Hardware: en este tipo de oráculo se incluyen aquellos que acceden a fuentes que toman datos producidos por dispositivos físicos y los convierten en valores digitales como pueden ser sensores de temperatura, escáneres de códigos de barras, etc. Esto podría servir por ejemplo para la trazabilidad de un sistema de seguros que registre periódicamente la temperatura media de una nave industrial, de tal manera que la información se recoja de los sensores de temperatura.
- Humano: esta fuente quizás se la más poco común. Consiste en que humanos proveen a los contratos respuestas a preguntas, teniendo estos la responsabilidad de validar la información con la que se alimenta al contrato. Estos humanos están registrados

ya que hacen uso de funciones criptográficas, lo que hace que la intención de fraude disminuya y además no son solo capaces de contestar a preguntas deterministas, sino también a preguntas más complicadas para la máquina.

3.1.2. Modelo de confianza

Dependiendo del número de nodos de los cuales los oráculos tomen datos para los smart contracts, encontramos dos tipos de modelos de confianza:

- Centralizados (*Figura 3.1*) : suelen ser on-chain (dentro de la propia red blockchain). Este modelo tiene una gran desventaja señalada con anterioridad y es que se vuelve un punto único de ataque para la blockchain, pudiendo afectar a la disponibilidad, accesibilidad y la validez de los datos que provee. Además, si es on-chain puede darse el caso de que la clave privada del dueño o administrador del contrato se vea comprometida y por tanto un atacante pueda pasar a tener el 100 % del control sobre lo que sucede.[1]

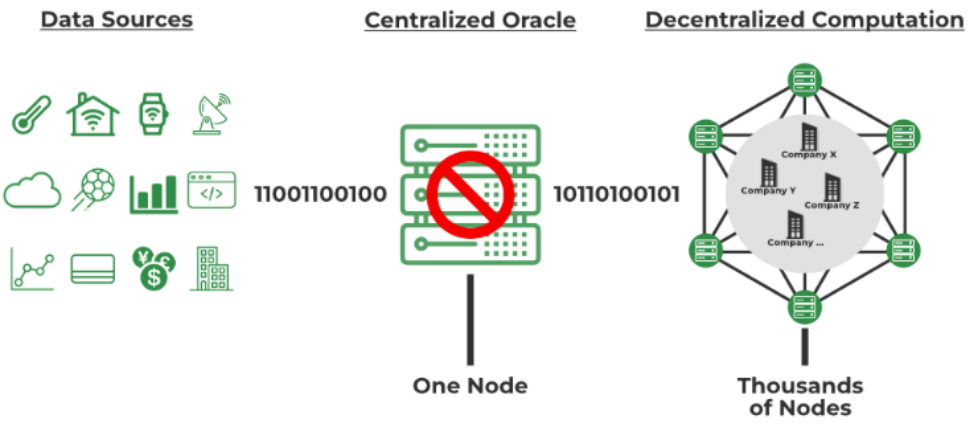

Figura 3.1: *Esquema de un oráculo centralizado[18]*

- Descentralizados (*Figura 3.2)*: este modelo resuelve el problema de ser un fácil punto de ataque, a costa de ser menos eficiente, haciendo que la latencia del sistema aumente. Esto se debe a que en este caso los oráculos recogen información de diversas fuentes. Esta información es sometida a consenso de tal manera que la red de oráculos decide qué información se va a considerar verdadera llegando a un consenso. Una manera sería que tomasen la media de todos los resultados. Una vez decidido, el resultado es envíado a la blockchain de la cual partió la solicitud de información.

[1] https://shorturl.at/zBET2

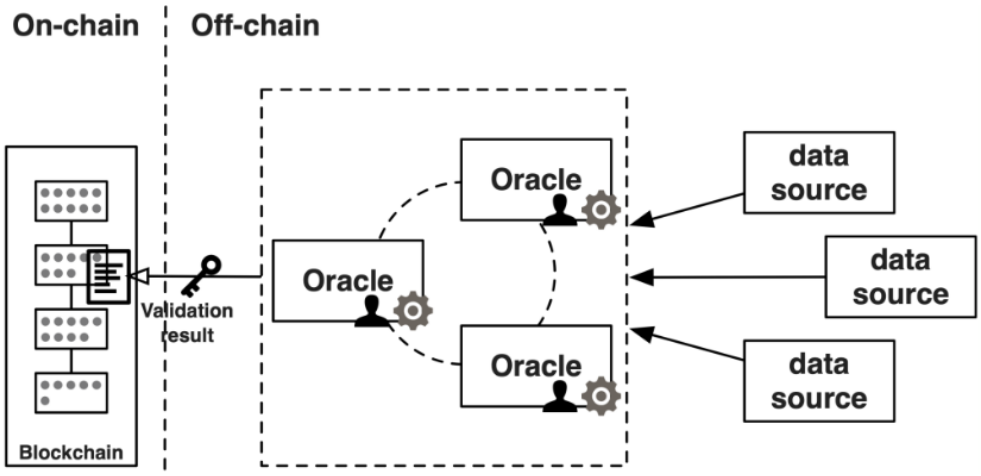

Figura 3.2: *Esquema de oráculo descentralizado [2]*

3.1.3. Patrón de diseño

Según el patrón de respuesta que siga un oráculo encontramos varios tipos, los más comunes son:

- Immediate-Read(*Figure 3.3)*: este tipo de contratos se utilizan para informaciones pequeñas que suele utilizarse para operaciones inmediatas y que pueden guardarse dentro de la parte de memoria permanente (storage) de un contrato. Esta información puede irse actualizando mediante transacciones al contrato. Una vez la información está guardada dentro del contrato, los clientes solo tienen que hacer una transacción de solicitud del dato al contrato para obtener la información.

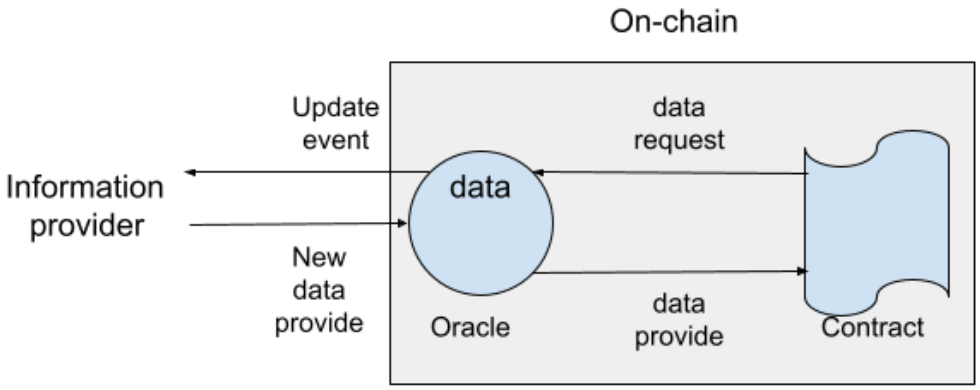

Figura 3.3: *Esquema del oráculo Immediate-Read*

- Publish-Subscribe: en este caso el oráculo provee un servicio de "broadcast"de la información con una frecuencia asignada a los llamados suscriptores, que son aquellos que

han pagado por el servicio. De esta manera los contratos suscritos reciben la información de manera periódica por ejemplo cada vez que se actualiza. Puede desarrollarse mediante un contrato on-chain que mantenga los suscriptores y la información en variables del contrato y sea el dueño el que decida cuándo actualizar la información, cuando eso suceda se realizará una transacción a los contratos subscritos con los nuevos datos (*Figura 3.4)*, o un sistema en el cual el cliente se "subscribe"mediante una solicitud al oráculo, que emite un evento con los datos del suscriptor y estos se guardan fuera de la blockchain. De tal manera que cuando se actualice la información ellos también la reciban. El funcionamiento sería parecido a una web RSS(*Figura 3.5)*.

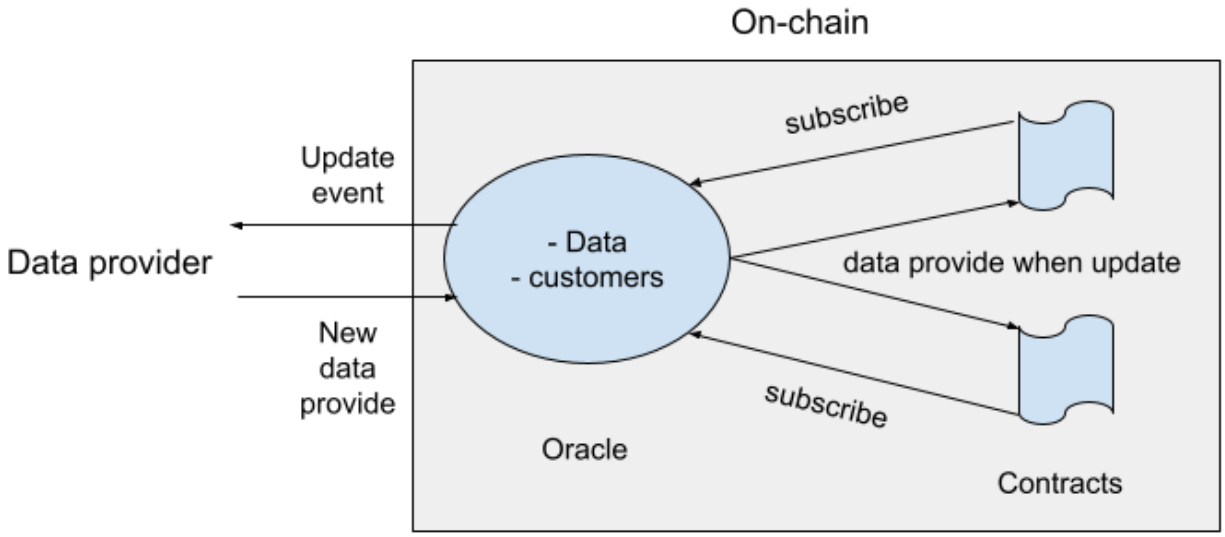

Figura 3.4: *Esquema del oráculo Publish-Subscribe on-chain*

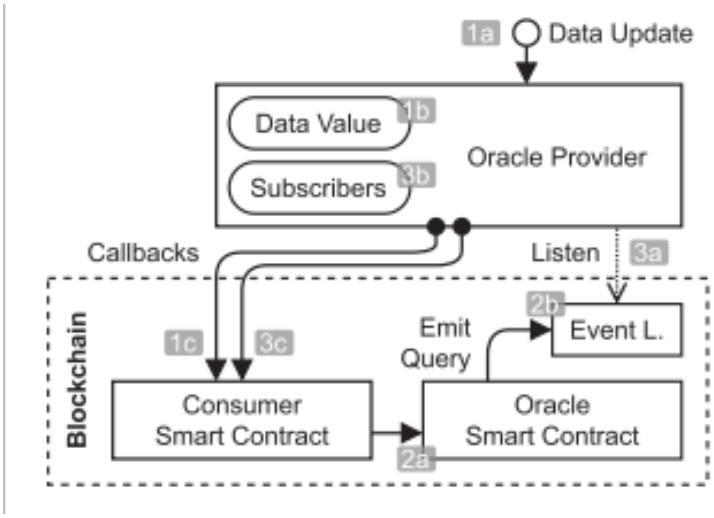

Figura 3.5: *Esquema del oráculo Publish-Subscribe off-chain [19]*

- Request-Response: este es el tipo de oráculo más complejo de implementar, esto es debido a que normalmente se desarrollan para datos demasiado grandes como para ser contenidos dentro de un contrato y por lo tanto han de ser almacenados fuera de la blockchain. Además el cliente suele querer solo una pequeña parte de los datos. Para implementar este patrón de diseño se utilizan dos componentes principales:

  - Componente on-chain: en forma de contrato, recoge la solicitud del cliente con los argumentos formando una consulta detallando que información específica se requiere. Una vez validad esta solicitud por el contrato oráculo, se mandará fuera de la blockchain en forma de evento o como cambio de estado de un contrato.
  - Componente off-chain: a parte del script de control de los eventos del oráculo, se utiliza una base de datos, aplicación, etc. A la que se le pasa la consulta. Los datos obtenidos son firmados por el dueño del oráculo para poder validar su confianza y así enviar de vuelta en forma de transacción la información obtenida.

  También cabe mencionar que debido a que son transacciones procesadas en el exterior, muchas veces como parámetros es necesario especificar qué permisos se tienen como cliente para acceso a los datos en los argumentos de la consulta.

  Podemos ver ejemplos de buenas y malas consultas en el artículo [13]

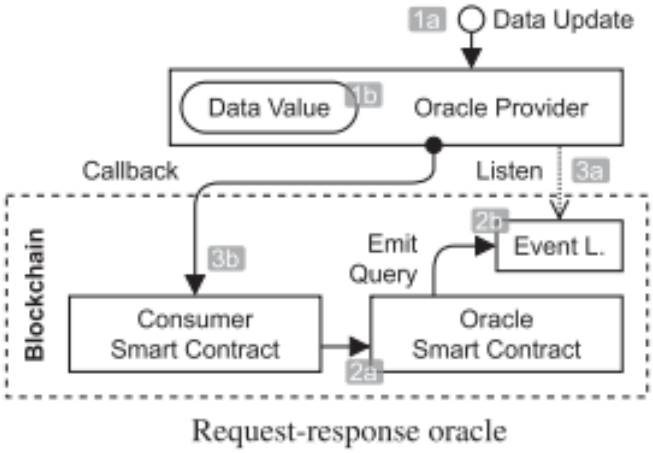

Figura 3.6: *Esquema del oráculo Request-Response [19]*

3.1.4. Interacción

Existen varias formas en las que los oráculos pueden interactuar con el mundo real dependiendo del flujo de origen de la información:

- Inbound: son aquellos oráculos que transmiten información del mundo exterior al interior de la blockchain. El ejemplo más claro son los contratos DEFI, es decir aquellos que

son dedicados al mundo de las finanzas y que necesitan valores del mercado actuales para la toma de decisiones.

- Outbound; son aquellos oráculos que transmiten información de la blockchain al mundo exterior, es decir que un evento que ocurre dentro de la blockchain tenga un impacto en el mundo exterior. Un ejemplo puede ser recibir una notificación en el móvil cada vez que se depositen criptomonedas en una wallet. Para hacer esto la información ha tenido que salir desde dentro de la blockchain a una aplicación externa que es la que ha mandado la notificación. Otro ejemplo es el oráculo que usaremos en nuestro prototipo para obtener el precio del gas en la Mainnet de Ethereum.

Aunque no hemos encontrado menciones, creemos que un oráculo podría implementarse tanto de manera Inbound como Outbound al mismo tiempo creando así un oráculo bidireccional, aunque como los contratos suelen desarrollarse con un propósito concreto es posible que esta implementación se haya explorado poco por el momento.

3.2. Sistemas existentes

Actualmente existen varias empresas interesadas en el uso de oráculos y qué utilizan protocolos muy diferentes entre ellos. Probar los productos de estas empresas no ha sido posible ya que todos eran modelos de pago, aún así, sí existía información relacionada con los sistemas utilizados por algunas de ellas.

3.2.1. Provable (Oraclize)

Provable [4], antes conocida como Oraclize, es una de las empresas principales dedicadas al mundo de los oráculos para redes como Ethereum, Rootsock o Hyperledger Fabric. El engine de Provable provee servicio tanto a aplicaciones on-chain como off-chain. El funcionamiento de este oráculo es el siguiente *(Figura 3.7)*:

1. El contrato hace una solicitud al Oráculo de Provable que tiene que seguir la estructura:
    - La fuente de datos a la cual quiere hacer la consulta de la información. Provable ofrece de manera nativa las siguientes opciones:
        - Una URL: de tal manera que se pueda hacer una consulta a una API o tener acceso a una página web.
        - IPFS: da acceso a ficheros del tipo IPFS.
        - WolframAlpha: activa el acceso a la red la inteligencia computacional de WolfamAlpha[2].
        - Random: provee bytes de manera aleatoria según una aplicación ejecutándose en un Ledger Nano S (también se conoce como cartera fría).

[2] https://www.wolframalpha.com

    - Computation: produce el resultado de una computación arbitraria.
  - La consulta: se considera una consulta al conjunto de parámetros que se le pasa, el primero suele ser obligatorio y es la fuente de datos, a partir de este se toman todos los demás como agumentos para un método HTTP Post.
  - Prueba de autenticación: aunque este parámetro es opcional, se puede pedir una prueba de autenticación de la información mediante algunos protocolos comom TLSNotary, Android Proof o Ledger Proof.
2. El engine de Provable recupera la información aportando las pruebas de verificación pertinentes a la consulta.
3. Manda la información al contrato que ha realizado la consulta en forma de call-back.

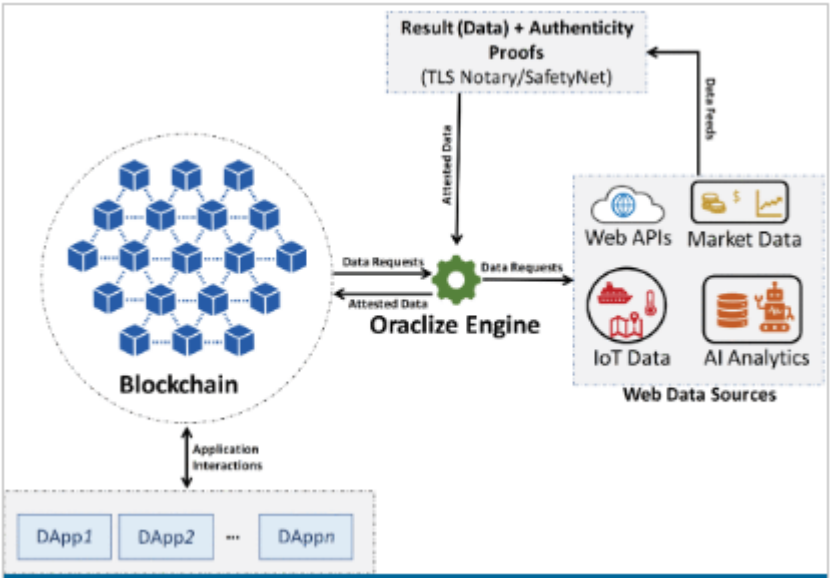

Figura 3.7: *Esquema de funcionamiento de Provable. [12]*

3.2.2. Chainlink

Chainlink [5, 6, 22] es posiblemente la empresa dedicada a los oráculos en blockchain con mayor uso en la actualidad, teniendo compatibilidad con redes como Ethereum, Bitcoin o Hyperledger. Ofrecen diferentes servicios a empresas como ser proveedor de datos mediante el uso de sus nodos o ser un nodo más de sus oráculos.

Chainlink ha afrontado el problema de los oráculos es mediante el uso de la descentralización. Su uso viene dado por las siguientes características:

- Diversidad de fuentes: Chainlink busca datos en múltiples fuentes. Esto reduce el riesgo de que una única fuente sea incorrecta o manipulada. Los oráculos pueden obtener los datos de diferentes API web, fuentes de datos tradicionales, dispositivos IoT...

- Descentralización: los oráculos operan de manera descentralizada. Esto significa que ya no dependen de una única entidad o servidor central para proporcionar los datos. En cambio, utilizan una red de nodos oráculo distribuidos, lo que ayuda a mejorar la seguridad y disponibilidad.
- Incentivos para los Nodos Oráculo: al igual que en muchas blockchains se premia económicamente el minado de bloques con un token, por ejemplo Bitcoin o Ethereum, en la red descentralizada de Chainlink se premia con LINK, un token nativo de esta red a aquellos nodos que provean datos precisos y verificables, de esta manera se garantiza en parte una buena conducta y servicios. Además el sistema está basado en reputación así que dar respuestas verdaderas es recompensado con mayor confianza.
- Pruebas de autenticidad: Los oráculos de Chainlink utilizan pruebas de autenticidad para verificar la integridad y fuente de los datos. Las pruebas pueden incluir firmas digitales, registros de tiempo y otros mecanismos para garantizar que los datos no han sido manipulados.
- Respuestas consensuadas: relacionado con la diversidad de fuentes y la descentralización, Chainlink utiliza un método de consenso entre los nodos para determinar la respuesta final que se proporciona al smart contract. Si la mayoría de nodos concuerda, la información se da como confiable, mientras que si hay desacuerdo o inconsistencia, se puede solicitar más datos o rechazar la respuesta.

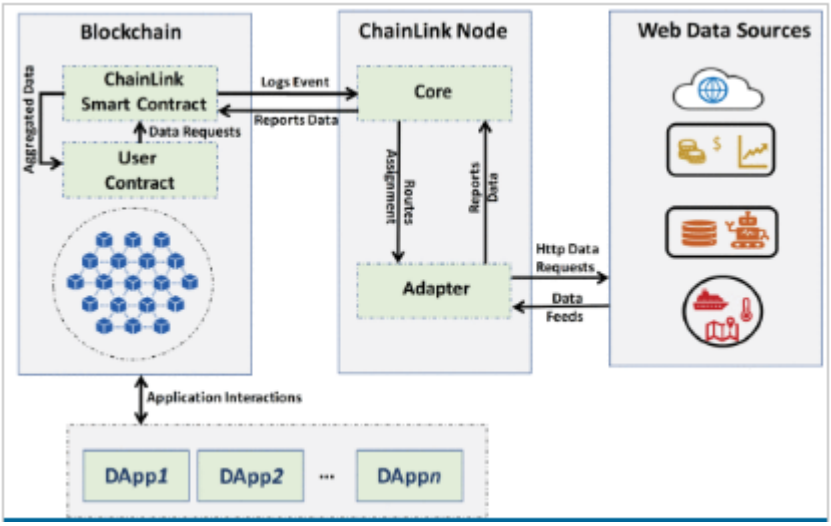

Figura 3.8: *Esquema del funcionamiento de Chainlink. [12]*

Además propone dos maneras de enviar esta información a la cadena [24]:

- On-chain aggregation(Agregación en cadena): de esta manera la agregación y el procesamiento de los datos se realizaría directamente en la blockchain. En este caso, los

nodos oráculo de Chainlink enviarían datos individuales a la blockchain, y luego, un smart contract de la cadena procesa y promedia estos datos para obtener un resultado final. Este tipo de agregación tiene la ventaja de que el resultado final se calcula de una manera totalmente transparente, ya que cualquiera puede consultar el smart contract que está recibiendo la información, y es verificable por cualquiera, lo que aumenta la confianza en el proceso.

- Off-chain aggregation (Agregación fuera de la cadena): de esta manera la agregación y el procesamiento de la información se realizan fuera de la blockchain. Los nodos oráculos envian la información a un proceso de agregación externo, por ejemplo un servicio proporcionado por el propio Chainlink, y luego el dato calculado se envia a la blockchain. Este enfoque tiene otras ventajas:

  - Eficiencia: suele ser mucho más eficiente ya que el procesamiento de datos suele ser muy costoso cuando se hace dentro de la propia blockchain.
  - Confidencialidad: en este caso si los datos a tratar son confidenciales, ofrece una privacidad que la agregación on-chain no te da.
  - Flexibilidad: Se puede utilizar para tipos de agregación diferente, desde promedios simples hasta cálculos más complejos que podrían dificultarse en la blockchain.

En resumen ambos métodos de agregación tienen sus ventajas y se selecciona dependiendo del caso de uso específico y los requisitos de seguridad, eficiencia y transparencia necesarios.

3.2.3. Astraea

Astraea [11, 17] es una propuesta interesante generada por parte de investigadores de Department of Electrical and Computer Engineering de la Universidad de Toronto pero que por el momento no parece tener una implementación conocida entre las empresas dedicadas a la tecnología blockchain.

La propuesta se basa en una blockchain de oráculos de propósito general que opera de manera pública y utiliza a los usuarios para un juego basado en un sistema de roles a su vez basado en el equilibrio de Nash [3]. Los usuarios se pueden divir en:

- Submitters: son aquellos usuarios que deciden qué proposiciones (de tipo boolean) entran al sistema y dejan dinero para ”financiar” esa propuesta y amortizar parte del esfuerzo que lleva validarla.

- Votante: juegan un papel seguro, es decir de bajo riesgo baja recompensa. Depositan una cantidad de dinero y luego se les permite votar sobre una afirmación aleatoria. El resultado de la votación se calcula sumando los votos, teniendo en cuenta el dinero

[3] https://es.wikipedia.org/wiki/Equilibrio_de_Nash

que cada votante ha depositado. Existe un límite máximo para la cantidad de dinero que se pueden apostar en un voto, y este límite es una regla fija en el sistema.

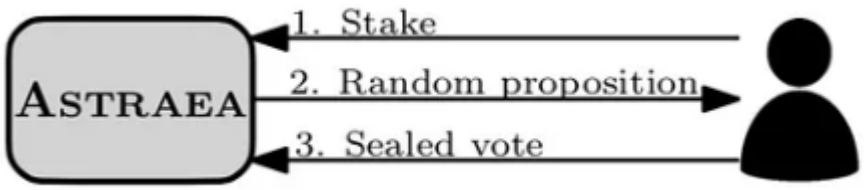

Figura 3.9: *Jugador votando en la propuesta Astraea. [11]*

- Certificadores: estos conllevan mayores riesgos que cualquiera de los otros dos roles. Eligen afirmaciones y ponen una gran cantidad de dinero en juego para certificarlas como verdaderas o falsas. La certificación se determina por la suma de las certificaciones ponderadas por los depósitos.

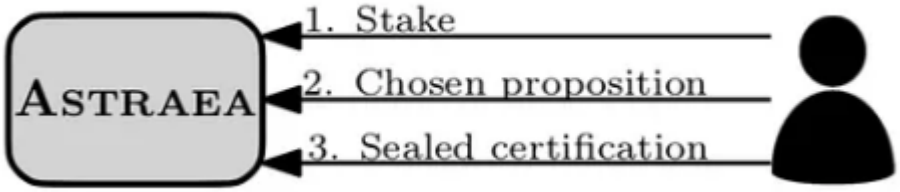

Figura 3.10: *Certificador en la propuesta Astraea [11]*

En el sistema no se requiere que todas las propuestas tengan certificaciones, ya que son los propios certificadores los que eligen las propuestas que certifican.

Una vez acaba el proceso de votación hay varios posibles escenarios:

- Votantes y certificadores coinciden: en este caso ambos bandos salen beneficiados a través de una recompensa basadas en la cantidad de fondos depositados.
- Votantes y certificadores no coinciden: en este caso ambos son penalizados también en base a los fondos depositados.
- El resultado es desconocido: en este caso como bien se ha mencionado con anterioridad, al tener un rol de bajo riesgo los votantes no son penalizados, sin embargo sí los certificadores.

Uno de los peligros de este juego está relacionado con la honestidad de los votantes, ya que si los votantes decidiesen mentir las manipulaciones del resultado serían más sencillas.

En el artículo se propone como posibles aplicaciones de este tipo de oráculos: para etiquetación de datos para su uso en Machine Learning, esto es debido a la falta de personas para etiquetar los datos debido a los bajos pagos por el trabajo y pocos incentivos a realizar la etiquetación de manera correcta, o para mediaciones que requieran de un voto público.

Capítulo 4

Diseño del prototipo de los oráculos

En nuestro caso para el estudio de diferentes modelos de oráculo decidimos implementar dos tipos de oráculos muy simples y así poder hacer estudio de eficiencia y coste y un modelo económico posible de esta implementación. Los oráculos elegidos fueron el modelo de immediate-read y el modelo de publish-subscribe.

Nuestro modelo de publish-subscribe presenta una variación al modelo usual. En el modelo usual los suscriptores se almacenan fuera de la propia red. En nuestro caso el almacenamiento tanto de de los datos a proveer como los clientes están almacenados dentro de la cadena. Así que, se puede decir que es un oráculo completamente integrado en la blockchain, obviando el script externo que envía la información.

A continuación vamos a hacer una breve introducción al software utilizado para el desarrollo de los prototipos, ya que es pertinente para el diseño de la aplicación mostrado en la *Figura 4.1*.

4.1. Entorno de ejecución del prototipo

La primera decisión tomada fue considerar sobre qué configuración se iba a implementar estos prototipos. Se empezó pensado en una implementación sobre una máquina Windows aunque tras unas breves pruebas se descartó por existir varios problemas entre el antivirus, Windows Defender, con el software que se estaba intentando utilizar, Geth, y por problemas de rendimiento. Finalmente se optó por el uso de una máquina Linux, en este caso, el sistema operativo elegido fue: Ubuntu 22.04 Jammy JellyFish[1].

Primero estudiamos la posibilidad de trabajar en una máquina en remoto, ya que no estábamos seguros, habiendo visto el rendimiento del primer intento, de si nuestro portátil poseía la capacidad física de ejecutar todas las herramientas que se iban a utilizar en el desarrollo de una manera cómoda para el trabajo. A falta de espacio en el servidor del equipo, se nos

[1] https://releases.ubuntu.com/jammy/

prestó un portátil el cual conectamos a la red de la Complutense tras solicitar un puerto ethernet para conseguir acceso a la red y, mediante el "Remote Desktop Protocol" y "ssh" ir trabajar sobre el proyecto. Se dieron diversos problemas como: si la luz se iba en la facultad, cosa que pasó varios fines de semana, el portátil pedía la conexión y había que reestablecerla a mano en el portátil, pero no se podía, o que por un fallo de interacción entre el sistema operativo y el hardware del equipo, el cual se intentó solucionar pero acabó siendo imposible, la pantalla quedaba bloqueada y era necesario un reinicio para desbloquearla además de tener que volver a exponer el ordenador mediante `RDP` . Por tanto se acabó trabajando en una torre personal utilizando una máquina virtual de "VirtualBox" con el mismo sistema operativo.

Además elegimos la red Ethereum como la base del proyectoya que esta red permite el desarrollo de aplicaciones descentralizadas mediante el uso de contratos, estos contratos son capaces de: envíar y transacciones, ejecutar piezas complejas de código, etc.

4.2. Software utilizado en el desarrollo

En este apartado se describen todas las herramientas, frameworks y software en general utilizado para llevar a cabo todo el desarrollo de los prototipos.

4.2.1. Visual Studio Code

Visual Studio Code fue el editor de texto elegido para el desarrollo debido a la cantidad de plug-ins que se pueden instalar, entre ellos compiladores y lectores de javascript, solidity y bash que son los principales lenguajes que se han utilizado a lo largo del proyecto.

4.2.2. Geth

Geth es el software que ejecutan los nodos de una red de blockchain Ethereum para procesar transacciones y generar nuevos bloques en la cadena. Además, Geth permite realizar el despliegue y ejecución de contratos inteligentes. Aunque hay varios sistemas que permiten a un nodo participar en una red Ethereum, Geth es el más utilizado, y por ello fue el primer software elegido para realizar este estudio.. Algunas de sus características son:

- Cliente Ethereum.
- Escrito en Go: esto hace que en teoría sea relativ amente eficiente en términos de recursos.
- Personalización: ofrece posibilidades de personalización y configuración para adaptarse a las necesidades del usuario, ya sea a la hora de crear nuevas cuentas o manejo de despliegues de contratos.

Además para su uso se intentó seguir la guída dejada por Matías Ruíz Lotito en su trabajo de fin de grado del año pasado [21], pero debido a problemas en la máquina inicial, Windows, sufriendo cosas como interacciones indebidas con Windows Defender, fallos durante la ejecución del programa y fallos de instalación, etc. Además ser un software poco amigable a la hora de visualizar datos, ya que es un programa que funciona por interfaz de línea de comandos (CLI), se decidió explorar otras posibilidades en auge en el desarrollo de la tecnología blockchain.

4.2.3. Truffle Suite

Truffle Suite es uno de los conjuntos de herramientas y entornos más utilizados a la hora de desarrollar cualquier tecnología blockchain en Ethereum Virtual Machines. Fue con la que finalmente se desarrolló nuestro , ya que las herramientas que ofrecen poseen una muy buena interacción entre ellas y un uso sencillo para el desarrollador. Los servicios de este conjunto que se han usado son:

- Truffle: es un framework de desarrollo de DApps que simplifica y agiliza el proceso de creación y despliegue de smart contracs. Ofrece un conjunto de herramientas que incluyen:
  - Compilación y migración de contratos: Truffle permite compilar tus contratos inteligentes escritos en Solidity y migrarlos a la cadena de bloques Ethereum.
  - Pruebas automatizadas: Proporciona un marco para escribir y ejecutar pruebas automatizadas para tus contratos inteligentes y DApps.
  - Despliegue de contratos: Facilita el despliegue de contratos inteligentes en una variedad de redes Ethereum, como la red de prueba o la red principal, en este caso nos centramos en una red de prueba propia.
  - Gestión de artefactos: Genera artefactos JSON que contienen información sobre los contratos inteligentes compilados, lo que facilita su uso en otras herramientas y aplicaciones.
  - Opciones de depuración avanzadas: con análisis de variables y la opción de compilar paso a paso.
  - Creación de Dashboards para conexiones con wallets y aplicaciones externas.
- Ganache: herramienta que cuenta con una interfaz de usuario muy simple y fácil de entender que facilita la creación de redes blockchain a los desarrolladores. Algunas de sus características son:
  - Blockchain local: permite la creación de una blockchain local de manera sencilla, con solo presionar un botón, o de forma avanzada, el usuario puede configurar el valor de varios parámetros de la blockchain.

- Desarrollo sin gastar ETH real: las cuentras que te pregenera Ganache y que el programador puede usar para cualquier prueba, vienen cargadas de ETH falso (simulado) para que puedas simular todos los escenarios que necesites.
- Visualización detallada: La GUI de Ganache te permite consultar en todo momento todo lo que está sucediendo dentro de la blockchain generada, desde valores de variables de los contratos que tienes desplegados, a los eventos que se van lanzando. Esto facilita la depuración ya que eres capaz en todo momento de consultar transacciones, saldos, bloques...

4.2.4. Node.js

Node.js es un entorno de tiempo de ejecución de JavaScript de código abierto que ha supuesto una revolución en la forma en que se desarrollan aplicaciones web y servidores. Algunas de sus características más destacables son:

- JavaScript en el Servidor: Node.js permite a los desarrolladores utilizar JavaScript tanto en el lado del cliente como en el servidor, lo que simplifica la puesta en marcha de aplicaciones web de extremo a extremo.
- Eficiencia y Rendimiento: Gracias a su arquitectura basada en el modelo de concurrencia de un solo hilo y el bucle de eventos, Node.js es altamente eficiente y puede manejar muchas conexiones simultáneas sin agotar recursos del sistema.
- Módulos y Ecosistema: Node.js cuenta con un vasto ecosistema de módulos y paquetes de código abierto disponibles a través de npm (Node Package Manager), lo que facilita la reutilización de código y acelera el desarrollo de aplicaciones. En este proyecto se ha usado npm para la instalación de las librerías web3 y fetch utilizadas en la parte de la aplicación de control de eventos.
- Escalabilidad: Node.js es ideal para aplicaciones que necesitan escalabilidad horizontal, lo que significa que se pueden agregar más servidores para manejar una mayor carga de trabajo, manteniendo un rendimiento óptimo.

En nuestro caso se ha utilizado tanto como para la creación de los scripts de control de eventos como por ser un requerimiento de uso de Truffle.

4.2.5. Remix IDE

Como su nombre indica Remix IDE es un entorno para el desarrollo de smart contracts. Aunque tiene su propio editor de texto y también compila automáticamente los contratos para ver fallos en ellos, lo hemos utilizado principalmente por su capacidad para conectarse a la red Ganache y así, con una interfaz muy simple, ser capaz de hacer las diferentes llamadas a los contratos desplegados en la red durante la realización de pruebas.

4.3. Arquitectura

Ahora que hemos hablado de las herramientas utilizadas para el desarrollo ,se va a ver más en profundidad la arquitectura que se ha creado para el funcionamiento de los prototipos y los factores y características que se han tenido en cuenta para su diseño.

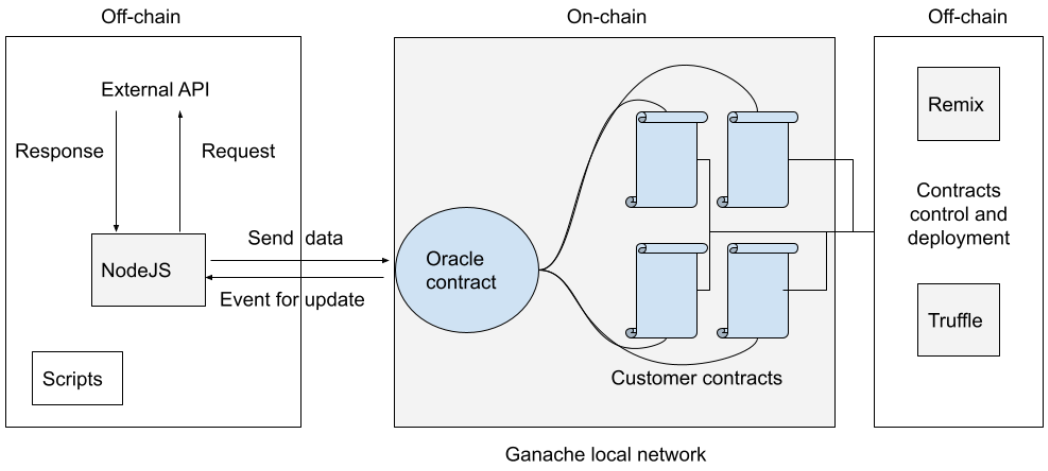

Figura 4.1: *Arquitectura completa de la aplicación*

En la *Figura 4.1* se muestra una representación gráfica de la arquitectura general de los prototipos.

En el recuadro off-chain de la izquierda encontramos dos cajas:

- NodeJs: Esta caja se refiere a una aplicación NodeJs que es la que va a obtener los datos de las APIs externas y enviará los datos a la blockchain mediante métodos de las librerías Web3 y Fetch. Va a actúar según el protocolo WebSocket para escuchar los eventos de la blockchain. Algunos aspectos más importantes al diseñar esta aplicación han sido:
  - Conexión con la red blockchain: En este caso al ser una red local, es necesario especificar dónde está expuesta para así poder escuchar los eventos y responder a ellos.

- Asincronía: Aunque puede que no sea demasiado importante para un prototipo a pequeña escala, en modelos reales es necesario que exista una asincronía y la aplicación JavaScript se quede esperando tanto las respuestas como los eventos de manera no bloqueante. Esto en parte es debido a la naturaleza de la blockchain. Como no se sabe con seguridad cuándo un nuevo bloque va a ser minado por un nodo, es decir, no hay un tiempo constante en la generación de bloques, es importante que la consulta de los eventos no se realice una vez y además que no bloquee procesos que no dependan de la generación del evento. Un ejemplo podría ser que los datos se siguiesen actualizando en el script fuera de la blockchain, pero esto no dependiese de si se genera el evento o no. En este caso se ha realizado mediante el uso de Promesas: `.then()` y `await`.
- Json build del contrato a utilizar: el json que se genera al compilar los contratos contiene el ABI y el Bytecode. El ABI o application binary interface, define qué métodos y variables del contrato se pueden utilizar para interactuar con un contrato. En el caso del Bytecode es la traducción del contrato a código máquina, aunque es observable en hexadecimal y, en el caso de las redes Ethereum, es lo que se despliega en la red y se ejecuta cuando interactuamos con otros smart contracts. Sin el ABI principalmente, no seríamos capaces de saber qué métodos tiene el contrato a la hora de hacer una llamada y por tanto la librería Web3 no encontraría el método que se está intentando llamar a la hora de realizar la transacción al contrato.
- Cuentas de los contratos [7]: especificar la dirección de la cuenta de propiedad externa y dirección del contrato es necesario para la el uso de las acciones con contratos. La cuenta externa es aquella que se considera dueña de los contratos que se despliegan utilizándola como base y es manejable por todo aquel que tenga su clave privada. La dirección del contrato es necesaria ya que sirve para interactuar con un contrato concreto. En nuestro caso la dirección de cuenta se utiliza para verificar que es el administrador del contrato quien está intentando enviar la información adquirida y la dirección del contrato para la instanciación de ese contrato gracias a la librería web3 y su ABI, permitiendo así hacer referencias a los métodos contenidos dentro de este.
- Especificar la apiKey. Como vamos a hacer uso de la api que ofrece etherescan [10] para conseguir el precio en tiempo real del gas, es necesario crearse una cuenta en la web y extraer la apiKey que luego habrá que añadir a la url de la API para tener acceso a los datos. Algo parecido hay que hacer con la API de la NASA[2]

Una vez se ha tenido en cuenta todo esto el funcionamiento debería ser el siguiente (*Figura 4.2*):

1. Se consulta cuál es el último bloque de la cadena que ha sido minado mediante métodos de la librería web3: `getBlockNumber()`. Esta función devuelve el número

[2] https://api.nasa.gov/

del próximo bloque a minar, pero en nuestro caso estamos interesados en obtener el último bloque minado, que corresponde con el bloque n-1, y escuchar los eventos que se han registrado en él.

2. Recogemos los eventos del último bloque minado que se llamen igual que el evento dentro del contrato que hemos programado.
3. Si ha habido evento entonces procedemos a hacer `fetch()` de las diferentes APIs y extraemos la información que precisemos de la respuesta en forma de json.
4. Una vez extraída la información utilizamos la instanciación del contrato contenida dentro del script de control y la librería web3 para generar una nueva transacción haciendo una llamada al método del contrato que se encarga de actualizar la información. Además habrá que especificar que la transacción se está haciendo mediante el address de la cuenta dueña del contrato para que este lo acepte.

En nuestro caso además existen dos diferentes scripts de node, uno para cada oráculo.

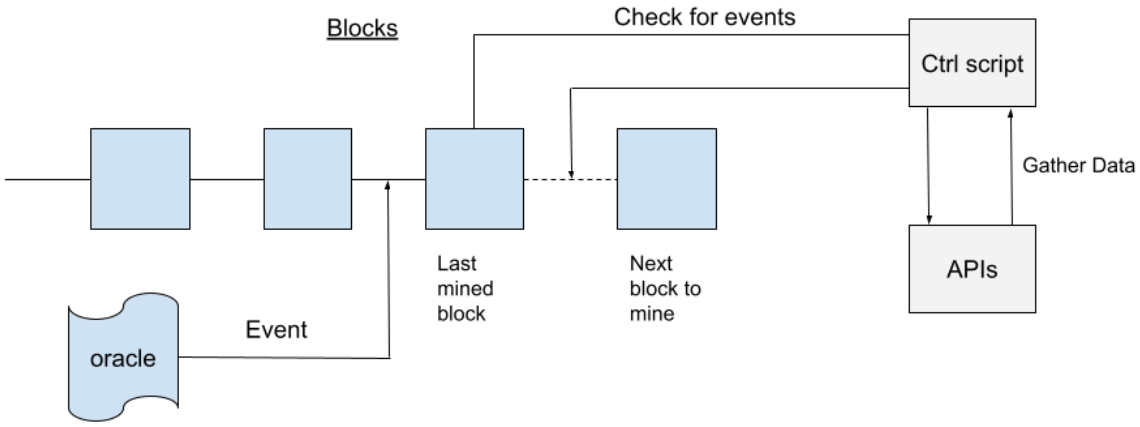

Figura 4.2: *Control de eventos en blockchain*

- Scripts: estos scripts controlan el funcionamiento de los scripts de JavaScript exponiéndolos y terminando los procesos una vez han actúado de tal manera que estos puedan responder a todos los eventos que se generen.

En la parte derecha de la *Figura 4.1* encontramos otra parte off-chain. En este caso son Remix y Truffle. El proyecto Truffle solo se utiliza para el despliegue dentro de la red local Ganache, mientras que Remix se utiliza para hacer pruebas en los contratos desplegados de una manera sencilla gracias a su interfaz, siendo capaces de ver la transacción completa de principio a fin.

Por último tenemos la parte central de la *Figura 4.1*, en la que encontramos la parte on-chain, la red Ganache. Respecto al diseño de la red local lo único a tener en cuenta son el puerto y la dirección donde se va a exponer la red para poder tener comunicación con ella y, en la configuración interna de la blockchain, el gasLimit.

Esto es debido a que este gasLimit controla la cantidad de gas que se puede utilizar en una transacción. Algo a tener en cuenta es el concepto de "gas " en las redes Ethereum:

"El gas hace referencia a la unidad que mide la cantidad de esfuerzo computacional requerido para ejecutar operaciones específicas en la red de Ethereum.

Como cada transacción de Ethereum requiere recursos computacionales para ejecutarse, cada transacción requiere una comisión. El gas hace referencia a la comisión necesaria para llevar a cabo una transacción en Ethereum con éxito." [8]

Por tanto este gasLimit representa el máximo gas que va a poder utilizar cualquier contrato a la hora de realizar transacciones aunque, en la práctica, todas las transacciones deben controlar el uso indiscriminado de gas. Perder el gas que tiene un contrato hará que la transacción vuelva a su estado inicial y el gas perdido no se recupere provocando grandes pérdidas monetarias para el dueño del contrato ya que, por ejemplo, el precio de una unidad de gas medio actualmente es de 27,7278 Gwei $\simeq 0,000045$ € , si tenemos en cuenta el límite medio actual: $30M * 0,000045 = 1{,}370,7812$ de pérdida en caso de sufrir un ataque y perder todo el gas. Esto es teniendo en cuenta que el precio del gas ha ido bajando a lo largo del tiempo, aún así se pueden ver días como el 6 de mayo de 2023 en el que 1 unidad de gas costaba 155.84 Gwei lo cual generaría aún más pérdidas [3].

En el siguiente capítulo se describe en detalle el diseño de los contratos desplegados en la red de blockchain que forman parte de los prototipos de oráculos desarrollados en este trabajo

[3] https://etherscan.io/chart/gasprice

Capítulo 5

Prototipos

Decidimos implementar estos prototipos [1] para hacer un estudio de costes, optimización y, en el caso del Publish-Subscribe, hacer un estudio que permita evaluar si merece la pena integrar la parte de subscripción dentro de la blockchain para mayor transparencia a costa de encarecer costes y tener que realizar un oráculo más complejo.

En nuestro caso utilizamos dos APIs como ejemplo de fuente externa de la cual extraeremos datos para enviarlos a la red blockchain local a través de los scripts de control y el contrato oráculo.

La primera API usada es un API de la NASA[2]: en este caso utilizamos la parte que nos da información acerca de cuerpos que han pasado cercanos a la Tierra en un intervalo de tiempo especificado. Dejar claro que este API podría ser sustituído por cualquier otra fuente de información del mundo exterior.

La segunda API utilizada es el API de Etherscan[3]: gracias a otro oráculo propio de la red Ethereum la API de Ethersan es capaz de obtener el precio interno del gas en la mainnet y lo traslada nuestro oráculo a través de las solicitudes hechas por nuestro script de control. Esto nos permitirá hacer cálculos utilizndo el precio real del gas en el momento de la actualización de nuestro dato. Este es otro ejemplo del uso de Oráculos para la obtención de datos internos de una blockchain por una aplicación externa.

[1] Código del proyecto: https://github.com/NachoSC/TFG
[2] https://api.nasa.gov/
[3] https://docs.etherscan.io/api-endpoints/gas-tracker

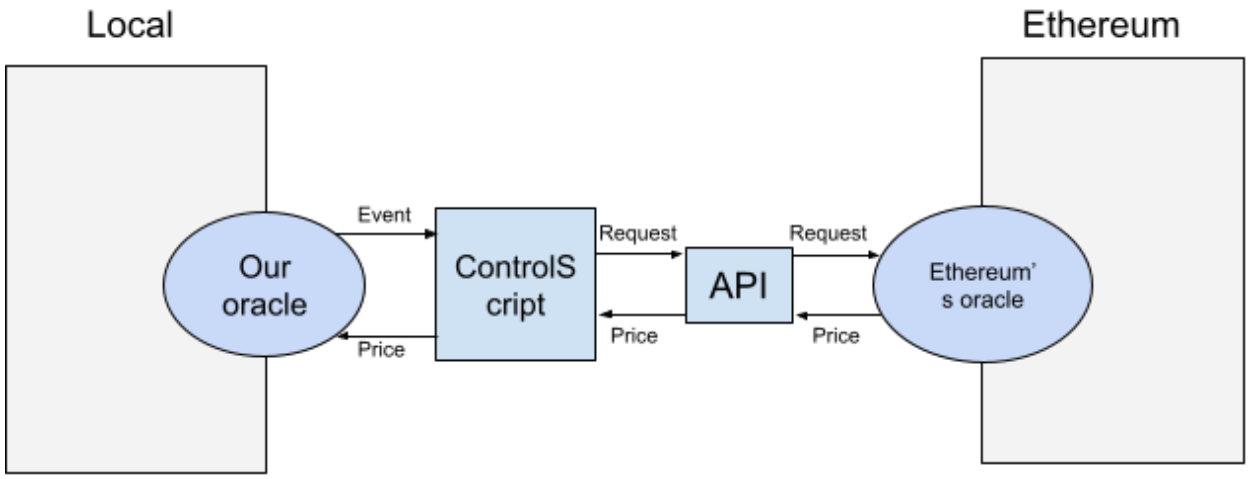

Figura 5.1: *Gas price obtention architecture*

5.1. Oráculo Immediate-Read

Como se ha indicado, se va a utilizar como ejemplo de fuente de datos externas el API de la NASA. Se ha elegido este ejemplo porque existe un oráculo muy sencillo sobre esta fuente de datos desarrollado por Alberto Lasa[4], publicado en su github [5].

Nosotros le hemos incorporado otro atributo que será el precio del gas en el momento de la actualización. Esto nos sirve para hacer diversos cálculos para un estudio de rentabilidad del oráculo ya que, el precio a pagar por parte de los consumidores estará basado en el precio de la actualización del propio oráculo en el momento de renovar el atributo del número de asteroides. La forma de pago se inlcuye mediante el cálculo del costo de actualización del dato, es decir, el dueño del oráculo calcula cuanto le cuesta actualizar el dato y esto es lo que pide a cambio de la información.

Tras diversas ejecuciones de prueba, podemos decir que el coste de generar el evento de actualización de los datos para ser capturado por los scripts de control más el coste de actualización de los atributos internos, ronda en torno a 800000 uds de gas. A día 10/09/2023 a las 18:34 el precio medio del gas es de 9 Gwei por lo tanto el precio del dato sería 720.000 Gwei. Este modelo de pago beneficia a los consumidores, ya que los precios de actualización de un dato al ser una asignación no suele ser demasiado altos y, al dueño del contrato le sale rentable a partir de la primera request, ya que todas las demás después de esta son beneficio.

Existe la posibilidad de realizar otros modelos económicos con un estudio en mayor profundidad del coste de la operativa por parte del dueño del contrato. Por ejemplo se podría

[4] https://github.com/AlbertoLasa
[5] https://github.com/AlbertoLasa/creacion-oraculo-ethereum

poner un coste fijo al precio del dato en función a un estudio del número habitual de usos que tiene nuestro oráculo, y repartiendo parte del precio de la actualización entre varios usos, de esta manera se abarataría el precio para los consumidores haciéndolo más atractivo para ellos, aunque se tardaría más en presentar la misma rentabilidad que la opción implementada por nosotros.

5.2. Oráculo Publish-Subscribe on-chain

Este oráculo parte de la implementación del Immediate-Read. En este caso el oráculo en vez de recibir una solicitud del dato recibe una solicitud de subscripción junto con un pago de fondos que se guardarán en el contrato oráculo. Cuando se renueve el dato, el contrato recorrerá la lista de subscriptores y a aquellos que tengan fondos suficientes depositados en el contrato se les enviará el dato actualizado.

En este caso se le añaden nuevos atributos al contrato aparte del precio del gas, la dirección de la cuenta dueña y el número de asteroides:

- `address[] private addresses`: este array va a contener todas las direcciones de los contratos que decidan suscribirse al oráculo. Es un array dinámico que irá creciendo o decreciendo según la necesidad del contrato.
- `mapping(address =>uint) private customers`: este mapa contiene la información de la cantidad de dinero que tiene invertido un cliente en el oráculo. De esta manera los clientes pagan al suscribirse, o posteriormente para aumentar sus fondos, y este dinero queda registrado por el contrato. En el momento en el que se envía la información, el contrato oráculo tendrá en cuenta qué personas tienen los fondos suficientes como para pagar por la información actualizada. De esta manera tras la comprobación realizará el envío del dato.
- `mapping(address =>uint) private addressInArray`: este mapa tiene como clave la dirección de los contratos suscritos y como valor se utiliza su índice del array (n). Como el valor por defecto de una clave que no existe en el mapa es de 0, en verdad lo que se guarda es la posición $n + 1$ ya que así podemos utilizar el valor de 0 para comprobar si una dirección existe o no en el array.

Existen otros atributos importantes como `bool private lock` o
`uint private nextToGiveData` que se tratarán más adelante en el análisis de seguridad del contrato.

Para entender la existencia de estos atributos primero se tiene que entender que en Solidity se aconseja evitar lo máximo posible el uso de bucles. Debido a la noción de gasLimit anteriormente mencionada, el uso de recursos durante una operación en la blockchain no es infinito, sino que viene marcado por el gas. Al utilizar bucles el consumo de gas se dispara y en caso de llegar al límite de gas establecido por la red u operación esta fallará, causando

lo que ya se ha mencionado con anterioridad. Es por esto que se crean estos tres atributos. Podemos observar un ejemplo de su uso en la *Figura 5.2*. En este caso se evita la realización de recorridos de arrays mediante el uso de los atributos propios manejados con la dirección del cliente, lo que hace que el número de accesos a las estructuras de datos sea constante. Esto último asegura que en caso de crecer en gran medida el número de clientes, el gasto en gas de la operativa sea ”constante”.

```
//pagado por el subscriptor
function unsubscribe() external {
    require(addressInArray[msg.sender]>0, "You are not subscribed!");
    uint toGiveBack = customers[msg.sender] - customers[msg.sender]/100;
    addresses[addressInArray[msg.sender]-1]=addresses[addresses.length - 1];
    addressInArray[addresses[addresses.length-1]]= addressInArray[msg.sender];
    addresses.pop();
    customers[msg.sender]=0;
    (bool sent,) = msg.sender.call{value: toGiveBack}("");
    require(sent, "Failed to send Ether");
}
```

Figura 5.2: *Función unsubscribe*

Otro de los aspectos más fundamentales del diseño era el procedimiento para mandar la información del contrato oráculo *Publish-Subscribe* a los contratos cliente. Esto se debe a que en el oráculo *Immediate-Read* la llamada se realiza desde el cliente generando una solicitud al oráculo a través de una función existente en el contrato del oráculo. En el caso del esquema *Publish-Subscribe* esto cambia. Es el contrato oráculo el que inicia la transacción realizando las llamadas a los clientes para entregarles el dato. Para que el óráculo pueda invocar correctamente el código de los clientes, se ha tomado la decisión de generar una interfaz, IOracleCustomer, que todo cliente que quiera suscribirse al oráculo ha de implementar para poder recibir la información de manera correcta.

En cuanto al modelo económico utilizado en este contrato, varía del modelo del Immediate-Read. En este caso se ha utilizado un modelo de porcentajes relativamente bajos 1 %, 2 % del precio de la operación como comisión. Además este oráculo tiene un gran coste añadido del cual el otro carece y es que este asume todo el gasto de gas de la transacción. Esto ocurre debido a que es él quien realiza las llamadas y, en el caso de Solidity, quien inicia la comunicación es quien incurre en el gasto de gas. Este coste de la operación se calcula y reparte entre todos los contratos que utilicen el contrato de suscripción. Además, también debido a esto a cada cliente se le proporciona un gasLimit de 200000 uds/gas para poder ejecutar código en la implementación de la interfaz. En cualquier caso, esta cantidad de gas, termine siendo usada o no, se les cobra.

5.3. Seguridad en el contrato del oráculo

En este apartado vamos a tratar diferentes problemas de seguridad encontrados y analizados durante del desarrollo y las formas en las que se han tratado cada uno de ellos.

5.3.1. Out of Gas

Una de las mayores problemáticas de los bucles en Solidity es que consumen mucho gas, sobre todo si estos bucles realizan llamadas externas. Para evitar quedarse sin gas durante la ejecución del bucle durante el cual se envían los datos a los clientes suscritos y que se produzca el revert de todas las ejecuciones anteriores se ha implementado lo que se denomina "función reanudable (resumable function)".

Este patrón de diseño se basa en tener el índice del bucle guardado como atributo del contrato. En nuestro caso recibe el nombre de nextToGiveData *(Figura 5.3)*. De esta manera se inicia el búcle en el estado en el que esté el atributo. Si el contrato se va a quedar sin gas, existe una función del sistema que nos informa del gas que le queda al contrato, lo que se hace es salirse del bucle y guardar el índice por el cual se ha quedado. De esta manera el administrador puede suministrar más gas y llamar de nuevo a la función para que esta continúe desde donde se quedó.

```
function sendData() public onlyOwner{
    uint i = nextToGiveData;
    // falta el calculo del precio a cobrar
    uint finalPrice = (price* 1000000000 * 200000 + (68000*price)/
    while(i < addresses.length && gasleft() > 300000){
        sendDataStep(addresses[i], finalPrice);
        i++;
    }

    if( i >= addresses.length){
        nextToGiveData = 0;
    }
    else{
        nextToGiveData = i;
    }
}
```

Figura 5.3: *Función resumable*

5.3.2. Reverts

Otro de los problemas más habituales de los bucles consiste en que si se realiza una llamada a otro contrato y se produce un revert, toda la ejecución del bucle se para y revierte, ya que el contrato volvería a su estado incial como si las llamadas nunca se hubiesen

realizado. Esta es la manera que tiene Solidity de asegurarse de que código malicioso no haya podido interactuar con ejecuciones anteriores.

Un ejemplo de este fallo podría verse a la hora de realizar la actualización de datos en uno de los clientes. Si se realizase una llamada de alto nivel a la función de la interfaz IOracleCustomer que implementan los clientes y esta fallase por algún motivo, se produciría un revert que haría que ningún contrato recibiese la información nueva y al contrato oráculo le supondría la pérdida del gas hasta la ejecución de esta llamada.

En este caso la solución encontrada es el uso de llamadas de bajo nivel [23] *Figura 5.4*. Las llamadas de bajo nivel te permiten llamar a funciones invocándolas con la ABI de un contrato específico y en caso de fallo, este se puede recoger en forma de booleano, lo cual permite que si en alguna de las iteraciones del bucle el cliente revierte la ejecución, se sigan procesando los demás clientes.

```
function sendDataStep(address _contract, uint finalPrice) private {
   // require(IOracleCustomer(_contract).getDataFromSubsOracle(,) = 0, "the contract does not implement IOracle so data cannot t
    if(customers[_contract] >= finalPrice){
        customers[_contract]-=finalPrice;
        //Oracle(_contract).getDataFromOracle{gas:50000}(numberAsteroids);
        (bool success,) = _contract.call{gas: 200000}(abi.encodeWithSignature("getDataFromSubsOracle(uint)", numberAsteroids));
    }
}
```

Figura 5.4: *Llamada de bajo nivel a la interfaz IOracleCustomer*

5.3.3. Reentrancy

Un ataque de reentrancy es uno de los más comunes y más devastadores ataques que se puede realizar a un contrato. El problema se genera cuando un contrato realiza una llamada a un contrato desconocido y este contrato desconocido hace una llamada al contrato original creando un bucle recursivo con el fin de dejarlo sin gas o fondos. De esta manera se bloquea el contrato que realizó la primera llamada.

Aunque en nuestro caso existe la posibilidad de reentrancy, la evaluación y análisis del ataque concluyó en que se iba a permitir ya que no suponía un coste para el dueño del oráculo. Esto es debido a que la posibilidad se da en el momento en el que el contrato envía la información al cliente. El cliente tiene dos posibilidades de reentrancy:

- Llamar a la función unsubscribe: como los fondos se le cobran de los que tiene almacenados en el contato oráculo antes de enviarse, aunque se desuscriba, el atacante hubiese perdido el dinero, además como al desuscribirse se realiza una comisión sería más dinero para el dueño del contrato. Si se llega al gasLimit, la operación falla, pero como la llamada es una llamada a bajo nivel no revertiría el contrato a su estado inicial, si no que se podría manejar como si fuese una excepción.

- Llamar a la función subscribe: en este caso como el atacante se supone que ya tiene

una cuenta creada, se haría un revert, pero como se ha explicado con anterioridad, la llamada de bajo nivel capturaría ese revert como si de una excepción se tratase.

Aún sabiendo que el ataque de reentrancy no supondría un costo para el dueño del contrato, una fácil solución es el uso de un atributo lock como se muestra en las *Figuras 5.5 y 5.6* . Si antes de realizar una llamada a un contrato cambiamos el estado del atributo booleano lock a false y comprobamos el estado de este atributo nada más entrar a la función subscribe o unsubscribe se puede parar el ataque. Esto es debido a que el estado cambiado del atributo lock se mantiene tras su llamada.

```
function sendDataStep(address _contract, uint finalPrice) private
    // require(IOracleCustomer(_contract).getDataFromSubsOracle(,)
    if(customers[_contract] >= finalPrice){
        customers[_contract]-=finalPrice;
        lock = true;
        //Oracle(_contract).getDataFromOracle{gas:50000}(numberAst
        (bool success,) = _contract.call{gas: 200000}(abi.encodeWi
    }
}
```

Figura 5.5: *Bloqueo de las funciones mediante atributo lock*

```
function subscribe() external payable{
    require(lock, "Reentancy attack");
    require(msg.value > 0, "No funds were found on the message");
    //problema con ver si existe ya en el array, puede dar duplica
    //mucho el coste de la operación si hay que ver si ya existe,
    //exist?
    customers[msg.sender] += msg.value;
    if(addressInArray[msg.sender]==0){
        addresses.push(msg.sender);
        addressInArray[msg.sender] = addresses.length;
    }
}
```

Figura 5.6: *Revisión del atributo lock antes de la ejecución de la función*

Por otra parte, otro de los motivos por los cuales un atacante no puede realizar un ataque basado en consumo excesivo de gas, es porque la limitación de 200000 ud. de gas que puede consumir la función del cliente impide que un cliente malicioso pueda consumir más del permitido.

5.4. Modificadores

En la programación del oráculo se han utilizado modificadores en todas las funciones, tanto los proporcionados por el lenguaje Solidity como otros programados para este oráculo, para controlar la disponibilidad de las funciones y limitar la visibilidad pública de las funciones que no sean imprescindibles. El lenguaje Solidity dispone de los siguientes modificadores de visibilidad:

- Internal: las funciones de este tipo solo pueden ser usadas por funciones pertenecientes al propio contrato.
- External: las funciones de este tipo solo pueden ser usadas por funciones no pertenecientes al propio contrato.
- Públic: las funciones de este tipo pueden ser usadas por todo el mundo, independientemente de si pertenecen o no al contrato.
- Private: las funciones de este tipo solo pueden ser usadas por funciones pertenecientes al propio contrato y no se derivan a otros contratos.

Además de estos modificadores se han utilizado otros como view, el cual permite retornar un valor e imposibilita que esa función altere el valor de estado del contrato o, modificadores propios, es decir, creados por nosotros como OnlyOwner que hacen que las funciones solo puedan ser ejecutadas por el dueño del contrato.

Capítulo 6

Resultados de las pruebas

Para realizar las pruebas nos hemos valido de la creación de un contrato que simula el uso simple de los oráculos por parte de un cliente[1] y de un cliente defectuoso que provocaría un revert. A continuación presentamos varias imágenes de la interacción de estos contratos y los oráculos:

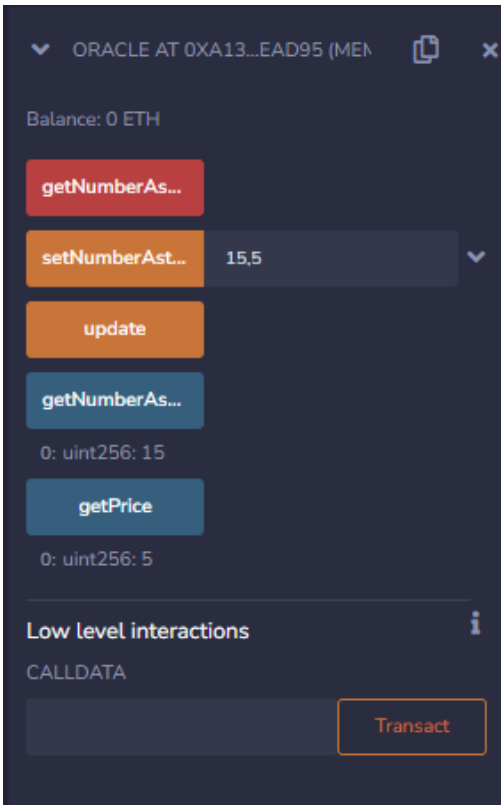

Figura 6.1: *Se actualizan los datos del número de asteroides y el precio del gas a 15 y 5 respectivamente en el oráculo Immediate-Read*

[1]https://github.com/NachoSC/TFG/blob/main/contracts/Customer.sol

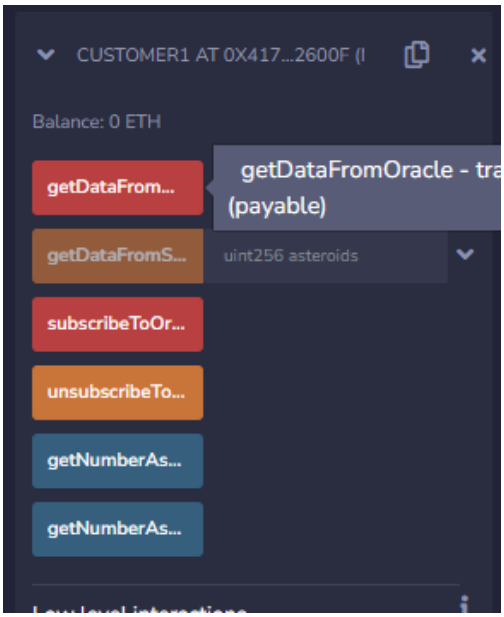

Figura 6.2: *Desde el contrato del consumidor se hace la request pagando 1 Ether al contrato Immediate-Read*

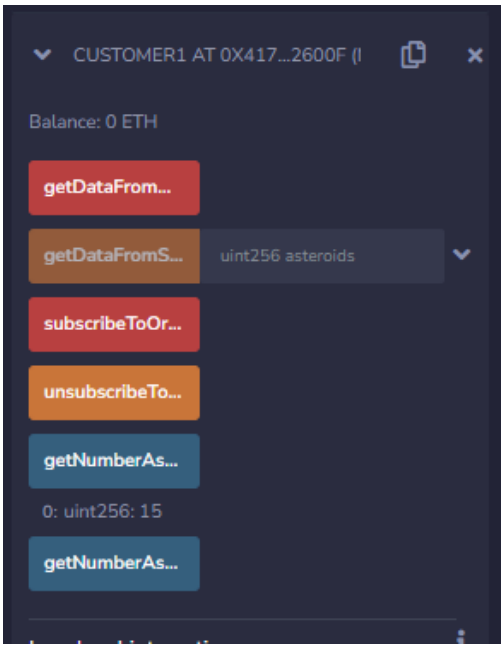

Figura 6.3: *El dato es recibido por el contrato consumidor*

Figura 6.4: *El Ethere del pago por el dato es recibido por el contrato oráculo*

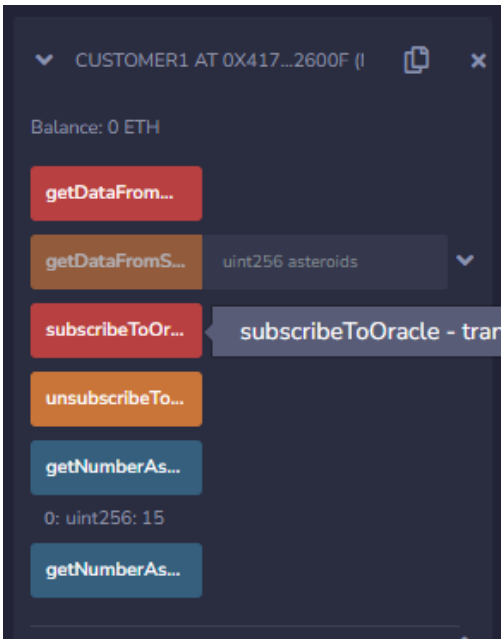

Figura 6.5: *El contrato cliente se suscribe al contrato Publish-Subscribe pagando 5 Ethers*

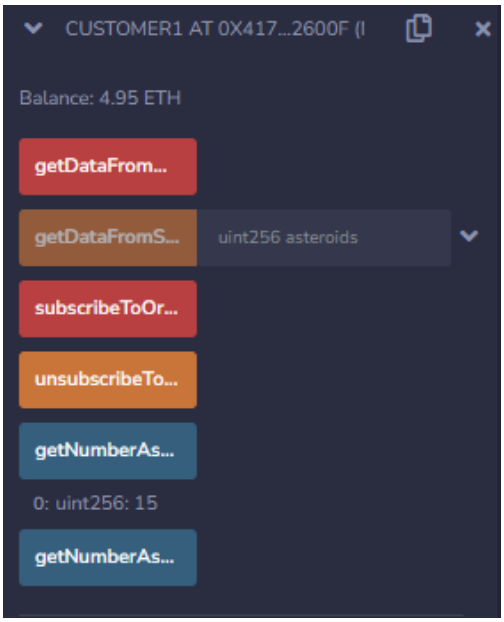

Figura 6.6: *El contrato cliente se desuscribe del contrato Publish-Subscribe y, el oráculo, se queda un 1 % de comisión por la desubscripción como se puede observar en la esquina superior izquierda*

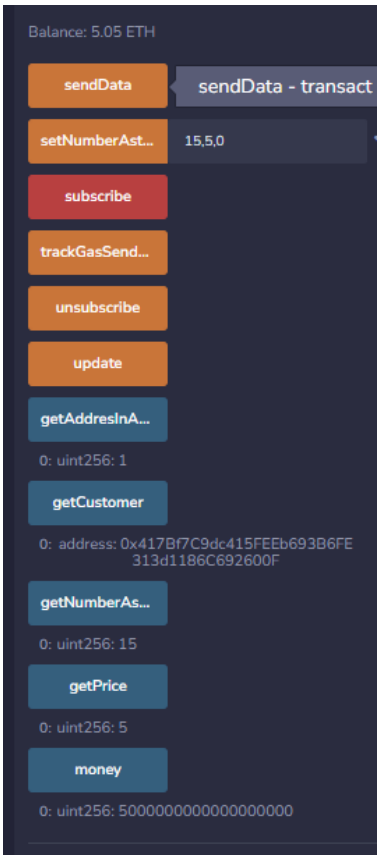

Figura 6.7: *El contrato cliente se vuelve a subscribir, por lo que podemos ver sus datos en la zona inferior izquierda y los fondos del contrato arriba a la derecha, y el contrato Publish-Subscribe procede a envíar el dato a sus subscriptores*

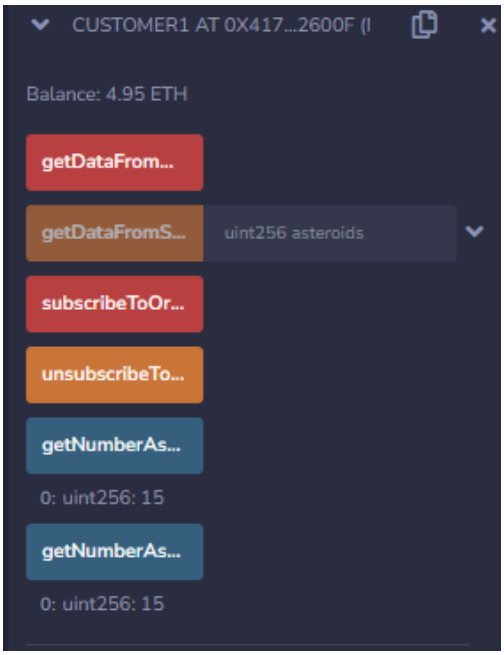

Figura 6.8: *El contrato cliente recibe el dato sin realizar ningún pago*

```
function getDataFromSubsOracle(uint256 asteroids) external{
    require(asteroids == 0, "Este revert fallaria");
    numberAsteroids2 = asteroids;
}
```

Figura 6.9: *Ahora se va a probar en el oráculo Publish-Subscribe a introducir un contrato defectuoso para ver cómo se evita hacer revert al producirse este en el contrato cliente*

```
[vm] from: 0xAb8...35cb2 to: SubsOracle.setNumberAsteroidsAndPrice(uint256,uint256,uint256)
call to BadCustomer1.getNumberAsteroids2
```

Figura 6.10: *Como se puede apreciar la operación no falla gracias a las llamadas de bajo nivel que controlan los reverts por parte de los clientes*

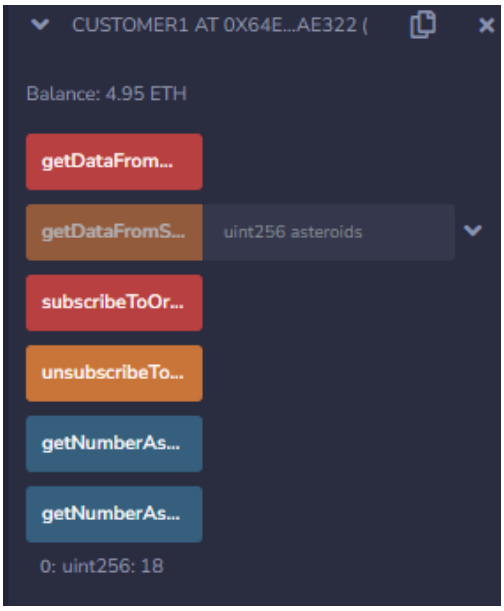

Figura 6.11: *Como se puede observar el dato ha llegado al cliente cuya función estaba bien implementada*

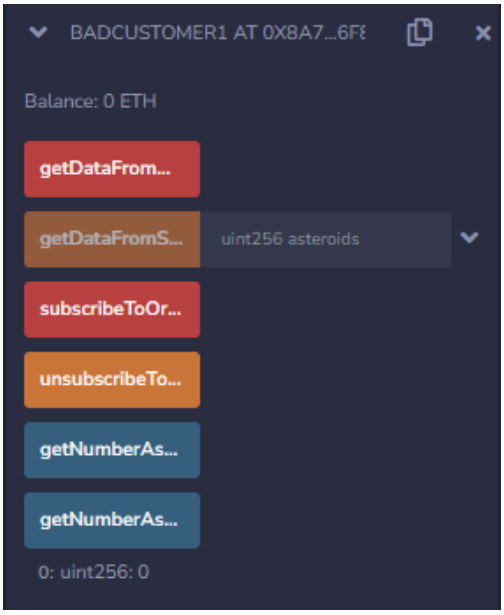

Figura 6.12: *Mientras que al cliente en el que se ha producido el revert no se le ha actualizado el dato*

Las capturas son tomadas de Remix IDE y son la única manera de mostrar los resultados de los contratos tras las operaciones.

Capítulo 7

Configuración de las herramientas

En este capítulo vamos a guiar el proceso para la creación de los prototipo y prueba de los contrtaos.

7.1. Creación del proyecto Truffle

Lo primero es crear un proyecto Truffle de tal manera que podamos manejar los contratos y luego conectarlo con la red Ganache, para ello debemos instalar NodeJS ya que Truffle necesita de `node-gyp` para funcionar. Vamos a utilizar `nvm` para instalarlo:

```
$ curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.
39.1/install.sh
$ curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.
39.1/install.sh | bash
$ source ~/.bashrc
$ nvm list-remote
```

Necesitamos una versión entre la v14 y la v18 ya que es un requerimiento de la propia herramienta. Para ver la lista e instalar la versión que queremos utilizamos:

```
$ nvm list-remote
$ nvm install v18.14.0
```

Para ver que se ha instalado correctamente podemos hacer:`node -version`.

Ahora que Node está instalado pasamos a realizar la instalación de Truffle: `npm install -g truffle`. Y comprobamos que se instala con: `truffle -version`. Tras esto, procedemos a iniciar un proyecto en el directorio deseado:

```
$ cd TFG
$ truffle init
```

A continuación se debe ejecutar el comando:

```
$ truffle init
```

que crea una nueva instalación de truffle. Este comando crea los siguientes directorios y ficheros::

- Directorio "contracts": Como su nombre indica aquí es donde se guardaran los `ficheros.sol` con el código de los diferentes contratos. Además también se crea un contrato `Migrations.sol` el cual Truffle utiliza para tener un control sobre los diferentes despliegues de contratos que se hagan, guardando el número de contratos desplegados.
- Directorio "migrations": En este directorio se encuentran pequeños fragmentos javascript que controlan el despliegue de los distintos contratos. El nombre de estos scripts es importante, ya que se al ejecutarse lo hacen en orden por nombre: 1_, 2_, 3_... Y esto puede influir en las dependencias entre contratos.
- Directorio "test": como su nombre indica sirve para guardar los diferentes test que se creen para los contratos.
- Fichero `truffle-config.js`: este fichero contiene toda la configuración que va a seguir nuestra blockchai: versión del compilador, conexiones con la red blockchain, conexiones con Wallets y bases de datos...

Ahora que tenemos creado el proyecto, ya se pueden empezar a añadir los diferentes contratos que tengamos pensados junto con su respectiva file de migrations.

7.2. Creación de la blockchain

Para conectar nuestro proyecto Truffle con la blockchain local, necesitamos descargarnos Ganche. Descargaremos la aplicación en forma de `.AppImage` desde la página web de Truffle Suit. Una vez que se ha descargado la imagen, procedemos a hacerla ejecutable y para abrir la aplicación.

```
$ chmod +x ganache.AppImage
$ ./ganache.AppImage
```

Abierta la aplicación, hay dos botones: uno para crear una red ethereum de manera rápida y otro para crear un workspace. Hacemos click en crear un nuevo workspace, en nuestro caso se llama¡ TFG. Pide seleccionar el fichero `truffle-config.js` para así poder conectarse con el proyecto que se quiera desplegar en la red. En el siguiente apartado, Server, seleccionaremos la red interna de pruebas `127.0.0.1 - Io`, port number: 7545 , que será donde se exponga la blockchain y por último el network ID que se puede dejar como viene por defecto, ya que

no tenemos la intención de usar varias redes diferentes. El apartado Accounts & keys se deja como viene por defecto y, por último, en el apartado Chain vamos a modificar el valor de gasLimit a 30M, ya que es el límite medio a fecha actual en la red principal de Ethereum.

Una vez preparado Ganache, falta configurar el fichero `truffle-config.js` con el puerto en el que se expone la red para que estén conectados. Para ello bajamos en el fichero hasta llegar a la parte de networks, descomentaremos las líneas de development y cambiaremos el puerto al mismo que hemos puesto en la red Ganache, de tal manera que quede como en la *Figura 7.1*.

```
networks: {
  // Useful for testing. The `development` name is special - truffle uses it by default
  // if it's defined here and no other network is specified at the command line.
  // You should run a client (like ganache, geth, or parity) in a separate terminal
  // tab if you use this network and you must also set the `host`, `port` and `network_id`
  // options below to some value.
  //
  development: {
   host: "127.0.0.1",     // Localhost (default: none)
   port: 7545,            // Standard Ethereum port (default: none)
   network_id: "*",       // Any network (default: none)
  },
```

Figura 7.1:

Además para terminar la configuración del fichero se ha de cambiar la versión del compilador a la: ”0.8.19”, de esta manera nos aseguramos que algunas de las funciones utilizadas en los contratos tienen soporte (*Figura 7.2*).

```
// Configure your compilers
compilers: {
  solc: {
    version: "0.8.19",      // Fetc
    // docker: true,        // Use
    // settings: {          // See
    //  optimizer: {
    //    enabled: false,
    //    runs: 200
    //  },
    //  evmVersion: "byzantium"
    // }
  }
},
```

Figura 7.2:

Tras seguir todos estos pasos y a falta de la creación y despliegue de los contratos, se ha

configurado la blockchain local sobre la que se desplegará la parte on-chain del prototipo.

7.3. Aplicación NodeJS

Creamos un nuevo directorio, en nuestro caso se llamará: `appNode`, e inicializaremos un proyecto `npm` dentro de este:

```
$ cd appNode
$ npm init -yes
```

Ahora falta añadir las dependencias de librerías externas que vamos a utilizar durante el desarrollo del script que sirven para la comunicación entre la API y los contratos. Para ello vamos a instalar 2 dependencias:

- node-fetch: librería que sirve para asegurar una mayor compatibilidad con los navegadores a la hora de hacer peticiones a las diferentes APIs.
- web3: es un conjunto de librerías que sirven para interactuar con nodos Ethereum usando diferentes protocolos como puede ser HTTP o WebSocket.

Podemos hacerlo con el siguiente comando de `npm`:

```
$ npm install node-fetch web3
```

Una vez instaladas todas las dependencias ya se puede proceder a crear el script que va a contener toda la lógica de la aplicación.

Este script de control es el encargado de escuchar los eventos lanzados desde el contrato oráculo de la blockchain, al detectarlos hacer una petición de los datos a las APIs y esperar la respuesta. Una vez recibidos, enviar de nuevo la información al contrato oráculo mediante una transacción. Para todo esto se necesitan varias cosas:

- Especificar el json del contrato que se va a utilizar.
- Especificar la red sobre la que se quiere escuchar los eventos mediante el uso de un objeto de la librería web3. Ya que queremos escuchar los eventos lanzados por los contratos oráculo, deberemos especificar que vamos a utilizar el protocolo WebSocket, ya que http es un protocolo de request-response mientras que WS tiene la ventaja de ser orientado a eventos. Para esto escribiremos `const web3 = Web3('ws://red:puerto')`, creando así la instancia de nuestra red.
- Especificar la dirección de la cuenta de propiedad externa y dirección del contrato.

  ```
  const conInstance = new web3.eth.Contract(conJson.abi, addressContract)
  ```

- Especificar la apiKey.

Una vez ya tenemos todo los datos necesarios procedemos a escribir el script.

7.4. Operaciones de los contratos

Una vez ya tenemos todo el flujo creado, los contratos creados y desplegados, la aplicación está lista para ser usada, en este caso la manera más sencilla es a través de la plataforma web Remix. Una vez conectados desde el navegador, tendremos que copiar el código de nuestros contratos para que Remix los compile y sea capaz de utilizarlos. Tras hacer esto iremos a última pestaña de la barra lateral "Deploy & Run transactions". Para realizar las pruebas con los contratos seguiremos el procedimiento señalado en la *Figura 7.3*:

1. Haremos click en el desplegable titulado "Enviroment" y seleccionaremos: Dev-Ganache provider. Pondremos la dirección en la que está expuesta la red y el puerto, de tal manera que queden conectados.

2. Seleccionaremos el contrato compilado que queremos manejar desde Remix.

3. Ya sea mirando Ganache o el despliegue por consola realizado con Truffle, copiaremos la dirección en la que se ha desplegado el contrato, la pegaremos en el recuadro y haremos click en el botón "At address" para así ya ser capaz de ver los contratos desplegados y manejarlos.

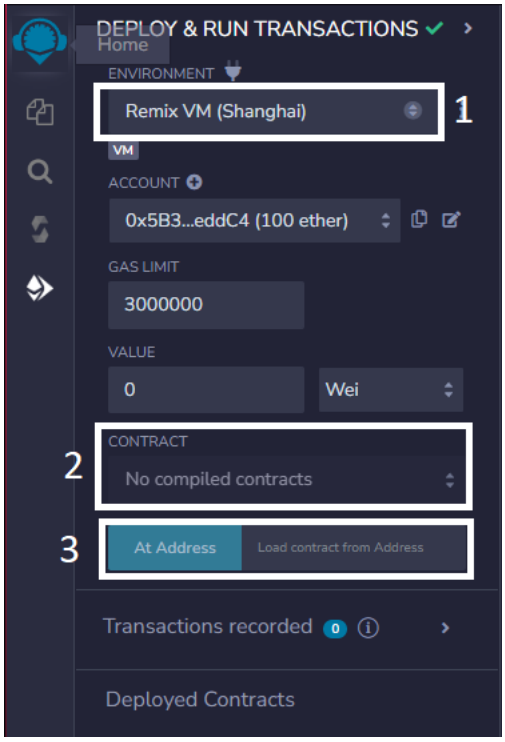

Figura 7.3: *Ventana de despliegue de contratos en Remix*

Capítulo 8

Conclusiones

En el transcurso de este trabajo, hemos explorado a fondo el papel fundamental que desempeñan los oráculos en la tecnología blockchain, su taxonomía, su funcionamiento y cómo su evolución puede tener un impacto significativo en el futuro desarrollo de esta revolucionaria tecnología. Los oráculos, como intermediarios entre el mundo real y los contratos inteligentes, son esenciales para permitir que la blockchain interactúe con datos y eventos externos, lo que amplía enormemente su utilidad y aplicaciones.

Hemos identificado que la confiabilidad y la reproducibilidad son dos desafíos clave que deben abordarse en la implementación de oráculos. La confiabilidad se refiere a la necesidad de garantizar que los datos proporcionados por los oráculos sean precisos y seguros, mientras que la reproducibilidad busca garantizar la integridad de la blockchain mediante resultados consistentes de las consultas. A medida que la tecnología blockchain continúa madurando y su adopción se expande a diversas industrias, la solución de estos desafíos se vuelve aún más crucial.

Además, hemos explorado algunas de las soluciones y proyectos innovadores en el campo de los oráculos, como Chainlink, Provable o Astraea, que buscan abordar estos desafíos y mejorar la fiabilidad y la descentralización de los oráculos desde puntos de vista y propuestas diferentes. Estos proyectos están desempeñando un papel fundamental en la construcción de un ecosistema blockchain más robusto y confiable.

También se han tratado las diferentes taxonomías que estos oráculos pueden presentar, el funcionamiento diferente entre los diferentes diseños y cómo se relacionan con el mundo externo a la blockchain.

Por último el desarrollo de prototipos nos ha acercado a conocer más a fondo los diferentes modelos económicos que se pueden presentar en los oráculos, los problemas de diseño del código y el desafío que afrontan los desarrolladores blockchain sobre todo a la hora de enfrentarse a la seguridad y optimización de los contratos, en un mundo en el que un fallo de este puede tener una gran repercusión económica.

8.1. Problemas encontrados durante el desarrollo

Durante el desarrollo surgieron varios problemas, si bien la mayoría de ellos se pudieron solventar, no con todos fue así. Por eso se realiza esta sección en la que vamos a destacar varios de ellos:

- Trabajo en remoto: la voluntad de los implicados en el trabajo fue trabajar en remoto en un equipo prestado. Como se menciona en un punto de la memoria esto no fue posible debido a problemas de configuración del equipo. Esto llevó más tiempo del esperado y por problemas de interacción entre el sistema operativo y el hardware del equipo se tuvo que abandonar la idea.

- Cambio de herramientas software: al principio el desarrollo se planteó utilizando otras herramientas, pero debido a la mala interacción de estas con el antivirus del primer sistema operativo planteado (Windows), se tuvo que cambiar a Ubuntu 22.04.

- Pérdida del equipo de trabajo: durante el desarrollo, el equipo donde finalmente se desarrolló el trabajo dejó de funcionar. Esto representó un gran problema y retraso ya que se tuvo que esperar a la reparación de este para el desarrollo del prototipo.

- Llamadas de bajo nivel en los contratos: errores durante el desarrollo mediante el uso de llamadas de bajo nivel dificultan la depuración del código en gran medida. Esto es debido a que en caso de producirse un revert en la función a la que se ha realizado la llamada, el contrato que la ha realizado no sabrá en qué momento se ha producido ya que actúa como caja negra. A esto se le añade la dificultad que representa la depuración de código Solidity, debido a que hay que trabajar a nivel de bytecode.

- Falta de información: este es un problema generalizado sobre el desarrollo actual de la en trabajos específicos sobre una parte de la blockchain. Esto se refiere a que al ser una tecnología tan reciente, hay escasez de artículos de calidad científicos y académicos sobre temas concretos como puede ser el desarrollo de oráculos.

8.2. Trabajo futuro

En esta sección nos gustaría presentar ideas que quedan abiertas y pueden extender el presente trabajo sobre los oráculos en la blockchain:

- Implementación de una aplicación DeFI: la posible implementación de una aplicación DeFI que haga uso de oráculos para la comunicación con el mundo exterior.

- Implementeción de Request-Response: es el único oráculo que no se ha implementado porque creíamos fuera del alcance de este trabajo, pero para futuros trabajos un análisis específico de este oráculo junto con una implementación parece posible.

- Implementación de diferentes estándares en nuevos prototipos: es posible realizar la implementación de diferentes estándares como el ERC-165 [15],que, más allá de la utilización de interfaces dentro de un contrato Solidity, permite la verificación externa de que los contratos con a los que se quiere realizar una llamada implementan esas interfaces.

- Estudio de diferentes modelos económicos a los propuestos: el estudio en profundidad de modelos económicos para rentabilizar oráculos es un campo extensible y que varía según los oráculos que se quieran investigar. Esto representa un campo en el que se puede expandir mucho más los conocimientos aportados por este trabajo.

En resumen, los oráculos en la blockchain desempeñan un papel fundamental en la construcción del futuro tecnológico. Su mejora constante y su adopción generalizada facilitarán el camino para el desarrollo de un mundo impulsado por la confianza, la transparencia y la automatización, redefiniendo la forma en que interactuamos con la información y los contratos en la era digital.

Capítulo 9

Introducction

Due to the nature of blockchain systems, where all information must be verifiable and reproducible to maintain a trust-based network among nodes, introducing information from external systems is not trivial. This information can be false either because it is extracted from a compromised source or for malicious purposes, meaning the owner intends to manipulate the blockchain. Furthermore, as this information is external, it cannot be verified within the network itself, making it untrustworthy information. To address this issue, the concept of "Oracles" is introduced, which are pieces of code designed to facilitate the connection between blockchain networks and the external world, including applications, scripts, databases, or even other blockchains.

The main objective of this undergraduate thesis is to conduct an in-depth study of these Oracles and their functioning, keeping in mind specific goals:

- State of the Art: To determine the current state of the development of oracles by various companies in the blockchain network sector, understanding their development, and, if possible, conducting a brief analysis of how these technologies function. In addition to companies, academic proposals in this area will also be considered.
- Taxonomy: To comprehend the composition of oracles through different articles and available information sources, with the aim of categorizing oracles based on various characteristics, such as:
    - Data Source: Whether the data originates from software, hardware, or human sources.
    - Proposed Trust Model: Whether it follows a centralized model with a single oracle or a decentralized model with a network of oracles that must reach consensus.
    - Design Pattern: Referring to the chosen model for interactions between clients and contracts with the oracle, primarily resulting in three models: Immediate-Read, Publish-Subscribe, or Request-Response.

    - Interaction between systems: refers to the flow of data between the blockchain system and the external system, i.e., whether data is sent from the blockchain to the external system or vice versa.

- Prototype Development: To create small-scale prototypes of some oracle models on an Ethereum network in order to understand their functioning, the security required for their code against various blockchain attacks, optimization measures and different economic models that can be implemented to make oracles profitable in their various designs.

The work plan agreed upon with the director consists of:

- Weekly one-hour meetings.
- A first phase that involves gathering various articles, books, etc., to conduct a subsequent analysis of the content and develop both the thesis and the prototypes.
- Preparing the working environment, including software tools such as code editors, Solidity development environments, local blockchain network, testing environment, etc. And preparing the equipment for use, such as: remote connection setup, system OS cleaning, and clean installation.
- Developing the prototypes based on the collected information.
- Studying the results obtained during execution and analyzing the created prototypes.
- Presenting the final conclusions of the work.

Capítulo 10

Conclusions

Throughout the course of this thesis, we have extensively explored the pivotal role that oracles play in blockchain technology, their taxonomy, their operation, and how their evolution can have a significant impact on the future development of this revolutionary technology. Oracles, as intermediaries between the real world and smart contracts, are essential in enabling blockchain to interact with external data and events, greatly expanding its utility and applications.

We have identified that reliability and reproducibility are two key challenges that must be addressed in the implementation of oracles. Reliability pertains to the need to ensure that the data provided by oracles is accurate and secure, while reproducibility aims to ensure blockchain integrity through consistent query results. As blockchain technology continues to mature and its adoption expands across various industries, resolving these challenges becomes even more crucial.

Furthermore, we have explored some of the innovative solutions and projects in the field of oracles, such as Chainlink, Provable, or Astraea, which seek to address these challenges and enhance the reliability and decentralization of oracles from different perspectives and proposals. These projects play a fundamental role in building a more robust and trustworthy blockchain ecosystem.

We have also delved into the different taxonomies that these oracles can exhibit, the varying operation among different designs, and how they relate to the external world beyond the blockchain.

Finally, the development of prototypes has provided us with a deeper understanding of the different economic models that can be found in oracles, the code design issues, and the challenges faced by blockchain developers, especially in terms of security and contract optimization, in a world where a failure in this regard can have significant economic repercussions.

10.1. Proublems found during development

During the development process, several issues arose, and while most of them were resolved, not all could. Therefore, this section highlights some of these challenges:

- Remote Work: The initial intention was for all team members to work remotely using borrowed equipment. As mentioned in a section of the report, this plan was not possible due to equipment configuration issues. This led to more time being spent than anticipated, and due to compatibility problems between the operating system and the hardware, the remote work idea had to be abandoned.
- Change in Software Tools: Initially, the development was planned using different software tools, but these tools had compatibility issues with the antivirus software on the first chosen operating system (Windows). Consequently, a switch to Ubuntu 22.04 became necessary.
- Loss of Work Equipment: During the development process, the equipment where the work was being done stopped functioning. This represented a significant problem and delay as the repair of the equipment had to be awaited before the prototype development could continue.
- Low-Level Calls in Contracts: Errors encountered during development when using low-level calls significantly difficulted code debugging. This is because in the event of a revert in the function being called, the contract that initiated the call would not know when it occurred since it acts as a black box. Additionally, debugging Solidity code is challenging as it involves working at the bytecode level.
- Lack of Information: This is a widespread issue in current blockchain development, particularly in blockchain-related projects. It pertains to the lack of high-quality scientific and academic articles on specific topics, such as oracle development, because blockchain technology is relatively new, comprehensive research in these areas is limited.

10.2. Proposals for future developement

In this section, we would like to present open-ended ideas that can extend the present work on oracles in the blockchain:

- Implementation of a DeFi Application: Consider the possibility of implementing a decentralized finance (DeFi) application that utilizes oracles for communication with the external world. This could involve creating a DeFi project that relies on real-world data provided by oracles to make financial decisions.
- Implementation of Request-Response: Although not included in this work due to perceived complexity, a specific analysis and implementation of the Request-Response oracle could be explored in future research. This would involve a detailed examination of how this particular type of oracle operates within the blockchain ecosystem.

- Implementation of Different Standards in New Prototypes: Explore the implementation of various standards, such as ERC-165, which goes beyond the use of interfaces within a Solidity contract and allows for external verification that contracts implement specific interfaces. This could enhance interoperability and compatibility among different blockchain components.
- Study of Different Economic Models: Conduct an in-depth study of economic models for making oracles profitable. This is a highly expandable field, with variations depending on the specific oracles under investigation. Further research in this area could significantly contribute to our understanding of the economics of oracles in blockchain.

In summary, oracles in the blockchain play a fundamental role in shaping the future of technology. Their continuous improvement and widespread adoption will pave the way for the development of a trust-driven, transparent, and automated world, redefining how we interact with information and contracts in the digital era.

Bibliografía

[1] 51https://academy.binance.com/en/glossary/51-percent-attack.

[2] Decentralised oracle. https://research.csiro.au/blockchainpatterns/general-patterns/interacting-with-the-external-world/decentralized-oracles/.

[3] Sybil attack. https://academy.binance.com/en/articles/sybil-attacks-explained.

[4] Provable documentation, 2019. https://docs.provable.xyz/#home.

[5] Chainlink documentation, 2023. https://docs.chain.link/architecture-overview/architecture-overview.

[6] Chainlink educational documentation, 2023. https://chain.link/education/blockchain-oracles#solving-the-oracle-problem.

[7] Ethereum accounts documentation, 2023. https://ethereum.org/es/developers/docs/accounts/.

[8] Ethereum gas documentation, 2023. https://ethereum.org/es/developers/docs/gas/.

[9] Ethereum oracles documentation, 2023. https://ethereum.org/es/developers/docs/oracles/.

[10] Etherscan. https://etherscan.io, 2023.

[11] John Adler, Ryan Berryhill, Andreas Veneris, Zissis Poulos, Neil Veira, and Anastasia Kastania. Astraea: A decentralized blockchain oracle. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 1145–1152, 2018.

[12] Hamda Al-Breiki, Muhammad Habib Ur Rehman, Khaled Salah, and Davor Svetinovic. Trustworthy blockchain oracles: Review, comparison, and open research challenges. *IEEE Access*, 8:85675–85685, 2020.

[13] Michael Bartholic, Aron Laszka, Go Yamamoto, and Eric W. Burger. A taxonomy of blockchain oracles: The truth depends on the question. In *2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–15, 2022.

[14] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform., 2014.

[15] Fabian Vogelsteller Jordi Baylina Konrad Feldmeier Christian Reitwießner, Nick Johnson and William Entriken. Erc-165: Standard interface detection, 2018. https://eips.ethereum.org/EIPS/eip-165.

[16] Beaver Finance. Defi security lecture 7 —price oracle manipulation, 2022. https://medium.com/beaver-smartcontract-security/defi-security-lecture-7-price-oracle-manipulation-d716cdeaaf77.

[17] Jeff Hu. Decentralized oracle: Bridging blockchains with the outside world, 2018. https://medium.com/3-minutes-blockchain-paper/3-decentralized-oracle-bridging-blockchains-with-the-outside-world-47cf2373d336.

[18] kushalmevada. Blockchain oracle: Types, uses and how it works, 2023. https://www.geeksforgeeks.org/blockchain-oracle-types-uses-and-how-it-works/.

[19] Jan Ladleif and Mathias Weske. Which Event Happened First? Deferred Choice on Blockchain Using Oracles. *Frontiers in Blockchain*, 4:758169, 2021.

[20] B. Liu, P. Szalachowski, and J. Zhou. A first look into defi oracles. In *2021 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*, pages 39–48, Los Alamitos, CA, USA, aug 2021. IEEE Computer Society.

[21] Matías Luis Lotito Ralli and Jesús Correas Fernández. Es la tecnología blockchain el futuro del voto? is blockchain technology the future of voting?

[22] Jesús Santaella. ¿qué es chainlink (link) y cómo es su crecimiento y perspectivas?, 2022. https://economia3.com/que-es-chainlink-link-plataforma-criptomoneda/.

[23] Rohit Sutar. Solidity abi encode and decode, 2023. https://coinsbench.com/solidity-abi-encode-and-decode-b339eb52c5b5.

[24] Julien Thevenard. Decentralise oralces: a comprehensive overview, 2019. https://medium.com/fabric-ventures/decentralised-oracles-a-comprehensive-overview-d3168b9a8841.