



TRABAJO FIN DE GRADO
INGENIERÍA EN INFORMÁTICA

Blockchain y Smart Contract Travel

Sistema de integridad de certificados sanitarios y billetes de avión para viajar en tiempos de pandemia basado en contratos inteligentes sobre la cadena de bloques de Ethereum

Autor

Jose Luis Pedraza Román

Directores

Jose Manuel Soto Hidalgo



Escuela Técnica Superior de Ingenierías Informática y de
Telecomunicación

—
Granada, Agosto de 2021



Blockchain y Smart Contract Travel

Sistema de integridad de certificados sanitarios y billetes de avión para viajar en tiempos de pandemia basado en contratos inteligentes sobre la cadena de bloques de Ethereum



ugr

Universidad
de **Granada**

Autor

Jose Luis Pedraza Román

Directores

Jose Manuel Soto Hidalgo

Blockchain y Smart Contract Travel:
Sistema de integridad de certificados sanitarios y billetes de avión para viajar en tiempos de pandemia basado en contratos inteligentes sobre la cadena de bloques de Ethereum

Jose Luis Pedraza Román

Palabras clave: Blockchain, Ethereum, cadena de bloques, Solidity

Resumen

Con este TFG Se pretende crear un sistema que permita asegurar la integridad de todos los pasajeros de un vuelo en tiempos de pandemia controlando la autenticidad de los certificados emitidos por una autoridad sanitaria en relación a un test negativo en los 3 días previos al vuelo de cada pasajero. Cuando un usuario compre un billete de avión, la compañía/aerolínea deberá asegurarse de que todos los pasajeros tengan un certificado sanitario con resultado negativo en la prueba del covid19 emitido en las últimas 72 horas antes del despegue de su vuelo. Para garantizar la integridad de los pasajeros y certificados sanitarios se hará uso de la cadena de bloques de Ethereum mediante la implementación de un contrato inteligente entre la compañía/aerolínea y el centro sanitario como actores principales y el pasajero como actor secundario. Las acciones del pasajero simplemente se reducirían a comprar el billete y proporcionarle a la aerolínea los datos del centro sanitario donde se realizará la prueba, para que esta los corrobore y envíe posteriormente por correo electrónico la dirección del contrato.

Asimismo, se plantea la opción de modificar el contrato inteligente si el pasajero se dirige con síntomas a un centro sanitario con un certificado previo emitido por un centro sanitario emisor que le habilitara para el viaje en las últimas 72 horas.

**Blochain y Smart Contract
Travel:
Integrity system for health
certificates and airline tickets
for travel in times of
pandemic based on smart
contracts on the Ethereum
Blockchain**

Jose Luis Pedraza Román

Keywords: Blockchain, Ethereum, Smart Contract, Solidity

Abstract

With this final degree project, the aim is to create a system that ensures the integrity of all passengers on a flight in times of pandemic by controlling the authenticity of the certificates issued by a health authority in relation to a negative test in the previous three days to the flight of each passenger. When a user buys a plane ticket, the company/airline must ensure that all passengers have a health certificate with a negative result in the Covid19 test issued in the last 72 hours before the take-off of their flight. To guarantee the integrity of passengers and health certificates the Ethereum Blockchain will be used through the implementation of a smart contract between the airline company and the health center as main actors and the passenger as secondary actor. The passenger's actions would simply be reduced to buying the ticket and providing the airline with the data of the health center where the test is carried out, so that it can corroborate them and later send the contract address by email.

Likewise, the option of modifying the smart contract is proposed if the passenger goes with symptoms to a health center with a previous certificate issued by an issuing health center that will enable him to travel in the last 72 hours.

Yo, **Jose Luis Pedraza Román**, alumno de la titulación Grado en Ingeniería Informática de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 80108849-X, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Jose Luis Pedraza Román



Granada a 30 de agosto de 2021.

D. **Jose Manuel Soto Hidalgo**, Profesor del Departamento de **Arquitectura y Tecnología de Computadores** de la Universidad de Granada.

Informan:

Que el presente trabajo, titulado ***Blockchain y SmartContract Travel, Sistema de Integridad de certificados sanitarios y billetes de avión para viajar en tiempos de pandemia basado en contratos inteligentes sobre la cadena de bloques de Ethereum***, ha sido realizado bajo su supervisión por **Jose Luis Pedraza Román**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 30 de agosto de 2021.

El director:



Jose Manuel Soto Hidalgo

Agradecimientos

Llegando al final de esta gran etapa que ha sido la Universidad, lo primero de todo, agradecer a mis padres el apoyo incondicional, la confianza y las fuerzas que me han dado durante todos estos años que he estado fuera de casa, y sobre todo por su enorme esfuerzo realizado para poder proveerme de la gran oportunidad que son unos estudios superiores de este nivel, os estaré tremendamente agradecido siempre.

También agradecer a todos mis compañeros y amigos que durante la carrera hemos sido fuentes de apoyo entre nosotros, ha sido un placer tener la oportunidad de conocer a gente tan buena y válida durante estos años. Con mención especial a Iván López Justicia y Pablo Jesús Martínez Ramírez, con quienes surgió el interés recíproco y comencé a sumergirme en el mundo del Blockchain en uno de los trabajos de una asignatura que cursamos juntos.

Por supuesto, agradecer a mi tutor del trabajo de fin de grado Jose Manuel Soto Hidalgo por aceptar mi propuesta sin dudarlo e intentar estar siempre a mi lado, aunque las circunstancias no han sido las más idóneas en este curso académico tan raro. Muchísimas gracias por la confianza depositada y por tu atención inmediata.

Por último, debo agradecer también a la empresa Firsap Sistemas S.L, en donde realicé las prácticas de la carrera, por su comprensión e interés porque terminase a tiempo el trabajo de fin de grado, otorgándome el tiempo necesario para ello y así poder seguir trabajando con ellos, una vez finalizados mis estudios.

Índice general

1. Introducción

- 1.1. Motivación
- 1.2. Finalidad
- 1.3. Estructura

2. Conceptos generales sobre la tecnología

- 2.1. Marco histórico
- 2.2. ¿Qué es Blockchain?
- 2.3. ¿Cómo funciona Blockchain?
- 2.4. Sumario
- 2.5. Debilidades

3. Ethereum

- 3.1. Historia y características
- 3.2. Funcionamiento
- 3.3. Herramientas para el desarrollo
- 3.4. Dapps, NFTs y más

4. Smart Contract Travel

- 4.1. Definición y objetivos
- 4.2. Implementación
 - 4.2.1. Desarrollo y análisis del código
 - 4.2.2. Despliegue con Remix I
 - 4.2.3. Despliegue con Remix II
- 4.3. Integración web
- 4.4. Ejemplo de uso DApp
- 4.5. Especificaciones mínimas equipo desarrollo
- 4.6. Planificación temporal
- 4.7. Líneas futuras

5. Conclusiones

6. Bibliografía

1. Introducción

1.1. Motivación

Con motivo de la pandemia de Covid19, mi creciente interés en la tecnología Blockchain y mis ganas de viajar, se me ocurrió la idea de profundizar y explicar en qué se basa esta tecnología y cómo podría aplicarse a un proyecto real y actual como es un pasaporte especial para estos tiempos de pandemia que permita viajar a la población de la manera más segura posible sin la necesidad de que se encuentren completamente vacunados, dado que lo importante del mismo es la certificación de que el viajero se encuentra libre de Covid19 mediante alguna de las pruebas existentes para ello.

1.2. Finalidad

Entender el funcionamiento de la tecnología Blockchain y la revolución que esta supone, así como proponer una solución viable que aplique esta tecnología contra la creciente falsificación y venta de certificados de test negativos del Covid19 para garantizar la seguridad de todos y cada uno de los viajeros de los vuelos de una compañía aérea. Para garantizar la integridad de los pasajeros y certificados sanitarios se hará uso de la cadena de bloques de Ethereum mediante la implementación y despliegue de un contrato inteligente entre la aerolínea y el centro sanitario como actores principales y el pasajero como actor secundario.

1.3. Estructura

Este trabajo se divide en varios puntos en los cuales se explican detalles de la tecnología y su aplicación en el caso práctico del Smart Contract Travel propuesto.

Punto 1. Introducción

En este punto introductorio se muestra la motivación y la finalidad que me han llevado a proponer este tema para el trabajo que nos ocupa. También un resumen de la estructura del mismo.

Punto 2. Conceptos generales sobre la tecnología

Punto en el cual se establece un marco histórico de la tecnología blockchain explicando varios conceptos básicos de la misma, así como una descripción profunda acerca de qué es el blockchain, las bases sobre las que se establece, su funcionamiento y las posibles debilidades y problemas con los que se puede encontrar.

Punto 3. Ethereum

El tercer punto del trabajo se centra en la tecnología usada para el desarrollo de los puntos posteriores, explicando la historia y las características generales de la misma, así como el funcionamiento y los diferentes tipos de redes que nos

podemos encontrar. También se explican algunas de las herramientas de desarrollo de que dispone Ethereum y una serie de aplicaciones interesantes del paradigma actual de dicha tecnología.

Punto 4. Smart Contract Travel

En esta parte se define el caso práctico propuesto y se establecen los objetivos, así como el diseño, el desarrollo, la implementación y explicación del funcionamiento de las herramientas utilizadas para este fin. También podremos encontrar un ejemplo de uso para tener una idea del funcionamiento real de esta propuesta y una serie de conclusiones, posibles mejoras y líneas futuras.

Punto 5. Bibliografía

Punto donde se podrán encontrar todas las referencias bibliográficas que han sido necesarias para el desarrollo de este trabajo.

2. Conceptos generales sobre la tecnología

2.1. Marco histórico

Hasta que surgió la tecnología Blockchain a finales del siglo XX, para llevar a cabo operaciones transaccionales a través de Internet, era necesaria la intervención de un mediador entre las partes intervinientes en la operación, a fin de validar y dotar de seguridad dicha transacción.

Este intermediario debía tener la confianza de ambas partes; sin embargo, de esta posición central, en ciertas ocasiones, se hace un uso abusivo, a fin de favorecer el propio interés de alguna de las partes intervinientes. Un claro ejemplo de esta influencia ocurrió en 2010 cuando el portal de pago PayPal cerró la cuenta de la plataforma WikiLeaks. [1]

La primera aproximación a la tecnología Blockchain fue creada en 1991 por los científicos Stuart Haber y W.Scott Stornetta al introducir una solución computacionalmente práctica para los documentos digitales dotándolos de un sello de tiempo para que no pudieran ser modificados o manipulados.

La tecnología Blockchain fue propuesta en 1997 por un informático y ex profesor de derecho Nick Szabo como solución a la falta de confianza entre los usuarios anónimos que realizaban intercambios de datos en Internet.

En el año 2008 un programador, o grupo de programadores, bajo el alias de Satoshi Nakamoto propuso la idea de Bitcoin, que en enero de 2009 se convirtió en la primera criptomoneda descentralizada del mundo. Esta moneda es completamente revolucionaria, dado que realmente es un software que por primera vez en la historia permite hacer transferencias de valor entre un par de personas en cualquier parte del mundo sin recurrir a una entidad centralizada como puede ser un banco, un gobierno o una compañía.

Esta idea revolucionaria resolvió simultáneamente y por primera vez en la historia, dos problemas fundamentales, que son:

1. El problema del tercero confiable: No hay necesidad de que existan intermediarios porque todas las transacciones son anotadas en tiempo real en un registro público 100% editable y que está distribuido de forma abierta y voluntaria en Internet. Cuando una computadora de esa red confirma un pago y realiza un cambio en su registro, al mismo tiempo se actualizan todos los demás registros distribuidos alrededor del mundo.
2. El problema del doble gasto: Basado en la idea de digitalizar transacciones de valor entre personas físicas. Por ejemplo, si yo te doy un objeto físico, la transacción termina ahí, yo dejo de tener ese objeto porque ya lo tienes tú. Pero para verificar en un caso digital que un archivo virtual es único e irrepetible debemos llevar un registro para verificar quién tiene qué, lo que se denomina “libro de contabilidad”.

2.2. ¿Qué es Blockchain?

Concepto

En un sentido estricto Blockchain es un sistema de almacenamiento de información digital. Dicha información puede tener un contenido muy variado, pudiendo abarcar literalmente cualquier cosa, por ejemplo, para Bitcoin, son transacciones con contenido económico (transferencias de valor de una cuenta a otra), pero pueden ser incluso archivos de cualquier tipo. La información está almacenada en forma de bloques, que están encadenados usando funciones criptográficas.

El valor añadido de esta tecnología radica en el modo en que la información es almacenada y agregada en la cadena de bloques, que produce una serie de características muy beneficiosas para el sistema:

- Invariabilidad de la historia.
- Invulnerabilidad del sistema.
- Persistencia de la información.
- No hay un solo punto de falla gracias a la descentralización.

Definición

A grandes rasgos, una definición de Blockchain sería: *Aquel registro compartido y digitalizado que no puede modificarse una vez que una transacción ha sido registrada y verificada. Todas las partes intervinientes en la transacción, así como un número significativo de terceros, mantienen una copia del registro, o lo que es lo mismo, de la cadena de bloques, lo que significa que sería prácticamente imposible modificar cada una de las copias del registro distribuido globalmente para falsificar una transacción.*

Una definición algo más técnica de Blockchain: *Es una estructura de datos descentralizada basada en una concatenación de bloques de datos ampliable en todo momento. Cada uno de estos bloques está vinculado al bloque anterior y al posterior por una función criptográfica, como si se tratara de los eslabones de una cadena, registrando transacciones entre pares en orden cronológico e irreversible y garantizando que se puedan añadir nuevos bloques sin sustituir ni modificar los anteriores. Es por esto que con el paso del tiempo y a mayor número de transacciones, la cadena de bloques será más y más larga, resultando así imposible modificar o eliminar la información almacenada, cuya trazabilidad está asegurada.*

Consenso

Blockchain debe entenderse como una **base de datos descentralizada** en la que cada participante tiene una copia completa de dicha base de datos (todos los participantes tienen una copia de la cadena de bloques). Para añadir una modificación (transacción) en esta base de datos es necesario que todos los participantes validen dicha transacción, aquí entra en juego el concepto de consenso.

Este **consenso** se lleva a cabo a través de una forma definida en la propia cadena de bloques, por ejemplo, el 51% de los participantes deben aceptar dicha transacción, corroborando que es correcta y validando su inclusión en la cadena.

El consenso en la red Blockchain hace referencia al proceso de lograr un acuerdo entre los distintos participantes de la red sobre las transacciones que van a escribirse en la cadena de bloques. Este consenso es el responsable de que no existan discrepancias, es decir, que todos los nodos de la red tengan los mismos datos, evitando así su manipulación.

La forma de implementar este proceso en la red es lo que se conoce como método o algoritmo de consenso (el cual veremos en puntos posteriores con mayor profundidad) y forma parte del núcleo de la misma red, aunque en algunas implementaciones de redes Blockchain se permite seleccionar este algoritmo entre varios disponibles.

Para entender este concepto mejor, imaginemos un comercio on-line de segunda mano en el que alguien desea vender un objeto que ya no utiliza y por tanto lo anuncia en una plataforma de venta de segunda mano destinada a este fin. Otra persona deseará comprarlo por lo que se establecerá un precio de venta. Estas dos personas viven en lugares distintos, de modo que el intercambio “en mano” no puede realizarse. Ambas partes se muestran reticentes y no confían en la otra parte. El vendedor no enviará el producto hasta tener el dinero, pero el comprador no pagará sin tener el producto, por miedo a que la otra parte se retracte provocando la pérdida del dinero o el producto.

Una solución a este problema podría ser el uso de un mediador, en el cual creemos y que se encargará de realizar tanto la entrega del producto como del dinero a sus respectivos destinatarios a cambio de cobrar una contraprestación por la transacción.

Aquí es donde Blockchain nos ofrece una alternativa al uso de mediadores externos, dado que estas transacciones en redes Blockchain no necesitan ser verificadas por terceros, ahorrándonos así los costes de gestión y provocando que el intercambio será más seguro y confiable ya que todo registro de una transacción queda guardado en un historial inmodificable donde sólo se puede añadir información nueva, quedando registros de todo cambio distribuidos y actualizados en todos los nodos de la red. Esto es denominado **libro contable distribuido (distributed ledger)**.

Token, Criptomoneda y Smart Contract

Un **token** es históricamente entendido como una ficha, vale o pseudo moneda que es utilizado como sustituto de una moneda real dentro del espacio para el que están concebidos. Un ejemplo claro de token podrían ser las fichas de un casino, que sustituyen al dinero real, pero son válidas únicamente dentro del propio casino y solo tienen validez siempre que este casino esté en funcionamiento.

En la historia se han utilizado tokens en diversas ocasiones y por motivos similares. En Hispania y también en la época colonial, los tokens aparecieron para suplir la escasez de moneda oficial. [5]

La diferencia fundamental entre un token y una moneda es que una moneda (o divisa) es emitida por una autoridad local o nacional y es de libre cambio de bienes u otras monedas, mientras que un token tiene un uso mucho más limitado y es a menudo emitido por una empresa privada, institución, grupo, asociación o persona, para su utilización exclusivamente dentro del ámbito al que se refieren, por lo que no son dinero de curso legal. [6]

Todas las **criptomonedas** son por definición tokens. Son una unidad de valor, emitida por una entidad privada, que tiene el valor que se le otorga dentro de una determinada

comunidad. Un token en el contexto del Bitcoin puede ser cualquier cadena alfanumérica que represente un registro en la base de datos descentralizada de consenso de Bitcoin, por ejemplo, la clave pública: “3J98t1WpEZ73CNmQviecnyiWrnqRhWNLY” es un token. Por tanto, un token es una representación, dentro de esa base de datos que es Blockchain, de un activo o de una propiedad que, a día de hoy las entidades, como bancos, empresas o gobiernos, gestionan.

Los **contratos inteligentes** (smart contracts en inglés) eran una de las grandes promesas de las blockchain, ya que permiten un nuevo paradigma que va mucho más allá de las transferencias de dinero, y es que, Bitcoin y la tecnología de la cadena de bloques (Blockchain) hacen posible que, por primera vez, se pueda implementar este concepto, cuya teoría fue desarrollada hace casi veinte años, y que no ha sido potencialmente viable hasta la invención de Bitcoin, ya que es necesario el uso de **dinero programable**.

Alex Puig, CEO de Alastria, una asociación sin ánimo de lucro que fomenta la economía digital a través del desarrollo de tecnologías de registro descentralizadas/blockchain, explica muy bien estos conceptos con el siguiente ejemplo: “Si tengo una flota de vehículos podría enviarles litros de gasolina, así no tendría que darles tarjetas de crédito, ni pedirles tickets. Les doy los litros y los programo para que sólo se puedan gastar en un horario determinado, en una zona y con un vehículo con una matrícula concreta. Realmente, afecta a todos los sectores. Y no necesitas que nadie lo certifique, lo certifica la tecnología porque Blockchain es un libro de contabilidad”. [4]

En este ejemplo, los litros de gasolina serían los tokens y el smart contract sería el “programa” que limita, condiciona y automatiza el uso de dichos tokens.

Los contratos inteligentes son códigos de programación autónomos y repetibles que se ejecutan en la cadena de bloques (Blockchain) que representan promesas unilaterales de proporcionar una tarea informática determinada. Se almacenan en una dirección específica en la cadena de bloques. Dicha dirección se determina cuando los contratos son compilados y enviados a la cadena de bloques. Cuando se produce un evento contemplado en el contrato, se envía una transacción a esa dirección y la máquina virtual distribuida ejecuta los códigos de operación del script (o cláusulas) utilizando los datos enviados con dicha transacción.

Los contratos inteligentes pueden estar codificados de modo que reflejen cualquier tipo de lógica basada en datos, buen ejemplo de ello sería, gestión de préstamos, depósitos en garantía, controles de gasto, herencias y donaciones, etc. [7]

Tipos de Blockchain: no permissionada, permissionada e híbrida

Con el tiempo se han ido desarrollando varios tipos o categorías de Blockchain orientados a diferentes casos de uso de esta tecnología:

- No permissionada o pública:

Fue el primer tipo en aparecer, son las referentes a aquellas a las que tenemos fácil acceso a través de internet, como el ya mencionado Bitcoin o Ethereum, con un simple registro en la plataforma y la descarga del software necesario. Estas Blockchain tienen todos los datos, software y desarrollo abiertos al público, de modo que cualquiera puede supervisar, extender u optimizar dicho software.

Esta misma razón acentúa la necesidad de implantar medidas de seguridad para prevenir cualquier tipo de ataque, aquí es donde cobra relevancia la tolerancia a fallas bizantinas, fuertes protocolos de consenso, protección contra ataques DDos o contra ataques de 51 %.

Las características de estas redes son:

- Cualquier persona puede acceder e interactuar en la red, bien como un mero usuario, un minero o administrador de un nodo y todo ello sin ningún tipo de limitación.
- Estas redes funcionan de manera abierta a terceros y todos sus datos pueden ser conocidos sin limitaciones desde el principio de la cadena, permitiendo que cualquiera pueda verificar el funcionamiento de dicha cadena de bloques.
- Las redes públicas se rigen por el principio de descentralización, es decir, sin que haya una autoridad centralizada reguladora encargada de supervisar el funcionamiento de la misma.
- El sistema integrado en estas redes determinará su gestión y sustento económico, consistiendo en la mayoría de los casos en el cobro de comisiones por transacción realizada, incentivando de este modo a los nodos mineros.

▪ Permisionada o privada:

Una vez las Blockchain fueron desarrollándose, muchas empresas comenzaron a interesarse por esta tecnología. Tanto éstas como las no permisionadas comparten los mismos elementos, con la diferencia de que en las permisionadas existe una entidad central que controla la actividad de la misma. Esta unidad central es la que da acceso a los usuarios, controlando sus funciones y permisos. Usualmente, con opciones de tipo software privativo, aunque con la existencia de software libre. Una de las Blockchain permisionada más importante es Hyperledger, proyecto iniciado por la Fundación Linux y varias empresas del sector tecnológico. Es el mayor ejemplo de Blockchain privada. [29]

Características de las Blockchain permisionadas:

- Restringido acceso a la red a elementos que solo pueden ser autorizados por la entidad central.
- El acceso al libro de transacciones o cualquier otra información del Blockchain es privado.
- El mantenimiento económico de la Blockchain depende generalmente de la empresa que sostenga el proyecto. Frecuentemente, las Blockchain privadas no cuenta con criptomonedas ni se realizan acciones de minería.

▪ Híbrida o federada:

Estas Blockchain son un intento de aprovechar lo mejor de ambas opciones. Aquí la participación en la red es privada, es decir, el acceso de la red es controlado por una entidad, pero, como en las no permisionadas, el libro de contabilidad es público, de modo que cualquiera tiene acceso. Suelen ser utilizadas por gobiernos u organizaciones que deseen almacenar o compartir datos de una forma segura. En el ámbito sanitario, por ejemplo, se están comenzando a almacenar los datos de trazabilidad en la fabricación y distribución de medicamentos. Estos datos pueden ser revisados por las autoridades competentes con el fin de controlar la calidad y veracidad. El objetivo es ofrecer un alto nivel de transparencia y confianza.

Las características de las Blockchain híbridas son:

- La entidad reguladora determina qué elementos tienen permiso para acceder a la red.
- El acceso a la información de la cadena de bloques es público.
- En este caso no existen las criptomonedas para incentivar a los mineros, por lo que se utilizan otros protocolos de consenso para verificar la corrección de los datos.
- Parcialmente descentralizado, proporcionando un mejor nivel de seguridad y transparencia. [8]

	No Permitida o pública	Permitida o privada	Híbrida o federada
Nivel de acceso	Sin restricción	Una o varias organizaciones	Una o varias organizaciones
Participación	Sin permisos Anónima	Con permisos Usuarios conocidos	Con permisos Usuarios conocidos
Seguridad	Mecanismo de consenso	Pre-aprobación de los participantes Votaciones/Consensos múltiples	Aprobación por autoridades competentes
Rendimiento	Baja velocidad de transaccionalidad	Livianidad Mayor velocidad [9]	Velocidad media

2.3. ¿Cómo funciona Blockchain?

Redes P2P

Con carácter previo, para entender cómo funciona y opera la tecnología Blockchain debemos conocer sobre qué está construida. Hemos hablado de la descentralización de la red lo cual nos lleva directamente al concepto de redes P2P (**Peer to Peer**) o “redes de comunicación de igual a igual”. En una red centralizada, normalmente los ordenadores o clientes de la red están conectados a un servidor central, siguiendo así una arquitectura de cliente-servidor. En una red P2P, por el contrario, los ordenadores se conectan y comunican entre sí sin usar ningún tipo de servidor central, optimizando y administrando así la capacidad de la red, empleando la mejor ruta entre todos los nodos u ordenadores que la conforman. De esta manera, cada nodo de la red puede comportar como cliente y servidor a la vez.

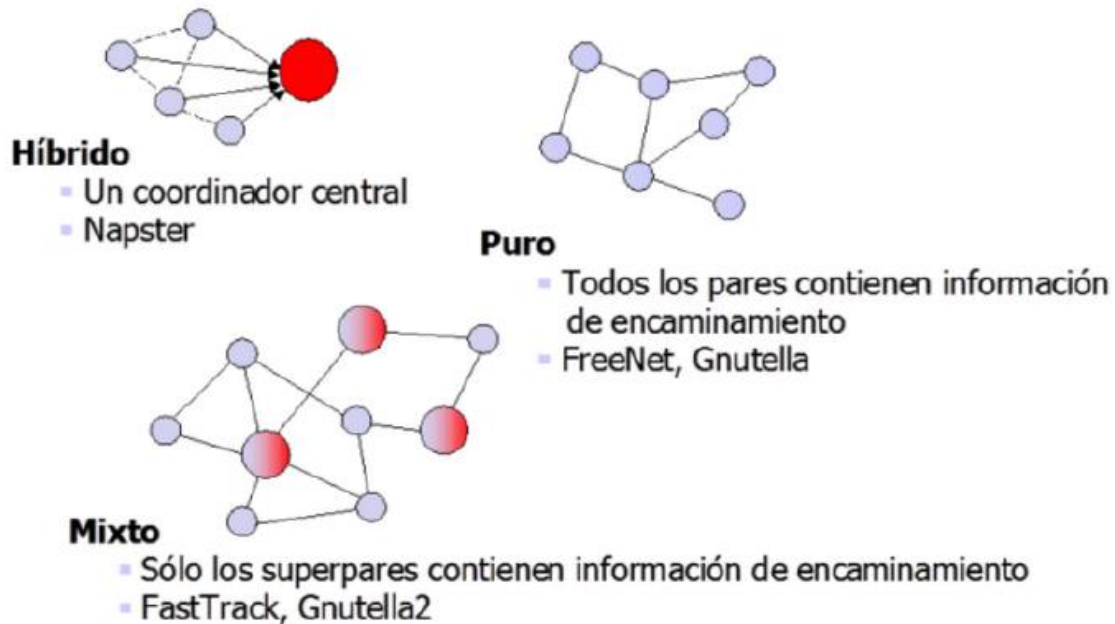
Así mismo, el elemento fundamental de toda red P2P es el “par” (o más exactamente “el igual”, si traducimos “peer” al español), el cual es la unidad de procesamiento básico de cualquier red de este tipo. Se tiende a pensar que un par es una aplicación ejecutándose en un ordenador conectado a una red como internet. Sin embargo, esta definición no engloba toda la potencialidad del concepto. Por ejemplo, esta definición no considera la posibilidad de que un par puede ser una aplicación distribuida a lo largo de varias máquinas trabajando en grupo, en lo que concretamente está basada parte de la tecnología Blockchain. Por esto, una definición más apropiada de par sería: una entidad capaz de desarrollar algún trabajo útil y de comunicar los resultados de ese trabajo a otra entidad de la red, ya sea directa o indirectamente.

Este tipo de redes han servido para hacer intercambios de archivos a través de internet, lo cual nos da la idea de disponer de un internet democrático dado que los intercambios se dan siempre entre pares. Cabe destacar el protocolo colaborativo BitTorrent para este tipo de tareas de intercambio de archivos, el cual permite el intercambio de datos a nivel mundial de forma anónima. También cabe destacar compañías como Skype y otras de telefonía, que también operan en base a este tipo de redes para la transmisión de voz y vídeo a través de internet. [14]

Para conocer más en profundidad estas redes, y en consecuencia Blockchain, veremos los tipos de redes Peer to Peer:

- **Red centralizada:** Una red P2P es considerada centralizada cuando existe un servidor central al que los ordenadores conectados hacen peticiones para localizar los nodos que poseen el contenido deseado. La desventaja de este tipo de redes sigue siendo la centralización y consecuencia de esta es que ese servidor central es un punto crítico muy importante para la consistencia de toda la red.
- **Red descentralizada y estructurada:** También conocidas como redes P2P híbridas, ya que no existe ningún directorio en un servidor central. En su lugar, existen diversos “nodos centrales” que facilitan el acceso a otros nodos y sus contenidos. De este modo, digamos que hay nodos de mayor importancia que otros para el correcto funcionamiento de la red.
- **Red descentralizada y no estructurada:** Este tipo es el más interesante desde el punto de visto de Blockchain, dado que no existen ordenadores o nodos que

actúan como controladores centrales de peticiones. Todos los nodos funcionan como clientes y como servidores según si están solicitando o compartiendo contenido.



Las aplicaciones que utilizan este tipo de redes poseen una serie de características muy interesantes, algunas de las cuales pueden ser apreciadas en las aplicaciones Blockchain en mayor o menor medida, estas son:

- **Descentralización:** Pueden gestionar conexiones variables y direcciones provisionales. Los ordenadores conectados se consideran iguales. Por el contrario, en el modelo cliente-servidor la información se concentra en servidores y los usuarios acceden a ella a través de programas de clientes, los cuales se comportan como meras interfaces de usuario. La principal desventaja de esta arquitectura es que lleva inevitablemente a ineficiencias, cuellos de botella y recursos desperdiciados.
- **Anonimato:** Es una característica muy deseable. Los usuarios pueden conectarse de forma anónima a la red. Aunque no todas las aplicaciones basadas en Blockchain cumplen esta característica por completo. Existen varias técnicas para alcanzar el anonimato en las redes P2P como la creación de multicasting para que el receptor de un contenido no pueda ser identificado, ocultación de la dirección IP y la identidad del emisor, comunicación mediante nodos intermedios a pesar de que sea factible contactar directamente con el destinatario, o ubicación involuntaria, encriptada y fragmentada de los contenidos.
- **Escalabilidad:** Gracias a no depender de un servidor central son fácilmente escalables, es decir, se puede crear un aumento en la capacidad de trabajo o de tamaño del sistema sin comprometer el correcto funcionamiento del mismo, y manteniendo la calidad necesaria de los servicios ofrecidos.
- **Rendimiento:** Los sistemas P2P mejoran el rendimiento agregando mayor ancho de banda, capacidad de almacenamiento y ciclos de computación de los

dispositivos diseminados por una red. Los sistemas descentralizados consiguen un mayor rendimiento, salvo para ciertas funciones como la búsqueda de recursos en la red (a veces tan necesaria). Por ello, las aplicaciones actuales P2P suelen tener una arquitectura mixta como hemos visto antes, incorporando el concepto de superpares, o pares en los que otros pares delegan las tareas de búsquedas de recursos en la red.

- **Seguridad:** La mayoría de requisitos de seguridad de este tipo de redes son comunes a los sistemas distribuidos tradicionales, como el establecimiento de relaciones de confianza entre nodos y objetos distribuidos y de esquemas de intercambio de claves de sesión. No obstante, en los sistemas P2P aparecen nuevos requisitos como la encriptación de las comunicaciones y almacenamiento de datos, gestión de derechos digitales, reputación e interoperabilidad con cortafuegos y NAT, etc. Esta característica es la razón principal de que las redes Blockchain se basen en la confianza entre los participantes que la conforman y de que hereden muchos de estos requisitos de seguridad anteriores.
- **Tolerancia a fallos:** En las arquitecturas cliente-servidor, si el servidor falla supone una pérdida total de la funcionalidad de la red (realmente es un punto muy crítico en este tipo de redes). Por el contrario, en las redes P2P no ocurre ya que el objetivo básico de diseño de estas redes es que esta no pierda su funcionalidad debido a fallos asociados a desconexiones de nodos o a nodos no alcanzables, caídas de red y fallos en los nodos. La forma en que se consigue esto es que varios nodos sean capaces de ofrecer los mismos recursos y servicios, replicándose espontáneamente en una serie de nodos de la red (de igual forma ocurre con Blockchain, como es el caso de la famosa hoja de contabilidad de la red Bitcoin).
- **Propiedad compartida:** Reduce el coste de la posesión de los sistemas y contenidos, así como el coste de su mantenimiento. El coste del sistema global se ve reducido porque en P2P se reducen las capacidades de cálculo, almacenamiento y ancho de banda ociosas.
- **Conectividad ad-hoc:** Las aplicaciones están preparadas para que en caso de que los nodos no estén disponibles todo el tiempo o que lo hagan de forma intermitente sigan funcionando correctamente. La naturaleza ad-hoc se reduce por ejemplo mediante proveedores de contenidos redundantes con técnicas de replicación espontáneas, o nodos de mayor peso que se encargan de mantener la información o mensajes destinados a pares temporalmente desconectados de la red.

Algunas de las aplicaciones P2P más conocidas siguen teniendo cierto grado de centralización, pudiendo ser completamente centralizadas, descentralizadas o un punto intermedio, pero es importante que el punto en común siempre es que todas siguen un sistema de computación de red distribuida donde todos los nodos o pares pueden actuar como clientes o servidores, comunicándose así de igual a igual. [10]

Qué son y cómo funcionan los nodos de la red

Cada equipo informático que participa en esta red se denomina nodo y comparten la responsabilidad de proporcionar servicios de red. Esto significa que cada nodo debe tener descargado el software de la Blockchain en cuestión para participar esta red entre pares. El conjunto de nodos coordina de forma descentralizada y distribuida las

acciones que cada usuario hace dentro de la red, lo que implica que esta red de dispositivos en todo el mundo se mantiene actualizada con las últimas transacciones realizadas y, como cada nodo tiene descargada la Blockchain, la red es redundante, segura y escalable.

Una vez el nodo tiene una copia actualizada, comienza a ser completamente operativo, permitiendo y verificando transacciones, que no son más que intercambios entre los nodos de la red que pueden ser bien variados, desde dinero hasta propiedades, valor, veracidad...Además, de respaldar una imagen completa o parcial de la Blockchain global. Normalmente las funciones de un nodo podrían resumirse en enrutamiento, base de datos de la Blockchain, minería y servicios de cartera o monedero, todo ello dependiendo de cada Blockchain en sí misma.

Los nodos son la mínima unidad de cómputo y los propietarios de los nodos contribuyen voluntariamente con sus recursos informáticos teniendo la oportunidad de cobrar las tarifas de transacción y ganar una recompensa por hacerlo. Un nodo puede ser un punto final de comunicación o un punto de redistribución de la comunicación, enlazando a otros nodos. Hemos dicho que cada nodo de la red se considera igual, aunque esto no quita que ciertos nodos mantengan diferentes roles en la forma en que soportan la red. Por ejemplo, no todos los nodos almacenarán una copia completa de una cadena de bloques ni validarán transacciones.

Cuando un usuario intenta agregar un nuevo bloque de transacciones a la cadena de bloques, este nodo transmite el bloque a todos los nodos que forman la red. Sobre la base de la legitimidad del bloque (validez de las transacciones), los demás nodos pueden aceptar o rechazar el bloque. Cuando un nodo acepta un nuevo bloque de transacciones, lo guarda y lo almacena sobre el resto de los bloques que ya ha almacenado, actualizando toda la cadena. El funcionamiento de un nodo a grandes rasgos se puede resumir en los siguientes puntos:

- 1) Comprueban si un bloque de transacciones es válido y lo aceptan o lo rechazan según esta validez.
- 2) Guardan y almacenan bloques de transacciones, manteniendo así una copia del historial de transacciones de la Blockchain a la que pertenezcan.
- 3) Transmiten este historial de transacciones al resto de nodos de la red que pueden necesitar sincronizarse con la cadena de bloques actual debido a que su historial esté desactualizado. [13]

Según la cantidad de información que poseen, podemos clasificar los nodos de una red Blockchain en diferentes tipos:

- **Nodos completos:** Son los que realmente admiten y proporcionan seguridad a la red y son indispensables para la misma. [11] Son los encargados de comprobar cada bloque dentro de la cadena y verificar que este cumpla con las normas y los parámetros establecidos para poder incorporarse. Este tipo de nodo tiene la potestad de rechazar una transacción que no cumpla con cualquiera de las normas previamente establecidas en la red. Cuando instalas un software de nodo completo (como Bitcoin Core, en el caso de la red Bitcoin), además de disponer del monedero más seguro, estarás descargando una copia completa de la Blockchain y pasará a ser un nodo más de la red. Así, emitirás tus transacciones, propagarás las emitidas por el resto de nodos y comprobarás que se cumplen con las reglas de consenso específicas de cada red.
- **Nodos livianos o parciales:** Estos no almacenan toda la información, como los nodos completos, por lo que no son tan seguros como estos y tampoco son

completamente independientes, lo cual los hace más vulnerables ante ataques, llegando a aprobar transacciones que realmente no cumplían con todos los protocolos de seguridad de la red. Es por esta razón que todas las transacciones deben pasar por nodos completos, ya que son estos los que disminuyen la vulnerabilidad de la red. Por tanto, podemos decir que estos nodos solo emiten transacciones y reciben la información de la Blockchain de manos de un tercero. Siguen lo que les dicta la mayoría de los nodos y son conocidos como “monederos ligeros”, más usados en dispositivos móviles o simplemente por personas que no tienen la necesidad de descargarse la Blockchain al completo. [12]

Una vez que tenemos el concepto más claro, podemos pasar a hablar sobre la seguridad que los nodos proporcionan a la red. Los nodos pueden estar online u offline dependiendo de si están recibiendo, guardando y transmitiendo los últimos bloques de transacciones desde y hacia otros nodos o no. Lo primero que debe hacer un nodo que ha estado offline durante un tiempo es ponerse al día con el resto de la cadena de bloques, actualizando así la que contenía antes de la desconexión del nodo, y efectuando la sincronización con la cadena de bloques actual de la red.

Teóricamente, una Blockchain completa puede ejecutarse en un solo nodo lo cual conlleva un serio problema de seguridad, dado que al ejecutarse en un solo dispositivo sería vulnerable a situaciones como cortes de energía, ataques de piratas informáticos o fallas del sistema. Esto nos conduce a la conclusión de que cuantos más nodos ejecuten la red Blockchain, esta será más segura y mejor será su capacidad de recuperación ante tales catástrofes. Cuando los datos de la red se distribuyen en tantos dispositivos, será extremadamente difícil para una entidad corrupta borrar o modificar todos los datos a la vez. Incluso si una gran cantidad de nodos se cae repentinamente y se vuelven inaccesibles debido a una gran crisis global, teóricamente un solo nodo podría mantener operativa toda la cadena de bloques de la red. E incluso si todos los nodos se desconectasen, teóricamente solo se necesita un nodo con el historial completo de Blockchain para volver a estar en línea y hacer que todos los datos sean accesibles de nuevo para todos los nodos.

A pesar de todo, los nodos pueden sufrir ataques, por ejemplo, un “pirata informático” puede romper la seguridad de un software en cuestión sin necesidad de alterar los datos contenidos en la Blockchain, redirigiendo la información o ganancias de dichos nodos a direcciones distintas a las programadas por sus dueños. Este es uno de los ataques más comunes a este tipo de software denominados “stealing address” (que profundizaremos más adelante), y es por ello que los desarrolladores de las diferentes Blockchain instan a sus usuarios a mantener actualizadas las versiones del software utilizado por la red. [13]

Transacciones

Como ya sabemos, las **transacciones** son una parte fundamental dentro de una cadena de bloques ya que todo lo que rodea a la cadena está diseñado para asegurar que las transacciones puedan ser creadas, propagadas por la red, validadas y añadidas al registro o cadena de bloques.

Las transacciones son agrupaciones de datos con firma digital que almacenan las transferencias de datos entre los remitentes (conocidos como entradas dentro de esas agrupaciones) y los destinatarios (salidas). Se transmiten a la red y, a medida que son validadas, se juntan y ordenan para formar los bloques de la Blockchain.

El **ciclo de vida de una transacción** comienza con la creación de la misma, que es luego firmada con una o más firmas indicando la autorización para gastar fondos o ejecutar cierto código en el contrato al que va dirigido. La transacción es entonces transmitida a la red, donde cada nodo participante en la red valida y propaga la transacción a los nodos a los que está conectado y así sucesivamente hasta que alcanza al resto de nodos en la red. Finalmente, la transacción es procesada por los nodos mineros o validadores e incluida en un bloque de transacciones registrado en la cadena de bloques. [19]

Cada Blockchain posee una estructura diferente para sus transacciones, de modo que no se puede definir en general una estructura de una transacción para Blockchain. Por ello existen diferentes **modelos de transacciones**, que explicaremos más adelante.

Firmas digitales: clave pública/privada y wallet/monederos

Un monedero o wallet es una cadena de números y letras del tipo: 18c177926650e5550973303c300e136f22673b74.

Esta es una dirección que aparecerá en varios bloques dentro de la Blockchain cuando se realicen transacciones. No hay registros visibles de quién realizó qué transacción con quién, únicamente el “número” de monedero o billetera. La dirección de cada monedero en particular también es llamada una clave pública. [3]

La criptografía de clave pública fue inventada en la década de 1970 y es la base matemática de la seguridad informática. Desde la invención de esta criptografía de clave pública, se han descubierto varias funciones matemáticas adecuadas, tales como exponenciación de números primos y multiplicación de curvas elípticas. Éstas funciones matemáticas son prácticamente irreversibles, lo cual significa que son fáciles de calcular en una dirección e inviables de calcular en la dirección contraria. Basada en estas funciones matemáticas, la criptografía permite la creación de secretos digitales y firmas digitales prácticamente infalsificables.

En la mayoría de Blockchain se utiliza la criptografía de clave pública. El par de claves consiste en una clave privada y una clave pública derivada de la privada. La clave pública se usa para recibir transacciones, y la clave privada para firmar transacciones y gastar los fondos asociados a la cuenta (en caso de haberlos). Lo verdaderamente potente de este concepto es que existe una relación matemática entre las claves pública y privada que permiten que la clave privada sea utilizada para generar firmas en mensajes. Estas firmas pueden ser validadas contra la clave pública sin necesidad de revelar la clave privada.

Cuando los fondos son gastados, el dueño actual de dichos fondos presenta su clave pública y firma (diferente cada vez, pero creada a partir de la misma clave privada) la transacción. A través de la presentación de la clave pública y firma, todos los participantes en la red pueden verificar y aceptar la transacción como válida, confirmando que la persona que transfiere los fondos los posee al momento de la transferencia.

Entonces, una **clave privada** es simplemente un número escogido al azar. La propiedad y control de una clave privada es la raíz del control del usuario sobre los fondos asociados con la dirección correspondiente. Se usa para crear las firmas requeridas para firmar transacciones demostrando la pertenencia de los fondos usados en una transacción. Por lo tanto, la clave privada debe permanecer en secreto en todo momento, ya que revelarla a terceros equivale a darles el control de la cuenta asegurada por dicha clave. También es conveniente hacer copias de respaldo de las

claves privadas para protegerlas de pérdidas accidentales, ya que si se pierde no puede ser recuperada y los fondos asegurados por dicha clave se perderán para siempre.

La **clave pública** es generada a partir de la clave privada usando multiplicación de curva elíptica, la cual es irreversible: $K = k * G$ donde k es la clave privada, G es un punto constante llamado el punto generador y K es la clave pública resultante. La operación inversa, conocida como “búsqueda del logaritmo discreto”, es decir, calcular k a partir de K , es tan difícil como probar todos los valores de k , es decir, una búsqueda por fuerza bruta. [33]



Una **dirección** es una cadena de dígitos y caracteres que puede ser compartida con cualquiera que desee enviarte transacciones (por ejemplo, en Bitcoin, las direcciones producidas a partir de una clave pública consisten en una cadena de números y letras, comenzando siempre por el dígito “1”). Esta dirección se obtiene a partir de la clave pública a través del uso de hashing criptográfico de sentido único, que comentaremos posteriormente. [32]

En definitiva, para poder gastar/mandar transacciones desde una dirección necesitamos demostrar que conocemos la clave privada, de la clave pública a la que se refiere dicha dirección. Y para poder demostrar que conocemos la clave privada sin revelarla públicamente es para lo que se utiliza la **firma digital**. Las firmas son elementos criptográficos que se calculan a partir de la clave privada y de una combinación de otra información incluida en la transacción. Es aquí donde entra en juego la magia de la criptografía, ya que gracias a ello es posible que una clave pública pueda usarse para verificar que dicha firma se ha creado usando la clave privada correspondiente. Además, permiten demostrar además del conocimiento de la clave privada, que el poseedor de la misma confirma los datos de dicha transacción. Por lo tanto, cada firma es únicamente válida para una transacción específica.

Por lo tanto, el dueño de la clave privada puede firmar una transacción y gastar fondos sin preocuparse de que nadie vaya a conocer su clave privada porque la clave privada en sí nunca queda expuesta públicamente y es prácticamente imposible de averiguar. La firma y la clave pública de la dirección de la que se envía la transacción se añaden al campo de inputs de la transacción, esto demuestra que el propietario de la clave privada tiene realmente la intención de efectuar esa transacción y se asegura de que no puede ser alterada.

Hash: funciones criptográficas

Hash es el nombre utilizado para identificar un tipo de función criptográfica utilizado en Blockchain (entre otros). Una función hash es un algoritmo que posee una serie de propiedades muy útiles para el cifrado de datos. ¿En qué consiste una función de este tipo?

Entender estas funciones es relativamente sencillo, para ello, primero veamos sus propiedades y una serie de ejemplos:

- **Eficiente computacionalmente.** Este es un requisito esencial ya que un ordenador debe ser capaz de resolver una función hash en un corto periodo de tiempo.
- **Determinismo.** Las funciones hash deben ser determinista, es decir, dada una entrada, siempre debe obtenerse el mismo resultado. Obviamente, si tenemos una entrada, pero nos da varias salidas diferentes, validar si esa entrada ha sido modificada o no, sería imposible.
- **Pre-imagen resistente.** Esto quiere decir que una salida de una función hash no debe revelar nada acerca del contenido de la entrada. Dicha entrada puede ser cualquier cosa: números, letras, palabras completas y un largo etc. A pesar de ello, la salida será una combinación alfanumérica de una longitud fija, evitando dar pistas de la longitud de la entrada.
- **Resistente a colisiones.** Esto quiere decir que debe ser imposible obtener la misma salida de dos entradas diferentes. De todos modos, la salida es de una longitud fijada, de modo que hay un límite de salidas (a pesar de que es una cifra muy elevada) que pueden ser producidas por una función hash, de modo que, estadísticamente hablando, más de una entrada, producirá la misma salida. [20]
- **Imposible aplicar ingeniería inversa.** Debe ser imposible utilizar esta técnica para revertir el proceso matemático que crea la salida. No existe la una función inversa para las funciones hash. [24]

Las funciones hash son llamadas **funciones unidireccionales** dado que, como hemos visto en las propiedades, no puede ser reversible. La forma más simple de presentar una función unidireccional sería por ejemplo a través de las funciones modulares (de la forma $X \bmod Z = Y$). Tomemos de ejemplo la ecuación $X \bmod 5 = Y$, obtendremos:

Entrada (X)	0	1	2	3	4	5	6	7	8	9	10
Salida (Y)	0	1	2	3	4	0	1	2	3	4	0

Como podemos ver, el patrón es muy sencillo de averiguar ya que solo hay 5 posibles opciones, la cuales rotan hasta el infinito. Todo esto es importante porque tanto la función como la salida pueden hacerse públicas, pero nunca será posible obtener la entrada asociada siempre y cuando, en la función utilizada de ejemplo, se mantenga en secreto el número elegido como 'X'.

Por ejemplo, si $X = 17$, el resultado es 2. Entonces, ahora publicamos nuestra función $X \bmod 5 = Y$, y que el resultado que nos da es 2. Nadie podría saber cuál fue la entrada ya que hay una cantidad infinita de posibles X que nos den 2 como resultado: 7, 52, 3492. . . Podemos ver que es un proceso irreversible que, aplicado a funciones más sofisticadas y mayores valores, se convierte en una tarea imposible el averiguar la entrada.

Dentro de las funciones hash podemos encontrar muchas clases, siendo las más comunes:

- **SHA**: Algoritmo de hash seguro (del inglés: Secure Hashing Algorithm) (SHA-2 y SHA 3).
- **RIPEMD**: Resumen de mensajes de evaluación de primitivas de integridad RACE (del inglés: RACE Integrity Primitives Evaluation Message Digest).
- **MD5**: Algoritmo de resumen de mensaje 5 (del inglés: Message Digest Algorithm 5).
- **BLAKE2**

Todas ellas son similares entre sí, pero con ciertas diferencias en cuanto a cómo el algoritmo genera un resumen o salida dada una entrada, además de diferir en los tamaños de las salidas.

Una vez entendido el concepto de función Hash veremos, mediante un ejemplo con intercambio de monedas, cómo se aplica en Blockchain.

Partimos de un grupo de personas que deciden hacer una moneda ajena a cualquier otra para ellos. Para tener un seguimiento, uno de ellos, llamado Bob, decide llevar una contabilidad en un diario:

1. *Ana dio 10 monedas a María.*
2. *María dio 5 monedas a José.*
3. ~~*José*~~ *María* dio 3 monedas a ~~Ana~~.
4. ...

Uno de ellos, José, quería robar monedas, para ello modificó el diario:

1. *Ana dio 3 monedas a María.*
2. *María dio 5 monedas a José.*
3. *José dio 3 monedas a Ana.*
4. ...

Evidentemente, Bob, el contable, nota que alguien había modificado el diario. Para evitar ello, decide utilizar una cosa muy útil llamada función hash, que convierte el texto en un aparente sinsentido de letras y números. Entonces Bob añade a cada nuevo registro en el diario, un hash:

6. *Ana dio 10 monedas a María*
cff4e860db57c2bfb7c010927c3f6fee
7. *Mary dio 5 monedas a José*
803c28370e9a16e628a23d46d3ebe711

José decide cambiar de nuevo algunos registros. De noche, coge el diario, cambia el registro y genera un nuevo hash:

6.	<i>Ana dio 10 monedas a María</i> <i>cff4e860db57c2bfb7c010927c3f6fee</i>
7.	<i>Mary dio 5 monedas a José</i> <i>803c28370e9a16e628a23d46d3ebe711</i> <i>4ae41f8cc3d4cc905ff664c75ceab9da0</i>

De nuevo, Bob, notó cambios y, por lo tanto, decidió añadir un extra de seguridad, cada nuevo hash, será generado a partir del nuevo registro y el hash del registro anterior:

<i>Entrada</i>	<i>Hash</i>
<i>Ana dio 10 monedas a María</i>	<i>8977e7c112aea5b0a62e9c5f30840203</i>
<i>María dio 5 monedas a José</i> <i>8977e7c112aea5b0a62e9c5f30840203</i>	<i>e37a8dlcc39ed9f54afadb6c6cafet39</i>
<i>María dio 3 monedas a Ana</i> <i>e37a8dlcc39ed9f54afadb6c6cafet39</i>	<i>5b9f0e325f58766f5a2dfe7eec623f6d</i>
<i>Ana dio 1 moneda a Andrés</i> <i>5b9f0e325f58766f5a2dfe7eec623f6d</i>	<i>55f28e65412b22aa3d6002bcf7d67201</i>

Ahora, para que José haga trampas, debería cambiar el Hash en todas las entradas siguientes, ya que sus hashes dependen del mismo. Pero como José quiere más dinero, se pasó toda la noche haciendo las modificaciones pertinentes.

Aquí entra en juego un nuevo concepto: **Nonce**. Este es el nombre que se le da a un número que se añade al registro el cual provoca que el hash generado tenga una terminación deseada, en este ejemplo, el nonce será un número aleatorio el cual provocará que el hash termine en dos ceros.

<i>Entrada</i>	<i>Hash</i>
<i>Ana dio 10 monedas a María 451</i>	<i>219711e62645a21f2742ada2c6f2a900</i>
<i>María dio 5 monedas a José 13</i> <i>219711e62645a21f2742ada2c6f2a900</i>	<i>1cc4c07fa0757848b439e2396ce87d00</i>
<i>María dio 3 monedas a Ana</i> <i>1cc4c07fa0757848b439e2396ce87d00</i>	<i>e43aal32f4b67c65ba6914824a39b3900</i>
<i>Ana dio 1 moneda a Andrés</i> <i>e43aal32f4b67c65ba6914824a39b3900</i>	<i>99012fe16897c19465941d3530afa900</i>

Ahora, la falsificación de registros es muy difícil, y no solo para una persona, sino también para una máquina, ya que no pueden descifrar el nonce rápidamente.

En conclusión, una función hash es una función de sentido único que produce una huella o “hash” a partir de una entrada de tamaño arbitrario y se usan extensivamente en Blockchain para:

- Obtención de una dirección a partir de una clave pública como se ha descrito anteriormente.
- Durante el minado, en la prueba de trabajo para ganar el reto (en caso de usar este algoritmo de consenso que describiremos posteriormente).
- Para generar árboles Merkle que se incluyen en los bloques minados y que se explicarán también posteriormente.

Métodos de consenso

Los métodos de consenso son el mecanismo a través del cual una red Blockchain alcanza un acuerdo. Este consenso se lleva a cabo entre los distintos participantes (nodos) de la red Blockchain sobre las transacciones que van a escribirse en la cadena, debido a que no dependen de una autoridad central, es decir, el consenso es el responsable de que todos los nodos de la Blockchain tengan los datos verificados, evitando a su vez su manipulación. Es aquí donde los algoritmos de consenso entran en juego, encargándose de asegurar que las reglas del protocolo son respetadas y garantizando que todas las transacciones tienen lugar de una forma fiable. [16]

Es importante entender la diferencia entre un algoritmo y un protocolo, ya que a menudo ambos términos son utilizados de forma indiferente, sin embargo, no se tratan de lo mismo:

- **Protocolo:** reglas primarias de una Blockchain.
- **Algoritmo:** mecanismo a través del cual el protocolo será seguido.

La red siempre funcionará sobre un protocolo que definirá la forma de funcionamiento del sistema, haciendo que tanto los elementos como los participantes de la red tengan unas reglas fijas que seguir. Teniendo este protocolo, el algoritmo se encarga de decirle al sistema qué y cómo se puede hacer algo de modo que se produzcan los resultados deseados. En una Blockchain, estos algoritmos determinan la validez de las transacciones y los bloques. Proof of Work y Proof of Stake son los dos ejemplos más característicos de algoritmos de consenso, mientras que Bitcoin y Ethereum serían protocolos.

A parte de los descritos posteriormente, existen una gran cantidad de métodos diferentes como son: Tolerancia a Fallas Bizantinas Delegada (dBFT), Prueba de Quemadura (PoB), Prueba de Tiempo Transcurrido (PoET), Prueba de Asignación (PoA), Prueba de Punto de Control (PoC), Prueba de Formulación (PoF), etc.

- **Proof of Work (PoW):**

Fue el primer algoritmo de consenso que apareció, remontándose 20 años atrás con la aparición de Hashcash, que fue una propuesta realizada por Adam Back en 1997 para combatir el correo basura o spam, el cual inspiró el mecanismo de prueba de trabajo usado en Bitcoin. Consiste en un mecanismo rápido de verificación en el que el remitente del mensaje tiene interés suficiente en dicho envío como para “pagar” con tiempo de CPU para poner una marca que demuestre que se manda por algún motivo.

Este pago con tiempo de CPU consiste en la resolución de complicados problemas computacionales que posteriormente, serán validados por la red. La principal característica de esta estrategia es su asimetría. La prueba de trabajo del cliente es ciertamente complicada, mientras que la verificación por parte de la red es sencilla. Es decir, se tarda en producir y es computacionalmente costoso, pero la verificación es sencilla, ya que la prueba diseña patrones que facilitan la verificación.

El funcionamiento de este algoritmo se puede separar en distintas etapas:

- **Etapas 1:** En el momento en el que el nodo se conecta a la red, ésta le asigna una tarea computacionalmente costosa que debe ser resuelta con el fin de obtener incentivos.
- **Etapas 2:** Se procede a la resolución de la tarea, lo cual necesita de mucha potencia de computación. A este proceso se le conoce como minería.
- **Etapas 3:** Resuelta la tarea, la solución se comparte con la red para ser verificada, comprobando que la tarea cumple los requisitos exigidos. Tiene acceso a los recursos de la red si los cumple o, por el contrario, es rechazado tanto el acceso a la red como la solución a la tarea. Aquí entra el juego el concepto de **protección contra el doble gasto**. Esta protección evita que, una vez presentada y verificada una tarea, esta sea utilizada más de una vez.
- **Etapas 4:** Con la confirmación de que la tarea se ha cumplido, el cliente consigue acceso a la red y sus recursos, recibiendo una ganancia por el trabajo computacional realizado. [17]

- **Proof of Stake (PoS):**

Es un protocolo de consenso creado para reemplazar al conocido Proof of Work aportando una mejor seguridad y escalabilidad a las redes que lo implementen.

En este caso no existen mineros como tal, dado que los nodos dedicados a verificar las transacciones en PoS se denominan validadores. La selección del nodo que debe validar un bloque de transacciones se hace de forma parcialmente aleatoria dado que también se siguen una serie de criterios previamente definidos en la red, como pueden ser: cantidad de activos poseídos o el tiempo de antigüedad en la red, pudiendo ser utilizados cualesquiera otros. Una vez definidos, comienza el proceso de selección entre los nodos participantes de manera aleatoria y al ser seleccionados, estos nodos tendrán la capacidad de verificar transacciones o añadir nuevos bloques a la cadena.

Como vemos este protocolo de Prueba de Participación (o Estaca) es un proceso completamente distinto al conocido protocolo de Prueba de Trabajo (PoW). Donde cada uno de sus nodos realizan un arduo trabajo de cómputo para resolver acertijos criptográficos. Lo que significa que PoW, a diferencia de PoS, necesita de grandes cantidades de energía y equipo especializado para realizar sus operaciones. En PoS, por el contrario, esto no es necesario. En PoS el proceso es mucho más sencillo y energéticamente amigable. Son estas razones por la que muchos proyectos Blockchain en la actualidad se interesan por este nuevo protocolo.

El funcionamiento de este protocolo de Prueba de Participación es bastante particular. Este sistema busca incentivar a los participantes para que posean en todo momento, una determinada cantidad de monedas. La posesión de monedas, les permite ser elegidos por el proceso de selección aleatoria que se realiza para designar tareas. Bajo este esquema, aquellos que tengan más reservas, tienen mayor peso en la red y mayores oportunidades de ser elegidos. Una vez elegidos pueden validar

transacciones y crear nuevos bloques dentro de la red. Permitiéndoles recibir ganancias e incentivos por el trabajo realizado. [18]

Como hemos explicado previamente, existen distintas maneras de seleccionar los nodos validadores que crearán los bloques con el fin de evitar una posible centralización debida a la cantidad de activos bloqueados o “apostados”, ya que la blockchain, al utilizar un proceso de selección pseudo-aleatorio, considerará otros factores de aleatorización junto a la cantidad y riqueza de las monedas.

Existen muchas derivaciones de este algoritmo, entre las más destacadas tenemos:

- Prueba de Participación Anónima (PoSA)
- Prueba de Importancia (PoI)
- Prueba de Almacenamiento
- Prueba de Tiempo de Juego (PoST)
- Prueba de Velocidad de Estaca (PoSV)

- **Obelisk**

Este algoritmo presenta como principal objetivo la supresión de fallos de los algoritmos previamente descritos. Esto hace posible que una Blockchain se mantenga distribuidamente sin necesidad de participación y tanto poder de cómputo. Se minimiza la intervención de minería, mejorando la rapidez y seguridad de las transacciones. Distribuye la influencia en la red conforme a un concepto denominado “**red de confianza**”, donde la densidad de la red de suscriptores de un nodo determina su mayor intervención en la cadena.

- **Prueba de Participación Delegada (DPoS):**

La Prueba de participación delegada (DPOS) es un mecanismo de consenso muy rápido al que a menudo se lo conoce como democracia digital, gracias a su sistema de votación ponderado por estaca.

La forma en que funciona es que los usuarios votan por "delegados" a quienes se les da el poder de obtener ganancias al ejecutar un nodo completo. El peso de su voto depende de su participación o bloqueo de monedas. Dado que los delegados desean recibir la mayor cantidad de votos posible, se les incentiva constantemente a crear cosas valiosas para la comunidad, ya que es probable que reciban votos adicionales por hacerlo.

Se supone que este método es más eficiente y protege a los usuarios de interferencias regulatorias no deseadas.

Este algoritmo se ha hecho popular entre los desarrolladores de aplicaciones descentralizadas (dApps). [27]

- **Prueba de Actividad (PoA):**

El concepto fue introducido por primera vez en el año 2012 como una alternativa a la Prueba de Participación, PoS. La Prueba de actividad es esencialmente una estructura alternativa para Bitcoin y es una mezcla de dos de los mecanismos de consenso más populares: Prueba de Trabajo y Prueba de Estaca. Se introdujo la Prueba de actividad para frenar los temores sobre el fin de la minería en Bitcoin al agotarse los 21 millones de monedas disponibles y que complementa la Prueba de Trabajo para ayudar a prevenir un ataque del 51%.

Este mecanismo funciona comenzando de una manera de Prueba de Trabajo donde los mineros esencialmente resuelven un rompecabezas criptográfico y reclaman su recompensa si tienen éxito. La diferencia radica en el hecho de que los bloques minados son solo encabezados y direcciones de recompensa de minería en lugar de contener transacciones.

Prueba de actividad, en resumen, selecciona un par aleatorio de la red para firmar un nuevo bloque. Este método requiere un intercambio continuo de datos. Para reducir el tráfico, la "plantilla" de bloque no incluye la lista de transacciones y, en su lugar, la agrega el último firmante.

Como punto en contra, hereda la desventaja tanto de la Prueba de Trabajo como de la Prueba de Participación en términos de altos recursos utilizados y validadores maliciosos. [27]

- **Prueba de Capacidad (PoC):**

La Prueba de capacidad es un mecanismo de consenso que utiliza un proceso llamado trazado. Con Prueba de Trabajo, los mineros usan la computación para adivinar la solución correcta; sin embargo, con Proof Of Capacity, las soluciones se almacenan previamente en almacenes digitales (como discos duros). Este proceso se llama trazado. Después de que se ha trazado un almacenamiento (lo que significa que se ha llenado de soluciones), puede participar en el proceso de creación de bloques.

Quien tenga la solución más rápida para el rompecabezas de un (nuevo) bloque, puede crear el nuevo bloque. Cuanta más capacidad de almacenamiento tenga, más soluciones podrá almacenar, mayores serán sus probabilidades de crear un bloque.

La prueba de capacidad contiene dos partes distintas en el trazado o la creación del archivo de parcelas y la extracción real de los bloques. El tamaño de su disco duro es el factor que define el tiempo que le llevará desarrollar los archivos de trazado. Esto varía de días a semanas pares. [27]

Minería

Dentro de Blockchain, la **minería** es la parte encargada de permitir que la red funcione de una forma descentralizada de igual a igual, sin necesidad de una autoridad controladora. Este proceso consiste en verificar las transacciones entre usuarios y agregarlas a la cadena que conforma la red.

Su funcionamiento consiste en que un nodo minero en la red recopila transacciones y trabaja para organizarlas en bloques. Cada vez que se realizan transacciones, los nodos mineros reciben y verifican las transacciones, las agregan a la agrupación de memoria y comienzan a ensamblarlas en un bloque de transacciones múltiples.

Antes de comenzar el proceso, al nuevo bloque se añade la primera transacción conocida como la transacción **coinbase**, que es una transacción cuyo valor base es el equivalente al de la recompensa activa en ese momento por la minería de dicho bloque. Es por esto que son llamadas **transacciones generadoras**, pues la recompensa por añadir dicha transacción al nuevo bloque, es una recompensa "virgen" que en el caso de criptomonedas, por ejemplo, se trata de nuevas monedas que no están en circulación. Así, una transacción coinbase, contiene única y exclusivamente nuevas monedas que nunca han estado en la blockchain. [51]

Una vez que se ha hashado cada transacción, estos hashes se organizan en algo llamado **árbol Merkle** o árbol hash, lo que significa que los hashes se organizan en pares y se concatenan hasta que se alcanza "la parte superior del árbol", también llamado hash raíz o una raíz de Merkle.

El hash raíz junto con el hash del bloque anterior y un número aleatorio llamado nonce se colocan en el encabezado del bloque. El encabezado de los bloques se procesa como un hash y produce una salida que servirá como identificador de los bloques.

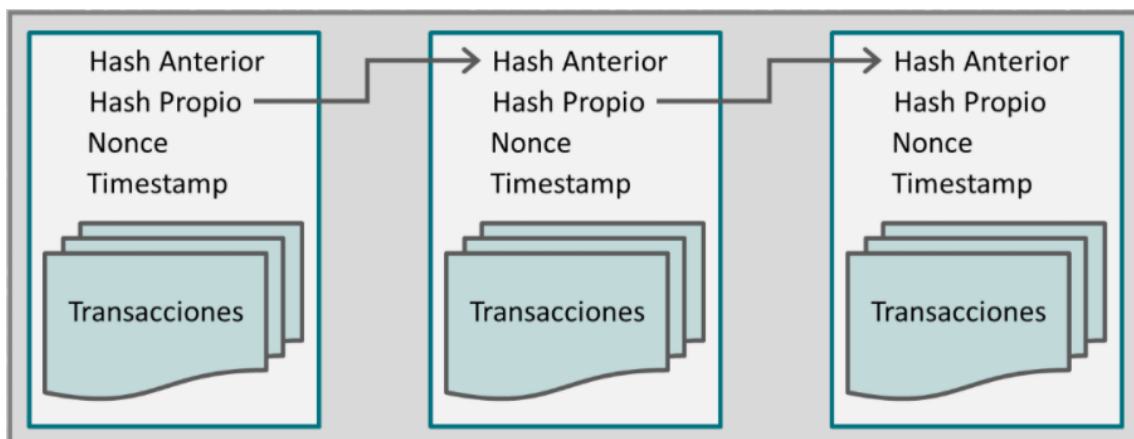
El identificador de bloques debe ser menor que un determinado valor objetivo establecido por el protocolo. En otras palabras, el hash de encabezado de bloque debe comenzar con un cierto número de ceros.

Este valor objetivo, también conocido como la dificultad de hash, se escala, asegurando que la velocidad a la que se crean los nuevos bloques se mantenga proporcional a la cantidad de poder de hashing en la red.

Los mineros mantienen el hash del encabezado una y otra vez mediante la iteración a través del **nonce** hasta que un minero en la red finalmente produce un hash válido. Cuando se encuentra un hash válido, el nodo fundador transmitirá el bloque a la red. Todos los demás nodos comprobarán si el hash es válido, agregarán el bloque a su copia de la cadena de bloques y continuarán con la minería del siguiente bloque.

Sin embargo, a veces sucede que dos mineros emiten un bloque válido al mismo tiempo y la red termina con dos bloques en competencia. Los mineros comienzan a explotar el siguiente bloque basándose en el bloque que recibieron primero. La competencia entre estos bloques continuará hasta que el siguiente bloque se mine según uno de los bloques competidores. El bloque que se abandona se llama bloque huérfano o bloque obsoleto. Los mineros de este bloque volverán a minar la cadena del bloque ganador. [26]

Cabe destacar los grupos de minería, ya que mientras que la recompensa en bloque se otorga al minero que descubre el hash válido primero, la probabilidad de encontrar el hash es igual a la porción del poder minero total de la red. Los mineros con un pequeño porcentaje del poder minero tienen una probabilidad muy pequeña de descubrir el siguiente bloque por su cuenta. Los grupos de minería se crean para resolver este problema, lo que significa que los mineros, que comparten su poder de procesamiento a través de una red, comparten la recompensa por igual entre todos los miembros del grupo, de acuerdo con la cantidad de trabajo que contribuyen a la probabilidad de encontrar un bloque.



2.4. Sumario

Veamos un resumen de las ideas principales de lo visto anteriormente dado que son muchos los conceptos nuevos que nos pueden llevar a confusión acerca de los fundamentos de Blockchain:

1. Si un usuario de la Blockchain posee activos digitales, entonces, necesariamente tendrá un “monedero”.
2. Un monedero es una dirección en la Blockchain, es decir, una llave/clave pública de la misma.
3. Un usuario que quiera realizar una transacción enviará un mensaje con la transacción firmada con su llave privada.
4. Previamente a su aprobación, se revisará por cada nodo que “votará” según el algoritmo de consenso (o protocolo) implementado por esta cadena de bloques.
5. La transacción es colocada en la cadena de bloques por los nodos mineros que validan con sus “votos” la transacción.
6. Los equipos de la red que sostienen la Blockchain se denominan nodos (como hemos visto, por lo general, no todos los nodos son iguales).
7. Los mineros colocan las transacciones en bloques en respuesta a las condiciones y desafíos del método de consenso de dicha Blockchain.
8. Por lo general, después de que los mineros cierran con éxito un bloque de transacciones, reciben una recompensa por el trabajo realizado.
9. Un bloque contiene una marca de tiempo, una referencia al bloque anterior, las transacciones y el problema de cómputo que tuvo que ser resuelto antes de que el bloque se incluyera a la cadena de bloques o Blockchain.
10. La criptografía es esencial en las Blockchain para frustrar a los ladrones que quiesieran “hackear” la red. Éstas claves suelen estar hechas por generadores o keygens. que utilizan matemáticas muy avanzadas que involucran números primos para crear claves.
11. La red distribuida de nodos que necesitan llegar a un consenso definido por el protocolo de la red hace que el fraude sea prácticamente imposible dentro de la red Blockchain. [3]

2.5. Debilidades

Fallas Bizantinas

Este es uno de los conceptos más importantes de la Blockchain. Sin la tolerancia a fallas bizantinas no sería posible esta tecnología tal y como la conocemos.

El concepto de falla bizantina, se deriva del Problema de los Generales Bizantinos. Es un problema lógico que supone, en resumidas cuentas, que los participantes deben acordar una serie de estrategias concretas para evitar fallas catastróficas del sistema. El problema viene cuando existe la posibilidad de que los actores que conforman el sistema puedan ser o no confiables. Ante este hecho, el sistema debe crear los mecanismos necesarios que garanticen que esos actores maliciosos no puedan conducir a la falla sin más remedio. El desarrollo de estos mecanismos de seguridad son los que proporcionan la tolerancia a dichas fallas.

Este es uno de los objetivos más complejos no solo de la tecnología blockchain sino de cualquier sistema informático distribuido. Cabe destacar que el primer diseño que consiguió solventarlo fue desarrollado por Satoshi Nakamoto con la cadena de bloques de Bitcoin, lo que marcó un antes y un después que acompaña a esta tecnología hasta nuestros días.

En concreto, la **Tolerancia a Fallas Bizantinas** es la capacidad de un sistema informático distribuido de soportar las llamadas fallas bizantinas, que pueden ser:

- Fallas de consenso.
- Fallas de validación.
- Fallas de verificación.
- Fallas de verificación de información.
- Fallas en el protocolo de actuación ante situaciones comprometidas de la red.

Esta tolerancia está directamente vinculada a la capacidad de que la red, en conjunto, pueda crear un mecanismo general de consenso con el fin de dar una respuesta coherente ante un fallo del sistema.

Esto se consigue mediante la definición de una serie de reglas que resuelva el complejísimo Problema de los Generales Bizantino. En las Blockchain (públicas por lo general) ésta tolerancia se considera un requisito y para poder alcanzarlo, un sistema o algoritmo tolerante a fallos bizantinos debe cumplir al menos con lo siguiente:

1. Se debe iniciar cada proceso con un estado no decidido (ni SI ni NO): En este punto la red propone una serie de valores determinísticos aplicables al proceso.
2. Para compartir valores, se debe garantizar un medio de comunicación: Con el fin de desplegar mensajes de forma segura, por lo que el medio también servirá para comunicar e identificar de forma inequívoca las partes involucradas.
3. Los nodos computan los valores y pasan a un estado decidido (SI o NO): Cada nodo debe generar su propio estado, el cual es parte de un proceso puramente determinista.
4. Una vez decididos, totalizan y gana el estado con mayor cantidad de decisiones a favor.

Pongamos un ejemplo simplificante de lo anterior, aplicado a la Blockchain de Bitcoin por ejemplo:

Supongamos que Pablo realiza una transacción en Bitcoin. Cada nodo en la red, comienza a compilar la transacción en un estado no decidido (transacción no confirmada). Como hemos visto la confirmación de esa transacción pasa por un trabajo de minería (aplicando el protocolo de consenso de la Blockchain). El proceso de minería, verifica que el hash de la transacción sea correcto y lo incluye en un bloque. Este proceso de verificación es intensivo en cálculos y solo es posible por medios determinísticos. Con cada nueva confirmación (estado decidido) de la transacción dada por la mayoría de la red, Pablo puede estar seguro de que la transacción ha sido tomada como válida.

La tolerancia a fallas bizantinas tiene la capacidad de resolver diversos problemas en diferentes casos de uso como pueden ser: compiladores de software, sistemas de almacenamiento de datos, sistemas de aviónica, protocolos de consenso en redes Blockchain, etc.

Cabe destacar este último, aunque consideremos todos de igual importancia, debido a que los protocolos de consenso en Blockchain como PoW en Bitcoin permiten alcanzar a una red distribuida un consenso en condiciones bizantinas. Cuando Satoshi Nakamoto diseñó Bitcoin, tomó en cuenta este tipo de tolerancia, creando una serie de reglas y aplicó el PoW para crear un software con tolerancia a estas fallas. Sin embargo, esta tolerancia no es del 100 %. Pese a ello, PoW ha demostrado ser una de las implementaciones más seguras y confiables para redes Blockchain. En este sentido, el algoritmo de consenso de prueba de trabajo, es considerado por muchos como una de las mejores soluciones para las fallas bizantinas. PoS y DPoS por su parte no son completamente tolerantes a fallos bizantinos, razón por la cual suelen complementarse con otras medidas de seguridad.

Para concluir esta debilidad (o en algunos casos virtud) veremos sus ventajas y desventajas:

En las ventajas cabe destacar: la capacidad para garantizar correctitud de datos e información en sistemas distribuidos, solución al problema de procesamiento de información en ambientes heterogéneos, alta eficiencia en términos computacionales y energéticos, ofrece implementaciones que impactan positivamente en la escalabilidad si son bien construidas y mientras más nodos aplicando la tolerancia a fallos bizantinos mejor y más seguro será el modelo.

Entre las desventajas que podemos encontrar tenemos que la creación de estas soluciones es bastante compleja lo que puede incurrir en otros problemas de seguridad con su implementación y que garantizar su correcto funcionamiento requiere que la distribución del sistema sea creciente, es decir, mientras más nodos aplicando el proceso, más seguro será, lo cual también tiene un impacto negativo en la escalabilidad y el ancho de banda de la red. [34]

Ataque del 51%, dispositivos ASIC y ataque del doble gasto

Un **ataque del 51%** sobre una red Blockchain ocurre cuando una entidad toma el control de más del 51% de la red, convirtiéndose en su centro y tomando el control sobre ella rompiendo las características de descentralización y seguridad que ésta proporciona. Si una sola entidad controla el 51% de la capacidad de decisión de la Blockchain, tiene autoridad sobre ella durante el tiempo que la controle. [62]

Las dos principales características detrás de un ataque de este tipo son la existencia de fallos de seguridad concretos que simplifican el proceso para el atacante y la manera de validar los bloques, es decir, el tipo de algoritmo de consenso que usa la red.

Un ejemplo de lo que puede hacer alguien con el 51% de la red es el **ataque del doble gasto**: se envía un pago con una transacción y justo antes empiezan a minar bloques en secreto. [41] La transacción es validada por toda la red y llega a su destino, con el pago aceptado como tal. Una vez lo que se haya comprado esté asegurado, el atacante hace pública la cadena de bloques que minó en secreto (también llamado **shadow mining** o minería en la sombra). Al tener el 51% de la red, esa cadena de bloques será validada por el resto de la red, en una especie de proceso democrático explicado con anterioridad. Al ser aceptada, la transacción que se hizo justo después desaparecerá, y el dinero del pago volverá al atacante, que podrá gastarlo otra vez. Además, reescribirán todos los bloques posteriores, en lo que se conoce como reorganizaciones de cadena profunda o **deep chain reorganizations**, lo que suele ser un auténtico caos para los usuarios de la red.

Los dos ataques más importantes de este tipo fueron contra la Blockchain de Verge que fue atacado en al menos dos ocasiones y el que más destaca por su popularidad, el ataque sufrido por Bitcoin Gold en mayo de 2018, cuando en uno de los forks de Bitcoin, el atacante hacía gastos por valor de 18 millones de dólares que luego eliminaba de la red. En total el atacante vendió Bitcoin Gold y volvió a recuperarlos 76 veces antes de que se detuvieran las transacciones con este activo. [35]

En este [enlace](#) se mantienen actualizados los costes en dólares de controlar el 51% de la red en una serie de Blockchains. Dado que el coste en hardware de estas operaciones es muy alto y haría la operación muy costosa, se incluye el tanto por ciento que podría alquilarse a través del servicio más popular de poder de minado NiceHash, donde cualquier persona puede alquilar la capacidad de minar en las principales Blockchain, o de ceder recursos recibiendo pagos por ello.

En el caso de Bitcoin el tanto por ciento de poder de minado que puede conseguirse con NiceHash es virtualmente cero, ya que su red es demasiado grande, y controlar un tanto por ciento significativo (más del 50 %) tendría un coste hardware inmenso. En menor medida, la situación es parecida con casi todas las Blockchain relativamente conocidas. Pero esto es mucho más preocupante para Blockchain pequeñas ya que por cantidades a menudo ínfimas se puede controlar la red. Pero si el mercado no es muy grande tampoco será rentable para el atacante. Son las monedas con fuerte potencia en el mercado, pero pocos mineros, o un algoritmo de cifrado mejorable las que más tienen que perder por este tipo de ataques. [37]

Algunos de los mecanismos de defensa contra estos ataques son: la modificación del protocolo de consenso, la desincentivación y en el que nos centraremos ahora, el cifrado y resistencia a **dispositivos ASIC** (Application Specific Integrated Circuits). [38]

La idea es combatir la centralización del minado, lo cual actualmente se consigue a través de estos dispositivos hardware contra los que no existe competencia alguna. Estos dispositivos son diseñados para un propósito único, un único cálculo en el que son mucho mejores que cualquier dispositivo genérico (como una CPU o algo más eficiente como una GPU), incluso las más potentes supercomputadoras no pueden competir con un pequeño grupo de ASICs operando en paralelo. Están específicamente creados para minar un algoritmo de cifrado concreto, lo cual es extremadamente más eficiente.

Esto es completamente legal, y además tiene un grave riesgo el montar una granja con estos dispositivos de minado hardware si se hace a pequeña escala: si el valor de la Blockchain baja, el beneficio bajará mucho pero el coste del hardware y de la electricidad que consume será el mismo. Además, un fork en la red puede actualizar el mecanismo de cifrado y hacer que todas las ASICs fabricadas para ello se vuelvan completamente inútiles.

Todo esto hace que los tokens que se minan con ASIC tiendan a concentrarse en un pequeño grupo de mineros, y una fuerte inversión puede permitir a un pequeño grupo dominar la red. Esto afecta a casi todas las Blockchain que funcionan con PoW, empezando por Bitcoin y Ethereum.

Ante esto tenemos varias soluciones:

- La primera es la creación de algoritmos de cifrado específicamente diseñados para no poder minarse con un ASIC. Son los algoritmos y tokens denominados ASIC-resistentes.
- La segunda es la realización de forks (bifurcaciones de la cadena) frecuentes, pero por sorpresa, para desincentivar a la red. Y sobre todo al anunciarse un dispositivo cuyo objetivo es tu red (mayor problema es cuando no se anuncian y se desarrolla y pone en funcionamiento el hardware en secreto, como ocurrió con Monero).

Es habitual combinar las dos cosas. La lista de Blockchain que han implementado algoritmos para solucionar estos es muy larga y podemos encontrar en ella a Ethereum y Bitcoin Gold, entre otras.

Por todo ello, en el mundo de Blockchain, se dice que la resistencia a ASICs es prácticamente imposible al 100 %.

Robo de direcciones

A pesar de la seguridad que tiene la red debido a la encriptación, redundancia de datos y demás sistemas de protección mencionados en este documento, los equipos informáticos no dejan de ser vulnerables a ataques, de modo que al igual que un atacante puede forzar un sistema para obtener acceso a una cuenta en una red social, el software cliente de una Blockchain también puede ser atacado, de modo que ese nodo tendría un funcionamiento que la red no desea. En este caso, sin necesidad de alterar la cadena, se pueden redireccionar los beneficios de un nodo minero, por ejemplo, llevando esos beneficios a direcciones distintas a las programadas por sus dueños. Estos ataques son de los más comunes que sufren este tipo de software.

Vulnerabilidades software en Smart Contracts: Caso DAO

Ya se ha hablado en este documento de que son y cómo funcionan los Smart Contracts, que a grandes rasgos son programas software que se ejecuta de forma transparente al usuario y que, básicamente, dice que “si algo pasa, se haga esto otro”, por lo que finalmente no pueden evitar tener fallos en el software. La red Ethereum, como hemos expuesto, utiliza smart contracts.

Un caso muy famoso con el que ilustrar estas vulnerabilidades es “el caso DAO”. “The DAO” es una organización creada por un grupo de desarrolladores liderado por

Christoph Jentzsch. Esta organización creó un Smart Contract que fue desplegado sobre la red Ethereum, de modo que cualquiera pudiera vincular Ethers a él, algo que hicieron varios miles de personas a nivel mundial, con la intención de utilizarlo como un ahorro o inversión a largo plazo.

“The DAO” se regía por su código, siendo este la ley y este programa marcaba las normas de todo lo que se puede hacer o dejar de hacer. Todas las personas que utilizaron “The DAO” aceptaron el código fuente abierto de este programa, tanto como las normas a cumplir. Como se ha mencionado, al fin y al cabo, es software, y ninguna de estas personas se percató de un error en el código. Sin embargo, hubo alguien que detectó el error, el cual permitía extraer Ethers sin necesidad de permiso de los demás. No se trataba de una letra pequeña que nadie leyese, se trató de un error de programación del que nadie se percató.

Este error permitió que el atacante retirarse pequeñas cantidades de Ethers, hasta obtener cerca de 50 millones de dólares. Este mismo atacante publicó una nota abierta en Internet en la que declaraba que todo lo que había hecho estaba en el código y, por tanto, al ser ley, no podían retirarle todo ese dinero.

Este caso se llevó ante tribunales, los cuales respondieron que en ese tipo de programas el código no siempre es la ley, y que, si tantas personas habían puesto su dinero en un fondo común, tenían derecho a recuperar lo que es suyo. Aquí entra en juego el funcionamiento de la Blockchain, Ethereum en este caso, ya que, por gracia para el atacante y desgracia de las víctimas, no es posible identificar quién fue. [49]

Pérdida de clave privada

Perder los fondos almacenados en una cartera por un descuido o accidente, de modo que sean irrecuperables, es una posibilidad muy lejana, y eso es positivo y habla muy bien de los criptoactivos y su tecnología subyacente como herramientas disruptivas; pero la posibilidad existe aún, remotamente, pero sigue allí. La diferencia yace en que en Blockchain, cada individuo es enteramente responsable de los activos que posea. En casos extremos, las compañías no pueden hacer más que desearte suerte. [48]

En caso de perder esta clave privada el intento de recuperarla puede llegar a ser una odisea, ya que, si no la recuerdas y fallas numerosas veces los intentos, se podría generar una cuenta de espera que se automultiplicará, haciendo a su vez que quien quisiera recuperarla/robarla tendría que pasar toda su vida intentando acceder a la cuenta.

3. Ethereum



3.1. Historia y características

Dado que la plataforma utilizada para el posterior desarrollo del Smart Contract Travel es Ethereum es conveniente hacer hincapié en algunos puntos importantes para entender cómo funciona y sus aspectos fundamentales.

Esta plataforma de código abierto fue propuesta en 2013 y puesta en funcionamiento en 2015 por el programador ruso-canadiense Vitalik Buterin, con el objetivo de crear un instrumento para aplicaciones descentralizadas y colaborativas. Esta plataforma dispone de un token para reealizar transacciones llamado Ether, usado para realizar transferencias de valor y como incentivo por el trabajo de los mineros de la plataforma.

Por lo tanto, Ethereum es una plataforma descentralizada que permite a los usuarios desarrollar contratos inteligentes entre dos partes, basándose en la tecnología de la cadena de bloques y donde cada desarrollador puede crear y publicar aplicaciones distribuidas en la plataforma.

Ethereum se desarrolló desde el principio sobre una plataforma de transacciones transparentes. Si bien hay un "organismo" central que creó Ethereum, este no tiene autoridad sobre los mineros que contribuyen a la descentralización global de la plataforma. Esto significa que los nuevos protocolos y procesos necesitan ser aceptados por el colectivo, independientemente de lo que el organismo central crea que es mejor.

Ethereum se propuso desarrollar una plataforma descentralizada que animara a la comunidad de desarrolladores a trabajar a partir de lo que en ese momento era una nueva tecnología con Contratos Inteligentes y Dapps, que ofrecen mayores posibilidades con cadenas de bloques.

Una de las características principales de Ethereum es que permite tanto las transacciones permissionadas como las no permissionadas:

- Las transacciones sin permiso permiten que cualquier ordenador de la red Ethereum confirme la transacción.
- Las transacciones permissionadas son revisadas solo por un grupo selecto de ordenadores, de modo que no es necesario exponer toda la actividad a todos los ordenadores siempre y cuando se sigan los protocolos que se han establecido. [63]

Smart Contracts

“Un smart contract o contrato inteligente es un programa informático que ejecuta determinadas acciones preestablecidas en su código bajo ciertas condiciones. Acciones que han sido revisadas y aceptadas por las distintas partes que han “firmado” dicho contrato. De esta manera, el smart contract hace valer sus condiciones programadas presentando una respuesta acorde a sus cláusulas de forma completamente autónoma”. [68]

Si un contrato tradicional describe los términos de una relación, un contrato inteligente se asegura de que esos términos se cumplen escribiéndolos en código. Son programas que automáticamente ejecutan el contrato una vez que las condiciones predefinidas se cumplen, eliminando el retraso y el coste que existe al ejecutar un acuerdo de manera manual.

Por poner un ejemplo sencillo, un usuario de Ethereum podría crear un contrato inteligente para enviar una cantidad establecida de ether a un amigo en una fecha determinada. Escribirían este código en la cadena de bloques y cuando el contrato se complete (es decir, cuando llegue la fecha acordada) los ether se enviarán automáticamente.



Esta idea básica puede aplicarse a configuraciones más complejas, siendo su potencial probablemente ilimitado, con proyectos que ya han logrado un notable progreso en sectores como seguros, registros de propiedad, inmobiliarias, servicios financieros, servicios legales, etc.

Los contratos inteligentes también poseen varios beneficios adicionales:

- Eliminan la figura del intermediario, ofreciendo al usuario control total y minimizando los costes extra.
- Se registran, encriptan y replican en la cadena de bloques pública, donde todos los usuarios pueden ver la actividad.
- Eliminan el tiempo y esfuerzo que suponen los procesos burocráticos cuando se quieren realizar de forma manual. [64]

Minería

Actualmente la red Ethereum funciona con el protocolo de consenso de Prueba de Trabajo (**PoW**) a través del algoritmo **Ethash**. Dicho algoritmo es computacionalmente muy exigente y está orientado a la minería con tarjetas gráficas, es por esto, que en los comienzos la minería estaba realmente descentralizada.

Este algoritmo implementa una función hash llamada Keccak, también conocida como SHA-3. Como ya hemos visto anteriormente, en esta vía se busca usar elementos criptográficos de alta seguridad, como son la utilización intensiva de memoria y de caché. Es por esto que se dice que Ethash está orientado específicamente a combatir la minería a través de dispositivos dedicados **ASIC**, para así asegurar la descentralización de la red. [68]

Emisión de Ether

La plataforma Ethereum por el momento emite anualmente una cantidad limitada de criptomonedas, unos 18 millones de Ethers por año, o lo que es lo mismo, la minería de Ethereum puede generar un máximo de 18 millones de monedas nuevas cada año. Por el contrario, la emisión total no está limitada, es decir, es infinita. Para conseguir esta emisión, esta red cuenta con un sistema generador o **transacciones coinbase** algo característico. Como es lógico, si un minero encuentra la solución para minar un bloque recibirá 2 ETH como incentivo a seguir minando. Pero si se diese el caso de que otro minero también consigue minar ese bloque en ese preciso momento, este también recibirá una recompensa. A grandes rasgos, de este modo se ponen en circulación nuevas monedas en la red de Ethereum.

La preventa de activos que se hizo para promover la blockchain de Ethereum en su lanzamiento estuvo relacionada con la emisión inicial de Ether explicada anteriormente. En ese momento, se crearon 60 millones de Ethers, de los cuales 12 millones se usaron para crear un fondo de desarrollo lo que marcó el inicio de la **Fundación Ethereum** que hoy conocemos.

No obstante, esta fundación y su cadena de bloques está en continua evolución. Uno de las mejoras que observaremos en los próximos meses será la sustitución del sistema de minería de Prueba de Trabajo (PoW) para dar paso a la Prueba de Participación (PoS). Con este notorio cambio, la creación de nuevas monedas en la blockchain de Ethereum se hará de forma radicalmente diferente a la primigenia, prescindiendo de los mineros e incentivando una mayor participación económica en la red. Como veremos a continuación, este proceso es largo y tedioso, pero ya ha comenzado. [68]

Cambios en el protocolo

Los cambios de protocolo, también conocidos como bifurcaciones duras (hard forks), pueden ser "planificados" o "no planificados". Una razón para una bifurcación planificada puede ser adaptar el sistema para gestionar nuevas necesidades, introducir protocolos de seguridad o agilizar el proceso de minado, entre otras posibilidades. Las bifurcaciones no planificadas pueden ser el resultado del descubrimiento de fallas de seguridad que algunos consideran que no deben ser parcheadas, u otros eventos que

no llegan a un consenso sobre cómo abordarlos. Por ejemplo, un ataque cibernético puede alentar a los mineros a adoptar cambios en el protocolo, mientras que otros pueden querer mantener el protocolo antiguo y abordar los problemas según se presenten. El mayor ejemplo de esto es la ruptura entre Ethereum y Ethereum Classic.

Esta separación se produjo después de una manipulación del sistema en 2016 comentada anteriormente, que permitió el robo de Ether con un valor equivalente a 50 millones de dólares. Algunos querían cambiar el protocolo para que el dinero robado fuera inútil, mientras que otros querían seguir con los protocolos originales, alegando que el dinero fue robado usando un agujero en el protocolo. Esta ruptura se conoce como el Evento DAO ya que el objeto del robo fue la Organización Autónoma Descentralizada (DAO, por sus siglas en inglés).

Ethereum Classic (ETC) se basa en el protocolo original y ha sido gestionada por un colectivo que intenta mantenerse fiel a la versión original de Ethereum. Ethereum (ETH) tiene un grupo de supervisión llamado Fundación Ethereum que sigue adelante y continúa desarrollando la plataforma. [63]

Casper y Ethereum 2.0

Casper es una actualización de la Red Ethereum, también llamada **Ethereum 2.0** o **Serenity**.

Esta actualización viene a intentar solucionar los **principales problemas a los que se enfrenta la red Ethereum actual, fundamentalmente relativos al mecanismo de consenso de prueba de trabajo (PoW)** que utilizan la mayor parte de las criptomonedas.

Sus carencias más importantes son:

- **Escalabilidad:** Es el problema más importante al que se enfrenta actualmente la red Ethereum, al soportar cientos de aplicaciones descentralizadas por lo que precisa procesar un gran número de transacciones por segundo.

Al haber crecido el uso de la red Ethereum, debido a la creación de más Dapps y la ejecución de muchas transacciones, ha aumentado también el tiempo y el coste de las transacciones, por lo que es necesario aumentar la velocidad de estas para que la red sea adoptada masivamente por los usuarios.

- **Sostenibilidad medioambiental:** Originado por el importante gasto de energía que viene dado por la necesidad de que los mineros, al usar estos mecanismos PoW, cuenten con equipos informáticos muy potentes que les permitan ganar recompensas, incidiendo así en el delicado equilibrio medioambiental y el cambio climático.
- **Descentralización:** Los mineros más potentes y rápidos tienen más probabilidad de validar transacciones, crear nuevos bloques y ganar recompensas. Por este motivo la minería de las criptomonedas que emplean la PoW se agrupan en unos pocos grupos de mineros que, en el caso de Ethereum, representan el 70% de la tasa de hash, yendo en contra de la filosofía de descentralización que caracteriza a las criptomonedas.

Gracias a la actualización de Casper, la blockchain de Ethereum pasará al algoritmo Proof of Stake (Prueba de Participación).

Las diferencias de Casper con otros protocolos PoW viene dada, en primer lugar, por la sustitución de los mineros por validadores, logrando el consenso a través del voto de los mismos. Cómo en cualquier algoritmo basado en PoS, el voto de cada validador dependerá de la cantidad de Ethers depositados, es decir, de su participación.

El mecanismo funciona de la siguiente manera:

1. Los validadores apuestan una parte de sus Ethers;
2. Después, comenzarán a validar los bloques que se van a añadir a la cadena;
3. Una vez que hayan añadido un bloque, recibirán una recompensa proporcional a sus apuestas.

Una característica muy interesante de Casper, que le diferencia de la mayoría de los otros protocolos PoS, es que está diseñado para funcionar en un sistema sin confianza y ser más tolerante a fallas bizantinas.

Esto significa que **utiliza un proceso mediante el cual se puede castigar a los validadores maliciosos**. Es decir, si un validador actúa de manera malintencionada será castigado de inmediato y se reducirá todo lo que está en juego.

Lo mismo pasa si los validadores pasan inactivos con su nodo largos periodos de tiempo. El descuido o la pereza les hará perder su apuesta.

Según noticias actualizadas, Ethereum 2.0 debería ser lanzado en 3 fases, dentro del 2022.

- **Ventajas:** Como adelantamos, la migración del mecanismo de consenso de Proof of Work a Proof of Stake de Casper solucionará problemas relacionados con la **escalabilidad**, aumentando la velocidad de las transacciones y garantizando la adopción masiva de Ethereum; la **sostenibilidad medioambiental**, ya que no necesitando equipos potentes ahorrará gastos de energía y la contaminación ambiental; la **descentralización**, evitando el problema de que la minería se concentre en manos de pocos y vaya en contra de la descentralización.

Otra gran ventaja de Casper es la **mejora del nivel de seguridad de la blockchain**. Con la prueba de participación es difícil que un ataque del 51% tenga lugar ya que, gracias al mecanismo de castigo de los validadores, no valdría la pena intentarlo.

- **Desventajas:** Al principio Casper **no promete una verdadera descentralización** ya que la participación requiere inicialmente un **depósito mínimo de 1,500 ETH**, (aproximadamente \$3172,47). Además, los validadores más ricos son elegidos con más frecuencia, ganando más recompensas.

Esto hace que la participación de Ethereum esté fuera del alcance del usuario promedio, **favoreciendo el dominio de las llamadas «ballenas»**. Según Vitalik Buterin el requisito mínimo se reducirá a 32 ETH una vez que Ethereum alcance el 100% de PoS.

Con Casper los validadores bloquean ETH en un contrato inteligente de durante 3 a 12 meses. **La volatilidad del precio de ETH expone a los validadores a un riesgo de iliquidez** significativo, desfavoreciendo su participación. [83]

3.2. Funcionamiento

Ethereum Virtual Machine (EVM)

La Máquina Virtual de Ethereum (EVM) es uno de los componentes primordiales en el funcionamiento de la cadena de bloques de Ethereum. Su cometido es el de permitir la ejecución de programas o contratos inteligentes con el objetivo de desplegar sobre dicha red un conjunto de funcionalidades extras para la mejora de la experiencia de usuario.

Esta máquina virtual que conforma el entorno de la cadena de bloques de Ethereum es capaz de ejecutar una extensa gama de instrucciones proporcionando gran flexibilidad en el momento de procesar distintas operaciones.

Otra herramienta clave del entorno Ethereum es el lenguaje de programación **Solidity**. Este lenguaje especializado de alto nivel fue concebido para facilitar la programación para esta máquina virtual y hacer más sencilla la creación de los **contratos inteligentes**. La EVM primeramente traduce este lenguaje de programación a los llamados **códigos de operación (OP_CODES)** para posteriormente transformarlos al **bytecode**, que será el código “entendido” y ejecutado por la máquina virtual para llevar a cabo las operaciones definidas en los contratos inteligentes.

En resumen, con la máquina virtual de Ethereum se consigue que la cadena de bloques de esta plataforma se convierta en una especie de gran ordenador global descentralizado debido a su característica de almacenamiento de contratos inteligentes en los nodos de esta, ejecutándose las órdenes programadas en los mismos, pudiendo resolver casi todo tipo de problemas computacionales dentro de este “gran ordenador” que conforma la red de Ethereum. [65]

La máquina virtual de Ethereum reúne unas características singulares que exponemos a continuación:

1. En primer lugar, **ayuda a proporcionar un alto nivel de seguridad**. Al ser una máquina virtual con limitaciones en las instrucciones y en la forma en cómo se ejecutan, EVM es capaz de ejecutar códigos no confiables sin consecuencias desastrosas.
2. **EVM es una construcción completamente descentralizada**. Cada nodo dentro de la red Ethereum ejecuta una copia de esta máquina virtual en conjunto y sincronía con el resto de nodos que forman la red. Esto garantiza que las instrucciones dadas las EVM se ejecuten siempre y cuando exista al menos un nodo en activo. Ello permite el acceso a dicho sistema desde cualquier parte del mundo, resistiendo la censura y garantizando el acceso a los recursos de la red. Además no requiere de la participación de terceros, tampoco pueden ser modificadas ni alteradas.

3. **La EVM permite el desarrollo de una mayor cantidad de aplicaciones**, y que éstas puedan ejecutarse sobre una misma red blockchain, sin afectar otras operaciones. Lo vemos claramente en el funcionamiento de las DApps de Ethereum y su explosión de desarrollo hasta la actualidad.
4. **EVM es capaz de hacer realidad a los smart contracts o contratos inteligentes de Turing completo.** Estos son programaciones específicas e invariables que pueden ejecutarse y hacerse cumplir por sí mismos, de una manera autónoma y automática.
5. **La EVM posee la capacidad de ejecutar una serie de códigos de operación u OP_CODES bien definidos**, fuera de estos códigos la EVM no es capaz de realizar absolutamente nada. Pero para que esta ejecución sea posible, dichos OP_CODES deben transformarse en bytecode o código máquina. Esto permite que EVM tenga un mayor rendimiento en la realización de operaciones en comparación con aproximaciones como las de Bitcoin.

Así, el **funcionamiento de la Ethereum Virtual Machine es un proceso bastante complejo**. Sin embargo, puede simplificarse de la siguiente manera: Primeramente, **los desarrolladores de EVM han creado un lenguaje de programación llamado Solidity**. Este lenguaje de programación de alto nivel es muy parecido al JavaScript, un lenguaje muy común en el ámbito de la programación Web. El fin de este lenguaje es **facilitar la tarea de escribir código para los contratos inteligentes de Ethereum y las aplicaciones descentralizadas asociadas a los mismos**.

En consecuencia, contamos con un proceso en el que el desarrollador usa Solidity para escribir sus contratos inteligentes de forma rápida y sencilla. Pero a partir de este punto, **este desarrollador usa herramientas especiales que convierten este código en OP_CODES y finalmente en bytecodes para que la EVM pueda llevar a cabo todas las instrucciones que se han programado**.

Así cada transacción y operación que se realiza en la blockchain de Ethereum pasa por este proceso, y su ejecución es llevada a cabo por la EVM, que al terminar graba todo en la blockchain de Ethereum, dejando registro público de esas operaciones. En pocas palabras, **cada transacción en Ethereum, desde un envío de Ether hasta alguna acción en un smart contract o DApps es llevada a cabo por la EVM. No existe nada en Ethereum que no pase por esta máquina virtual.** [77]

Tipos de cuentas

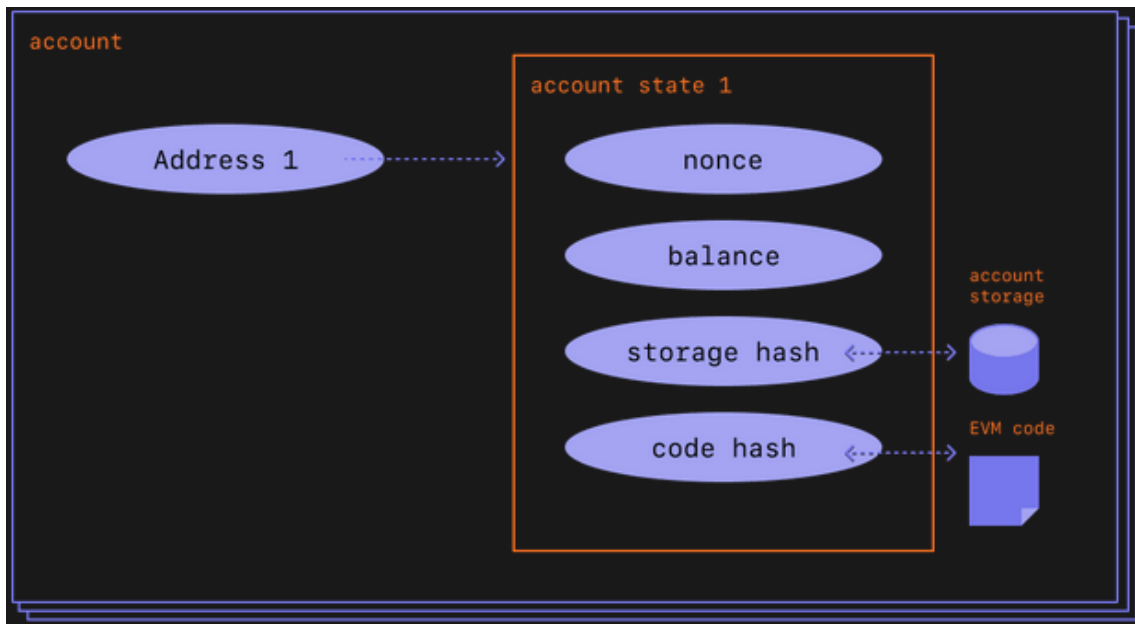
Por definición, una cuenta de Ethereum es una entidad con un saldo de Ether (ETH) que puede enviar transacciones en Ethereum. Las cuentas pueden ser controladas por el usuario o implementadas como contratos inteligentes. Es decir, en Ethereum existen dos tipos de cuentas, ambas con la capacidad de recibir, mantener y enviar ETH y tokens; y ambas con la capacidad de interactuar con contratos inteligentes implementados. Los dos tipos y sus propiedades son:

- De propiedad externa: Controlada por cualquier persona que tenga las claves privadas.
 - Crear una cuenta no tiene costo alguno.

- Puede iniciar transacciones
- Las transacciones entre cuentas de propiedad externa solo pueden ser transferencias de ETH.
- Contrato: Un contrato inteligente implementado en la red, controlado por código.
 - Crear un contrato tiene un costo porque está usando almacenamiento en la red.
 - Solo puede enviar transacciones en respuesta a recibir una transacción, es decir, los smart contracts son cuentas reactivas.
 - Las transacciones de una cuenta externa a una cuenta de contrato pueden activar un código que puede ejecutar muchas acciones diferentes, como transferir tokens o incluso crear un nuevo contrato.

Las cuentas de Ethereum se definen a través de cuatro componentes:

- Nonce: Contador que indica el número de transacciones enviadas desde la cuenta. Esto asegura que las transacciones solo se procesen una única vez. En una cuenta de contrato, este número representa el número de contratos creados por la cuenta.
- Balance: El número de *wei* propiedad de esta dirección. $1\text{Ether} = 10^{18}\text{Wei}$.
- codeHash: Este hash se refiere al código de una cuenta en la máquina virtual de Ethereum (valor inmutable del hash asociado a la cuenta). Las cuentas de contrato tienen fragmentos de código programados que pueden realizar diferentes operaciones. Este código EVM se ejecuta si la cuenta recibe una llamada de mensaje. No se puede cambiar a diferencia de los otros campos de la cuenta. Todos esos fragmentos de código están contenidos en la base de datos de estado con sus correspondientes hashes para su posterior recuperación. Este valor hash se conoce como codeHash. Para las cuentas de propiedad externa, el campo codeHash es el hash de una cadena vacía.
- storageRoot: A veces conocido como hash de almacenamiento. Es un hash de 256 bits del nodo raíz de un árbol Merkle Patricia que codifica el contenido de almacenamiento de la cuenta (un mapeo entre valores enteros de 256 bits). Este codifica el hash del contenido de almacenamiento de esta cuenta y está vacío de forma predeterminada.



Como ya hemos visto anteriormente, una cuenta está formada por un par de claves criptográficas: pública y privada. Ayudan a demostrar que una transacción fue realmente firmada por el remitente y previenen las falsificaciones. Su clave privada es lo que usa para firmar transacciones, por lo que le otorga la custodia de los fondos asociados con su cuenta. Realmente nunca se poseen criptomonedas, sino claves privadas: los fondos siempre están en el libro mayor de Ethereum.

Esto evita que los actores malintencionados difundan transacciones falsas porque siempre se puede verificar el remitente de una transacción. Por ejemplo:

Si Alicia quiere enviar Ether desde su propia cuenta a la cuenta de Bob, Alicia necesita crear una solicitud de transacción y enviarla a la red para su verificación. El uso de Ethereum de la criptografía de clave pública asegura que Alicia pueda demostrar que originalmente inició la solicitud de transacción. Sin mecanismos criptográficos, una adversaria maliciosa Eva podría simplemente transmitir públicamente una solicitud del estilo a “enviar 5 ETH” de la cuenta de Alicia a la cuenta de Eva”, y nadie podría verificar que no provenga de Alicia.

Por último, cabe resaltar que una cuenta no es una billetera (wallet). Una billetera es el par de claves asociado con una cuenta propiedad del usuario, que le permite a un usuario realizar transacciones desde la cuenta o administrar la misma.

Transacciones en Ethereum

Como ya sabemos, las **transacciones** son una parte fundamental dentro de una cadena de bloques ya que todo lo que rodea a la cadena está diseñado para asegurar que las transacciones puedan ser creadas, propagadas por la red, validadas y añadidas al registro o cadena de bloques.

Cada Blockchain posee una estructura diferente para sus transacciones, de modo que no se puede definir una estructura de una transacción para Blockchain. Por ello existen diferentes **modelos de transacciones**.

Uno de estos modelos de transacciones, utilizado por Bitcoin y la mayoría de las Blockchains, es el **UTXO (Unspent Transaction Output)**. Una UTXO, o transacción pendiente de gasto, es una salida (de fondos) que un usuario recibe para poder gastar en el futuro como una entrada para alguien más. [20]

Por otro lado, Ethereum, en lugar de utilizar UTXO's, utiliza un modelo diferente, el **modelo basado en cuentas (Account Based Model)**. Este modelo es similar a como funciona el sistema de banca tradicional. Básicamente cada cuenta vive mediante transferencias directas de valor e información con transiciones de estado. [21]

Este modelo puede ser explicado de una forma muy simple: supongamos que Bob y Alice son amigos y Bob quiere mandarle 5 tokens a Alice. Partimos de que Bob posee 10 tokens en su cuenta y Alice ninguno. Entonces, Bob le manda 5 de sus 10 tokens a Alice, de modo que Bob se queda con 5 tokens, el resto tras extraer los 5 que se envían a Alice. Finalmente, tanto Bob como Alice tienen 5 tokens cada uno. Cabe destacar que existen dos tipos de cuentas: cuentas de usuario controladas por clave privada y cuentas de usuario controladas por código de contrato (smart contracts). Y es por esto que Ethereum utiliza este modelo basado en cuentas.

Como ya sabemos, Ethereum utiliza un lenguaje de programación completo (Solidity) y una de sus características principales son los smart contracts, obteniendo una mayor simplicidad utilizando este modelo sobre UTXO. Hay una gran cantidad de aplicaciones descentralizadas que poseen código y estado arbitrario y propio, es por ello que utilizar UTXO disminuiría la habilidad de ejecución de los smart contracts.

Cada una de estas cuentas tiene su propia contabilidad, almacenamiento y espacio de código para llamar a otras cuentas o direcciones. Una transacción será válida inicialmente si el remitente de la misma posee suficientes medios (fondos, por ejemplo) para pagar por ello.

Para hacernos una idea de la posible estructura de una transacción y entender mejor este concepto, veremos la composición de una transacción de la red de Ethereum.

El término transacción en Ethereum hace referencia a un paquete de datos firmados que almacena un mensaje enviado desde una cuenta externa, es decir, una cuenta administrada por un humano, no un contrato. Por ejemplo, si Bob envía a Alice 1 ETH, se debe endeudar la cuenta de Bob y acreditar la de Alice. Esta acción de cambio de estado tiene lugar dentro de una transacción.



Las transacciones, que cambian el estado del EVM, deben transmitirse a toda la red. Cualquier nodo puede transmitir una solicitud para que se ejecute una transacción en el EVM; después de que esto suceda, un minero ejecutará la transacción y propagará el cambio de estado resultante al resto de la red.

Entonces una transacción enviada incluye la siguiente información:

- El receptor del mensaje: la dirección de recepción (si es una cuenta de propiedad externa, la transacción transferirá valor; si es una cuenta de contrato, la transacción ejecutará el código del contrato).
- La firma digital que identifica al emisor: el identificador del remitente. Esto se genera cuando la clave privada del remitente firma la transacción y confirma que el remitente ha autorizado esta transacción.
- Value: Cantidad de fondos para transferir del remitente al destinatario (en wei)
- Data: Campo opcional para incluir datos arbitrarios.
- gasLimit: La cantidad máxima de unidades de gas que puede consumir la transacción. Las unidades de gas representan pasos computacionales, es decir, este campo indica el máximo número de pasos computacionales que se permite ejecutar a la transacción.
- gasPrice: Representa el valor máximo que desea pagar el remitente en forma de comisión por unidad de gas, es decir, por cada paso computacional.

```

1  {
2    from: "0xEA674fdDe714fd979de3EdF0F56AA9716B898ec8",
3    to: "0xac03bb73b6a9e108530aff4df5077c2b3d481e5a",
4    gasLimit: "21000",
5    gasPrice: "200",
6    nonce: "0",
7    value: "10000000000",
8  }

```

El gas es una referencia al cálculo requerido para procesar la transacción por parte de un minero. Los usuarios deben pagar una tarifa por este cálculo. El gasLimit y el gasPrice determinan la tarifa máxima para la transacción que se paga al minero. [76]

Ethereum introdujo el concepto de gas con una doble finalidad:

- Pagar a los mineros por validar transacciones.
- Evitar que la ejecución de una transacción bloquee el sistema. Ésto se debe al hecho de que como el lenguaje de programación en el que se programan los contratos de Ethereum es Turing completo (como ya hemos comentado) se podría dar el caso de que un contrato se ejecutara indefinidamente, con un bucle infinito, por ejemplo, bloqueando así la red.
- Evitar que malos actores intenten saturar la red enviando una inmensa cantidad de transacciones haciendo que paguen por el uso del sistema.

Dado que las transacciones requieren de gas para ser ejecutadas en la red de Ethereum, este gas se paga como comisión en forma de ether (la moneda de Ethereum) por el uso de la red. Las transacciones necesitan proveer gas suficiente para cubrir toda la computación y almacenaje requerida por la transacción. Si durante la ejecución de la transacción se requiere más gas de la cantidad máxima indicada, la ejecución será abortada y el contrato quedará en el mismo estado que estaba previo a la ejecución de dicha transacción. Es decir, las instrucciones enviadas con una transacción se ejecutan de forma atómica.

Además de la cantidad de gas máxima, el usuario debe indicar el precio que está dispuesto a pagar por cada unidad de gas. El precio indicado es importante porque hará que un minero esté dispuesto a procesar nuestra transacción antes o después de otras transacciones dependiendo del precio que estemos dispuestos a pagar.

Terminar diciendo que Ethereum además permite la comunicación entre contratos mediante mensajes. Un mensaje es un conjunto de instrucciones enviadas a un contrato desde otro contrato y es muy importante tener claro que un mensaje no constituye una transacción y es enviado cuando un contrato está siendo ejecutado por la EVM. [19]

Tipos de redes

Como ya sabemos, al ser Ethereum un protocolo, pueden existir múltiples “redes” independientes que se ajusten a dicho protocolo y que no interactúen entre sí.

Las redes son diferentes entornos de Ethereum a los que podemos acceder para casos de uso de desarrollo, prueba o producción. La cuenta de Ethereum funcionará en las diferentes redes, pero el saldo de la cuenta y el historial de transacciones no se transferirán de la red principal de Ethereum. Por ello es útil saber qué redes están disponibles con fines de prueba y cómo obtener acceso a estas testnets para poder probar en ellas antes de lanzar cualquier desarrollo a la red principal.

- **Redes públicas:**

Estas redes son accesibles para cualquier persona en el mundo con conexión a Internet. Cualquiera puede leer o crear transacciones en una cadena de bloques pública y validar las transacciones que se están ejecutando. Y como ya hemos visto, el acuerdo sobre las transacciones y el estado de la red se decide por consenso de pares.

- **Mainnet:**

Es la cadena de bloques de producción pública principal de Ethereum, donde las transacciones de valor real ocurren en el libro mayor distribuido. Cuando los exchanges de criptomonedas y las personas discuten acerca de los precios de ETH, están hablando de esta red principal de Ethereum.

- **Redes de prueba o testnets:**

Además de la red principal, existen redes de prueba públicas. Estas son redes utilizadas por desarrolladores de protocolos o desarrolladores de contratos inteligentes para probar tanto las actualizaciones del protocolo como los posibles contratos inteligentes en un entorno similar a producción antes de la implementación en la red principal.

En general, es importante probar cualquier código de contrato que escriba en una red de prueba antes de implementarlo en la red principal. Si está creado una aplicación descentralizada que se integra con los contratos inteligentes existentes, la mayoría de los proyectos tienen copias implementadas en testnets con los que poder interactuar.

La mayoría de las redes de prueba utilizan un mecanismo de consenso de prueba de autoridad. Lo que significa que se elige una pequeña cantidad de

nodos para validar transacciones y crear nuevos bloques, poniendo su identidad en el proceso. Esto es así dado que es difícil incentivar la minería a través del mecanismo de prueba de trabajo en una red de prueba ya que puede dejarla vulnerable.

Algunas son:

- Görli
- Kovan
- Rinkeby
- Ropsten: Esta red es la única de las anteriores que funciona con el método de consenso de prueba de trabajo (PoW) lo que significa que es la representación más homogénea de la red principal de Ethereum, y la que usaremos posteriormente para pruebas.

▪ Redes privadas:

Una red de Ethereum es una red privada si sus nodos no están conectados a una red pública (es decir, mainnet o testnet). Por lo tanto, en este contexto, privado significa aislado, en lugar de protegido o seguro.

• Redes de desarrollo:

Para desarrollar una aplicación de Ethereum, lo normal es que se quiera probar en un entorno local, es decir, en una red privada para ver cómo funciona antes de desplegarla en producción. De manera similar a cuando nos creamos un servidor local en nuestro ordenador para el desarrollo web, podemos crear una instancia de blockchain local para probar nuestra aplicación descentralizada. Esto permite una interacción mucho más rápida que en una red de prueba pública.

Con este tipo de redes podemos ejecutar un nodo (como Geth u OpenEthereum), pero dado que estas redes se construyen para fines de desarrollo, estas tienen unas características especiales como:

- Siembra de datos deterministas en la blockchain local (por ejemplo, cuentas con saldos ETH).
- Extraer bloques instantáneamente con cada transacción que recibe, en orden y sin demora.
- Funcionalidad mejorada de depuración y registro.

Una de las herramientas más utilizadas para este fin, es *ganache*, que forma parte de un framework para desarrolladores de Ethereum llamado *truffle*.

▪ Redes de consorcios:

En este tipo de redes el consenso está controlado por un conjunto predefinido de nodos que son confiables. Por ejemplo, una red privada de instituciones académicas conocidas, cada una de las cuales gobierna un solo nodo, y los bloques son validados por cierta cantidad de firmantes dentro de la red.

Si una red pública de Ethereum es como un Internet (público), podemos entender este tipo de redes como una especie de Intranet (privada). [74]

3.3. Herramientas para el desarrollo

Lenguajes de programación de Smart Contracts

Un gran aspecto de Ethereum es que los contratos inteligentes se pueden programar utilizando lenguajes relativamente amigables para los desarrolladores. Los dos lenguajes más activos y con mayor soporte son:

- Solidity:
 - Lenguaje de alto nivel orientado a objetos.
 - Lenguaje de corchetes que ha sido profundamente influenciado por C++.
 - Tipado estático (los tipos de variables se conocen en el momento de la compilación).
 - Soporta:
 - Herencia (puede aumentar la funcionalidad de otros contratos).
 - Bibliotecas (puede crear código reutilizable que se puede llamar desde diferentes contratos, como funciones estáticas de una clase estática en otros lenguajes de programación orientados a objetos).
 - Tipos complejos definidos por el usuario.
- Vyper:
 - Lenguaje de programación Pythonico.
 - Fuertemente tipado.
 - Código del compilador pequeño y comprensible.
 - Deliberadamente tiene menos funciones que Solidity con el objetivo de hacer que los contratos sean más seguros y fáciles de auditar. Vyper no admite:
 - Modificadores
 - Herencia
 - Ensamblaje en línea
 - Sobrecarga de funciones
 - Sobrecarga de operadores
 - Llamadas recursivas
 - Bucles de longitud infinita
 - Puntos fijos binarios

Remix Ethereum

Remix IDE es una herramienta de código abierto para desarrollar, compilar y desplegar contratos inteligentes con el lenguaje Solidity en redes de Ethereum directamente desde el navegador (aunque también tiene versión de escritorio). Es una plataforma realmente útil para hacer pruebas, depuración y despliegue de contratos inteligentes.

Esta herramienta escrita en JavaScript fomenta un ciclo de desarrollo rápido y tiene un amplio conjunto de complementos con interfaces intuitivas. Se utiliza para todo el viaje del desarrollo de contratos, además de ser un patio de recreo para aprender y enseñar Ethereum.

Después de indicarle la versión del compilador que necesita para el código en concreto, Remix actúa como otros IDEs, mostrando los errores que contiene el código según en la versión que esté el compilador y las posibles correcciones que solucionarían dicho error. La interfaz que proporciona para interactuar con los contratos es bastante simple y su ventaja principal es que permite desplegar y borrar contratos tantas veces como nos sea necesario, cosa que en la red principal no es posible.

Para acceder a esta plataforma basta con dirigirse a nuestro navegador web y escribir la dirección: <https://remix.ethereum.org>

Metamask

Ethereum ha sido capaz de crear un amplio y rico ecosistema de aplicaciones descentralizadas a su alrededor. Sin embargo, el uso de estas DApps siempre había resultado difícil y poco amigable para el usuario. Algo que han conseguido revertir con MetaMask y su capacidad de simplificar el uso de DApps gracias a una sencilla extensión para navegadores web.

Para que el usuario pueda relacionarse con las DApps de cadenas de bloques como la de Ethereum, estas aplicaciones precisan de un enlace o puente, y esta es la valiosa utilidad de Metamask, que facilita la utilización de estas aplicaciones distribuidas desde el navegador que en cada caso utilice el usuario, puesto que esta herramienta aunque originalmente empezó siendo una extensión de Chrome, actualmente tiene soporte para otros navegadores como Firefox, Opera o Brave, acercando así las DApps al público en general.

Aaron Davis y Dan Finlay fueron los encargados de diseñar y desarrollar Metamask a mediados de 2016, definiendo de manera revolucionaria las bases para que cada usuario pudiera utilizar su navegador web como interfaz para comunicarse con sus aplicaciones descentralizadas preferidas de manera rápida, sencilla y segura.

Para ello, MetaMask usa la interfaz y API web de Ethereum, *web3.js*. Esta librería oficial de Ethereum sería la base fundamental del mundo de posibilidades ofrecidas por MetaMask. Gracias a ella sería posible crear un proxy o puente comunicacional entre las DApps, MetaMask y los usuarios.

El principal escollo técnico que se encontraron los desarrolladores fue garantizar la seguridad necesaria, que no es poca, para un correcto funcionamiento de la herramienta, logrando a mediados de julio de 2016 mostrar su primera versión para el navegador Chrome. Algo después se implementó esta extensión para el navegador Firefox y desde entonces ha evolucionado hacia una interfaz tan fácil de usar que ni siquiera requiere configuración previa por parte del usuario.

Como ya hemos mencionado, el funcionamiento de esta herramienta tiene su fundamento en una librería oficial de Ethereum denominada *web3.js*. Fue concebida con el claro objetivo de permitir el desarrollo de aplicaciones web capaces de relacionarse con la cadena de bloques de Ethereum a través de un monedero, logrando su objetivo estableciendo una línea de comunicación entre el plugin y la DApp para que una vez reconocida la extensión, se habilite en el navegador automáticamente pudiendo ser fácilmente utilizada por el usuario, que podrá llevar a cabo todas las operaciones habilitadas por la DApp que abarcan desde la compra-venta de tokens hasta el acceso a todo tipo de servicios ofrecidos por dicha aplicación

descentralizada. Como ya sabemos, cada una de estas operaciones suele conllevar un precio que debe ser abonado en Ethers o en el token especificado para dicha acción. De cualquier forma, Metamask tiene la capacidad de gestionar tales interacciones.

En conclusión, esta herramienta no solo nos provee de una billetera de criptomonedas, sino que gestiona cada interacción del usuario con la aplicación descentralizada, llevando a cabo las operaciones necesarias para que estas interacciones se realicen de forma satisfactoria a través de un canal de comunicación seguro gracias al uso de una fuerte criptografía, dado que Metamask tiene la capacidad de generar sus propias claves asimétricas, almacenarlas localmente y gestionar el acceso a las mismas. Debido a todo lo anterior, esta herramienta es realmente segura. [67]

Truffle Suite

Truffle es un conjunto de herramientas (framework) de programación dirigidas a contratos inteligentes para desarrollar aplicaciones sostenibles y profesionales sobre la blockchain utilizando para ello la Máquina Virtual de Ethereum, así como realizar las distintas pruebas en un entorno de desarrollo integrado y amigable para el desarrollador.

El objetivo principal de Truffle es proveer un entorno de desarrollo en la blockchain que facilite la labor de los desarrolladores que se dedican a la creación de aplicaciones (DApps) y contratos inteligentes en Ethereum.

Truffle provee un marco de prueba y una canalización de activos para Ethereum, que hacen que el proceso de desarrollar aplicaciones para esta red sea un proceso más sencillo e intuitivo.

En la medida que evoluciona, el equipo de Truffle añade nuevas herramientas y características a este entorno, con el objetivo de que los desarrolladores tengan dentro del mismo espacio de trabajo todo lo necesario para la creación, prueba, simulación y otras tareas que permitan afinar las aplicaciones antes de ponerlas a disposición de los usuarios finales.

Minimizar los inconvenientes que se derivan de las pruebas de contratos inteligentes en Ethereum, que generalmente se desarrollan utilizando su propio lenguaje de programación Solidity.

En este sentido, Truffle permite a los desarrolladores realizar estas pruebas sobre la Máquina Virtual de Ethereum (EVM), para resolver estos inconvenientes cuando se trata de probar e implementar en la red Ethereum, especialmente aquellos proyectos más grandes con múltiples contratos.

El entorno de trabajo de Truffle consta de tres componentes principales:

1. Truffle: proporciona una herramienta de desarrollo con la capacidad de probar e implementar los proyectos. Esta herramienta ha crecido notablemente en popularidad.
2. Ganache: Ganache es una blockchain de Ethereum personal que se utiliza para probar contratos inteligentes donde puede implementar contratos, desarrollar aplicaciones, ejecutar pruebas y realizar otras tareas sin ningún costo debido a que corre dentro de un servidor local.

3. Drizzle: es una colección de bibliotecas en JavaScript que se agrupan un amplio conjunto de funciones que se utilizan para el desarrollo Front-End que se puede conectar a los datos de un contrato inteligente.

En general, todo este conjunto de herramientas conforma la Suite de Truffle, y con ella se pueden realizar operaciones como:

- Soporte integrado para compilar, implementar y vincular contratos inteligentes.
- Prueba de contrato automatizada.
- Admite aplicaciones de consola y aplicaciones web.
- Gestión de red y gestión de paquetes.
- Consola Truffle para comunicarse directamente con contratos inteligentes.
- Admite una estrecha integración.
- Canalización de compilación configurable con soporte para procesos de compilación personalizados.
- Marco de implementación y migraciones programables.
- Consola interactiva para comunicación contractual directa.
- Reconstrucción instantánea de activos durante el desarrollo.
- Ejecutor de scripts externos que ejecuta scripts dentro de un entorno Truffle.

Por lo tanto, con la Suite de Truffle se tiene un entorno de desarrollo basado en la blockchain de Ethereum, en el que se puede desarrollar DApps, compilar contratos, implementar contratos, inyectarlos en una aplicación web, crear front-end para DApps y realizar pruebas. [78]

3.4. DApps, NFTs y más

Aplicaciones descentralizadas (DApps)

Las DApps son aplicaciones que funcionan en base a una red descentralizada en la que cada usuario tiene el control total sobre el funcionamiento de esta y que de manera muy segura permite al usuario el acceso a todos sus servicios, y ello a través tanto de ordenadores personales como smartphones y/o aplicaciones web, no dependiendo de servidores centrales sino de nodos que interrelacionan entre sí.

Sin embargo, este concepto no es novedoso dado que las DApps primigenias se originaron con los protocolos de compartición de archivos como BitTorrent (sistemas peer-to-peer que ya hemos mencionado). Aunque verdaderamente fue Bitcoin la primera aplicación descentralizada basada en la tecnología blockchain.

No obstante, la industria de las aplicaciones descentralizadas no ha llegado a su máximo esplendor hasta la presentación de Ethereum en 2014, el desarrollo de su lenguaje Solidity y con ello, la posibilidad de implementación y ejecución de contratos inteligentes. De este modo, las DApps que funcionan con esta tecnología de bloques se encuentran en continuo e imparable crecimiento.

Las DApps y las Apps tradicionales tienen muchos elementos en común, sin embargo, su diferencia radica en cómo interactúan con dichos elementos. Estos son:

- Frontend:

Esta primera capa, viene a ser la interfaz que los usuarios utilizan para interactuar con la aplicación. En este caso, tanto las DApp como las App tradicionales, pueden hacer uso de los inmensos recursos gráficos existentes para ello. La finalidad de esta capa es simplemente, dar al usuario la capacidad de interactuar, recibir y enviar información a la aplicación que esté usando.

- Backend:

Este segundo nivel, se refiere a la lógica principal de la aplicación, resultando centralizada en el caso de las aplicaciones tradicionales y descentralizada en las aplicaciones distribuidas, debido a que estas últimas se encuentran estrechamente relacionadas con los contratos inteligentes ejecutados sobre una cadena de bloques como Ethereum. De este modo, un contrato inteligente encapsula la programación que garantiza que la DApp se ejecute conforme a lo especificado en el mismo, de forma transparente y segura debido a, como ya sabemos, los contratos inteligentes son visibles y públicos.

El backend es soportado por las API (Interfaz de Programación de Aplicaciones) y capacidades de la blockchain. Por ejemplo, en Ethereum como ya hemos visto, existen diversas API para controlar la interacción con el usuario con las capas de almacenamiento o autenticación por poner algunos ejemplos.

- Almacenamiento de datos:

En una aplicación tradicional, esta capa también es centralizada. Normalmente los datos son almacenados en el computador del usuario o en servidores controlados por terceros. Esta forma de trabajo, tiene muchos puntos débiles como ya sabemos.

En este punto, la diferencia básica entre las aplicaciones tradicionales y las DApps, es que, en el primer caso, los datos se almacenan en servidores de terceros o en la propia memoria del ordenador del usuario, resultando como ya sabemos susceptible de fallos y ataques al sistema con la consecuente pérdida de información, mientras que en las segundas este almacenamiento de información se realiza de forma absolutamente descentralizada; así cada usuario de la aplicación descentralizada posee un historial completo de las transacciones que se van llevando a cabo en la red. Además de lo anterior, estas acciones quedan almacenadas en la cadena de bloques, utilizando algoritmos criptográficos seguros que evitan manipulaciones o accesos ilegítimos por terceras personas. En consecuencia, si el ordenador de un usuario donde se ejecuta la DApp quedara inservible, sería suficiente con volver a ejecutar en un nuevo dispositivo esta aplicación descentralizada y de manera completamente segura volveríamos a contar con toda la información aparentemente perdida debido al alto nivel de redundancia de datos en los demás participantes de la red. Evidentemente, esto será posible siempre y cuando existan más usuarios que usen dicha DApp.

Como hemos visto, una DApp funciona de manera similar a una red blockchain. Así **cada usuario de la DApp viene a ser un nodo dentro de la red**, velando por su adecuado funcionamiento y vigilando las operaciones que se ejecutan en la misma.

La DApp utiliza la cadena de bloques como conducto de comunicaciones dejando un registro de cada operación que se realiza en el contrato inteligente que controla la DApp. La programación del contrato inteligente determina que las operaciones de los

usuarios sean aceptadas o rechazadas garantizando así que todos los intervinientes actúen conforme a lo definido en el contrato.

El contrato inteligente actúa en este caso como **un punto intermedio encargado de ratificar la validez de cada interacción** resultando que en el momento en que se produce una nueva operación en la DApp, la información de la plataforma se actualiza en cada nodo. Así se asegura que la información quede guardada en cada uno de los nodos. En consecuencia, cada usuario contribuye a sustentar la DApp con los recursos de su computadora. Esta estructura también garantiza que la plataforma siempre estará operativa, pues resulta inviable que puedan anularse todos los nodos de la red al mismo tiempo, por ejemplo, en el caso de un ataque informático u otros supuestos como la censura.

Por ello podemos decir que las DApps aprovechan las capacidades de seguridad, privacidad y anonimato de la cadena de bloques sobre la que se ejecutan, asegurando además que los datos usados por la DApp sólo son accesibles por el usuario que originó dicha información. Con lo que los usuarios mantienen un control absoluto de sus datos en todo momento.

Aunque tienen muchas ventajas, también es conveniente destacar algunas de sus limitaciones más importantes:

- Dificultad de explotar el entero potencial del hardware de los dispositivos de los usuarios. Esta dificultad afecta principalmente a DApps que se ejecutan desde navegadores web, y ello por el gran número de capas de abstracción, así como por el alto nivel de ejecución de los lenguajes de programación utilizados.
- La complejidad de la aplicación muchas veces puede hacer difícil su depuración y revisión de seguridad, especialmente en la programación de los smart contracts ya que una vulnerabilidad en los mismos puede afectar a todos los usuarios de la DApp al mismo tiempo.
- El crecimiento y mejora de las DApps está íntimamente unido al incremento de las mejoras dentro de las cadenas de bloques.
- Dificultad para implementar funcionalidades necesarias para el correcto funcionamiento de las DApps. Algunas DApps son muy sencillas en sus requerimientos de programación, pero otras no tanto. [66]

Non-Fungible Tokens (NFT)

Los NFT están tomando por asalto el mundo del arte digital y los coleccionables. Los artistas digitales están viendo cómo sus vidas cambian gracias a las enormes ventas a una nueva audiencia criptográfica. Y las celebridades se están uniendo al descubrir una nueva oportunidad para conectarse con sus seguidores. Pero el arte digital es solo una forma de utilizar NFT. Realmente se pueden usar para representar la propiedad de cualquier activo único, como la escritura de un artículo en el ámbito digital o físico.



Los NFT son tokens que podemos usar para representar la propiedad de elementos únicos. Nos permiten tokenizar cosas como arte, objetos de colección e incluso bienes raíces. Solo pueden tener un propietario oficial a la vez y están protegidos por la cadena de bloques Ethereum: nadie puede modificar el registro de propiedad o copiar / pegar una nueva NFT para que exista.

NFT significa token no fungible. No fungible es un término económico que podría usar para describir cosas como sus muebles, un archivo de canciones o su computadora. Estos elementos no son intercambiables por otros elementos porque tienen propiedades únicas. [86]

Los artículos fungibles, por otro lado, se pueden intercambiar porque su valor los define en lugar de sus propiedades únicas. Por ejemplo, ETH o dólares son fungibles porque 1 ETH / \$ 1 USD es intercambiable por otro 1 ETH / \$ 1 USD.

Las NFT y Ethereum resuelven algunos de los problemas que existen en Internet en la actualidad. A medida que todo se vuelve más digital, es necesario replicar las propiedades de los elementos físicos como la escasez, la singularidad y la prueba de propiedad. Sin mencionar que los artículos digitales a menudo solo funcionan en el contexto de su producto. Por ejemplo, no puede revender un mp3 de iTunes que haya comprado o no puede cambiar los puntos de fidelidad de una empresa por el crédito de otra plataforma, incluso si hay un mercado para ello.

Así es como se ve una Internet de NFT en comparación con la Internet que la mayoría de nosotros usamos hoy en día.

Futuro de Internet con NFT	Internet de hoy
Los NFT son digitalmente únicos, no hay dos NFT iguales	Una copia de un archivo, como .mp3 o .jpg, es igual que el original.
Cada NFT debe tener un propietario y esto es de registro público y fácil de verificar para cualquier persona.	Los registros de propiedad de los artículos digitales se almacenan en servidores controlados por las instituciones; debe confiar en su palabra.
Los NFT son compatibles con todo lo creado	Las empresas con elementos digitales

con Ethereum. Una entrada NFT para un evento se puede intercambiar en cada mercado de Ethereum, por un NFT completamente diferente. Se podría cambiar, por ejemplo, una obra de arte por una entrada.	deben construir su propia infraestructura. Por ejemplo, una aplicación que emite entradas digitales para eventos tendría que crear su propio intercambio de entradas.
Los creadores de contenido pueden vender su trabajo en cualquier lugar y pueden acceder a un mercado global.	Los creadores confían en la infraestructura y distribución de las plataformas que utilizan. Suelen estar sujetos a condiciones de uso y restricciones geográficas.
Los creadores pueden conservar los derechos de propiedad sobre su propio trabajo y reclamar regalías de reventa directamente.	Las plataformas, como los servicios de transmisión de música, retienen la mayor parte de los beneficios de las ventas.
Los artículos se pueden utilizar de formas sorprendentes. Por ejemplo, puede utilizar obras de arte digitales como garantía o aval en un préstamo descentralizado.	

Las propiedades especiales de los NFT podemos resumirlas en los siguientes puntos:

- Cada token acuñado tiene un identificador único.
- No son directamente intercambiables con otros tokens con una proporción de uno a uno. Por ejemplo, 1 ETH es exactamente igual que otro ETH y en el caso de los NFT no es así. También cabe resaltar su indivisibilidad, dado que no se pueden dividir en partes más pequeñas.
- Cada token tiene un propietario y esta información es fácilmente verificable.
- Viven en Ethereum y se pueden comprar y vender en cualquier mercado NFT basado en Ethereum.

En otras palabras, si tenemos un NFT:

- Podemos demostrar fácilmente que lo poseemos.
- Nadie puede manipularlo de ninguna manera.
- Podemos venderlo y, en algunos casos, esto generará beneficios de reventa al creador original.
- O podemos guardarlo para siempre, descansando cómodamente sabiendo que nuestro activo está protegido por nuestra billetera en Ethereum.

Y si creamos un NFT:

- Podemos demostrar fácilmente que somos el creador.
- Podemos tener beneficios cada vez que se venda.
- Podemos venderlo en cualquier mercado NFT o peer-to-peer. No está encerrado en ninguna plataforma y no necesitamos a nadie para intermediar.
- Determinamos la escasez del mismo.

Este último punto es muy interesante, por ejemplo, consideremos una entrada para un evento deportivo. Así como un organizador del evento puede elegir cuántas entradas vender, el creador de una NFT puede decidir cuántas réplicas existen. A veces, estas son réplicas exactas, como 5000 entradas de admisión general. Pero a veces, se

acuñan varios que son muy similares, pero cada uno ligeramente diferente, como una entrada con asiento asignado. En otros casos, el creador puede querer crear un NFT en el que solo se acuñe uno como un coleccionable raro o especial. [75]

4. Diseño y Desarrollo del proyecto: Smart Contract Travel



4.1. Definición y objetivos

Se pretende crear un sistema que permita asegurar la integridad de todos los pasajeros de un vuelo en tiempos de pandemia controlando la autenticidad de los certificados emitidos por una autoridad sanitaria en relación a un test negativo en los 3 días previos al vuelo de cada pasajero. Cuando un usuario compre un billete de avión, la compañía/aerolínea deberá asegurarse de que todos los pasajeros tengan un certificado sanitario con resultado negativo en la prueba del covid19 emitido en las últimas 72 horas antes del despegue de su vuelo. Para garantizar la integridad de los pasajeros y certificados sanitarios se hará uso de la cadena de bloques de Ethereum mediante la implementación de un contrato inteligente entre la compañía/aerolínea y el centro sanitario como actores principales y el pasajero como actor secundario. Las acciones del pasajero simplemente se reducirían a comprar el billete y proporcionarle a la aerolínea los datos del centro sanitario donde se realizará la prueba, para que esta los corrobore y envíe posteriormente por correo electrónico la dirección del contrato.

Asimismo, se plantea la opción de modificar el contrato inteligente si el pasajero se dirige con síntomas a un centro sanitario con un certificado previo emitido por un centro sanitario emisor que le habilitara para el viaje en las últimas 72 horas.

Finalidad e idea principal

Asegurar la integridad de todos los pasajeros de un vuelo en tiempos de pandemia controlando la autenticidad de los certificados emitidos por una autoridad sanitaria que indican un test negativo en los 3 días previos al vuelo de cada pasajero haciendo uso de la cadena de bloques de Ethereum.

Cuando un usuario reserve y por tanto quede realizado el pago de un billete de avión, la compañía/aerolínea deberá asegurarse de que todos los pasajeros tengan un certificado sanitario con resultado negativo en la prueba del covid19 emitido en las últimas 72 horas antes del despegue de su vuelo. Para ello se hará uso de la Blockchain de Ethereum, asegurando mediante la implementación de un contrato inteligente la autenticidad de dicho certificado habilitador.

En nuestro supuesto existen 3 actores principales:

- Usuario o pasajero: Persona o grupo de personas que desean viajar con toda garantía sanitaria.

- Compañía/Aerolínea: Entidad que debe cerciorarse que todos los pasajeros cumplan con las normas sanitarias para efectuar un viaje determinado.
- Centro sanitario: Entidad emisora del certificado sanitario que todos los pasajeros deben validar con la compañía en las últimas 72 horas antes del vuelo.

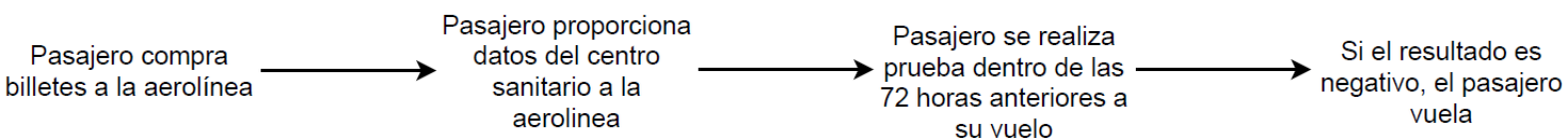
La idea sería que en esta validación no esté involucrado el interesado (pasajero) para facilitar este trámite, por lo que reduciríamos el número de actores del smart contract a 2, la compañía/aerolínea (emisora del contrato) y el centro sanitario (validador del contrato). Las acciones del pasajero simplemente se reducirían a comprar el billete y proporcionar a la aerolínea la información necesaria del centro sanitario de su elección para la realización de la prueba correspondiente.

Funcionalidad extra: capacidad del contrato para ser modificado si el pasajero se dirige con síntomas a un centro sanitario cuando previamente éste ya le emitió el certificado habilitador para el viaje en las últimas 72 horas.

Actores/usuarios y acciones

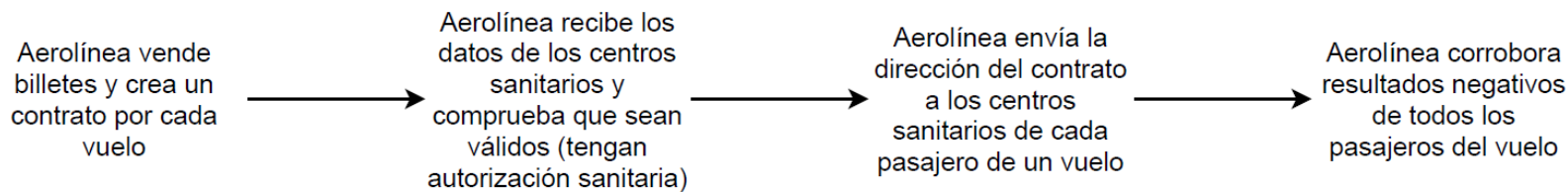
Como hemos comentado, los actores implicados en el funcionamiento de este contrato inteligente serían, con un papel principal dentro de la aplicación, la aerolínea y el centro sanitario del viajero y con un carácter secundario el propio pasajero. Pasamos a describir las acciones que deberán realizar los mismos para un correcto funcionamiento de la aplicación:

- El pasajero: El pasajero comprará los billetes de su vuelo correspondiente a la aerolínea, introduciendo sus datos personales y los datos identificativos del centro sanitario donde se realizará la prueba del Covid19 en las 72 horas anteriores a la salida de su vuelo, como son el NIF, el nombre del centro sanitario y su correo electrónico, así como la dirección de la cuenta en Ethereum que deberá poseer el centro sanitario, para poder relacionar así la cuenta en la blockchain de Ethereum con el NIF y los datos del centro sanitario correspondiente.



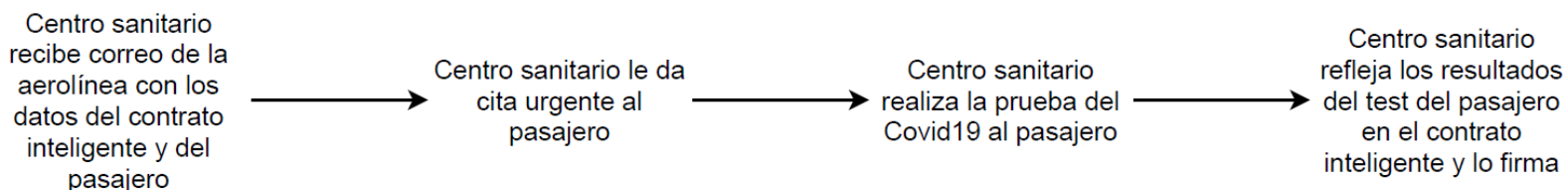
- La aerolínea: La aerolínea será la encargada de vender los billetes de avión a los pasajeros correspondientes, y de recibir y almacenar los datos facilitados por el mismo, tanto personales como del centro sanitario donde este se realizará la prueba que deberá tener un resultado negativo en los 3 días anteriores al vuelo correspondiente. La aerolínea deberá implementar y crear el contrato inteligente en la red de Ethereum, para lo cual deberá poseer una cuenta para desplegarlos, y enviar los datos del mismo al centro sanitario facilitado por el pasajero justamente 72 horas antes de la salida de su vuelo, para así asegurar la temporalidad del test de Covid19. También deberá corroborar que los datos proporcionados por el pasajero acerca del centro sanitario son correctos y de que este tiene un registro sanitario que le habilite para realizar dichas pruebas, para así evitar que cualquier persona con cuenta en la blockchain de Ethereum a la que se le facilite el contrato

no pueda hacerse pasar por un centro sanitario que dispense pruebas negativas falseadas.



- El centro sanitario: El centro sanitario será el encargado de la realización del test del coronavirus al pasajero. Para ello, una vez que el centro sanitario recibe el correo con los datos del contrato y pasajero por parte de la aerolínea tres días antes del vuelo, deberá darle una cita “urgente” al mismo siempre y cuando este todavía no la haya concertado con ellos, es decir, si el pasajero concreta la cita con su centro sanitario para por ejemplo 4 días antes del vuelo, el centro sanitario todavía no tendrá los datos necesarios para reflejar los resultados del test, por lo que no le servirá de nada, ya que esta cita deberá ser siempre en las 72 horas anteriores a que el avión despegue. Una vez que el centro sanitario le realice la prueba al pasajero, deberá seleccionar la función correspondiente dentro del contrato inteligente facilitado por la aerolínea para registrar el resultado del test, rellenando los campos requeridos en el mismo, como son, su NIF, su nombre, nombre del pasajero, DNI del pasajero, identificador de la prueba realizada y el resultado de la misma. Y posteriormente introducirá su número de cuenta de la blockchain de Ethereum, que deberá coincidir con el número que firmará la transacción, por lo que si un centro sanitario A quiere registrar los datos como si fuese un centro sanitario B, a la hora de revisar los centros sanitarios facilitados por cada pasajero, se observaría que el centro sanitario B ha registrado el resultado del pasajero en nombre del centro sanitario A, por lo que se tomarían las medidas oportunas por parte de la aerolínea.

Aunque sea evidente, cabe destacar que los datos personales del pasajero rellenados en el smart contract por el centro sanitario deberán coincidir con los datos del pasajero en el billete de avión adquirido por el mismo, asegurando así que no se utiliza la prueba con resultado negativo de otra persona diferente al pasajero, dado que en el centro sanitario se deberá acreditar debidamente la identidad de la persona que va a realizarse el test.



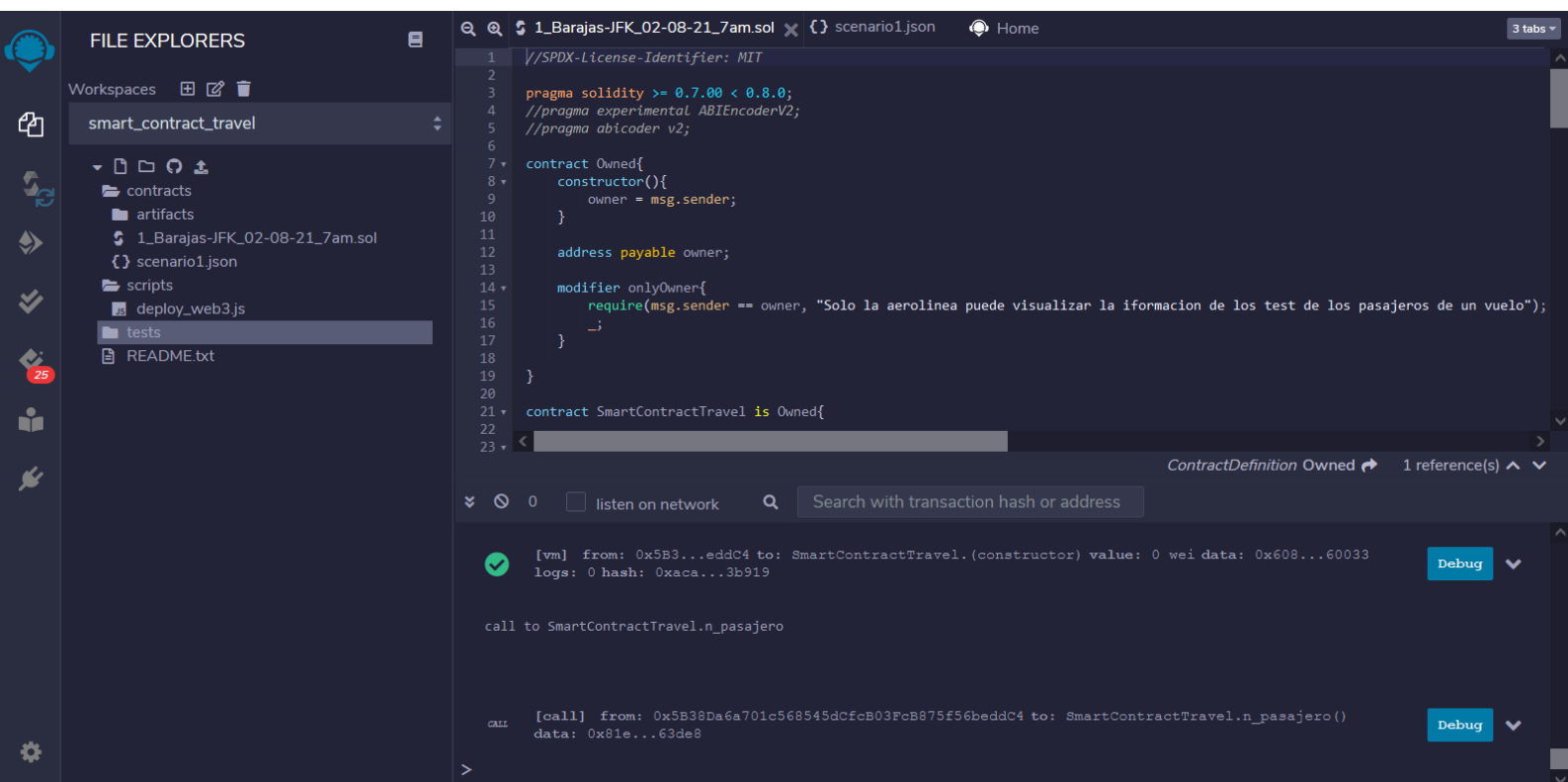
4.2. Implementación

Entorno de desarrollo: Remix

Dado que ya sabemos lo que queremos desarrollar y sabemos que la mejor manera para comenzar es a través del IDE online Remix iremos viendo paso a paso como se usa el mismo y cómo hemos desarrollado el Smart Contract Travel con el lenguaje de programación Solidity.

Una de las mejores bazas que tiene remix es que sin necesidad de descargar absolutamente ningún archivo y desde el propio navegador incorpora la opción de programar, de compilar, de subir el código e interactuar con cualquier blockchain y nos deja elegir distintos tipos de compilación lo cual es bastante cómodo.

Lo primero de todo, nos dirigimos a nuestro navegador favorito y buscamos <https://remix.ethereum.org>:



Analizamos un poco la interfaz y vemos que se parece mucho a un IDE offline cualquiera con su Home de bienvenida con accesos directos a múltiples opciones, el bloque de entrada de código, así como su propia terminal donde podremos visualizar los detalles de las transacciones, depurar los contratos y ejecutar scripts de Java Script.

También vemos el explorador de archivos, en donde tenemos un espacio de trabajo por defecto con una estructura definida en tres directorios ya creados y según en la versión que estemos trabajando también contendrán algunos contratos de ejemplo, scripts para desplegar contratos en la blockchain y algún test de ejemplo que borramos antes de empezar.

En las siguientes imágenes de nuestro entorno real del proyecto que nos ocupa, vemos un gran menú con diversas opciones, de las cuales hemos usado mayoritariamente para el desarrollo del SmartContractTravel las tres primeras:

- El explorador de archivos, visto anteriormente.
- El compilador: En esta parte llamada *Solidity compiler* es donde se compila el código comprobando que lo hace sin errores e indicando la versión del compilador, en este caso, para evitar incompatibilidades hemos usado la 0.7.6, ya que nos vale cualquiera entre la 7 y la 8 por como veremos posteriormente que se ha indicado en el código.

Una vez compilado el código nos da las opciones de publicarlo directamente en un servidor descentralizado y ver los detalles de compilación, desde el nombre del contrato, el ABI y el código máquina generado (bytecode), hasta el código ensamblador, estimaciones del coste en gas del contrato, metadatos, y más información.

Cabe mencionar que el ABI de la aplicación es el encargado de hacer la comunicación de nuestro frontend en nuestra web contra el ByteCode que habremos desplegado en la blockchain, es por decirlo de alguna manera un traductor, además nosotros mismo podremos leer sin problema la ABI para hacernos una idea de que es lo que contiene nuestro contrato y que es lo que podremos hacer con él.

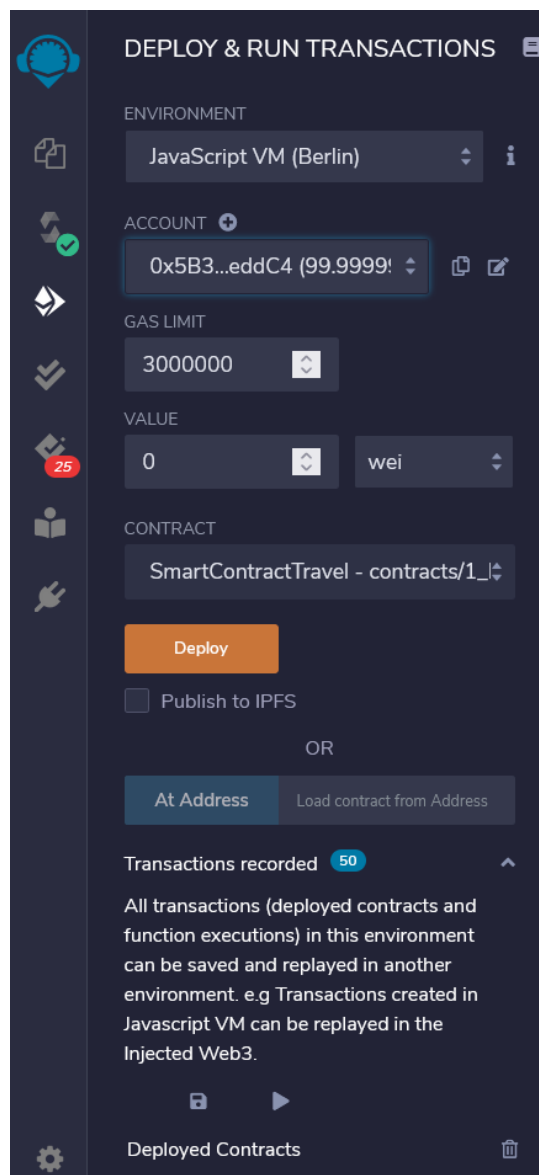
The image shows a web-based Solidity compiler interface. On the left, a sidebar contains icons for file explorer, compiler, and other tools. The main panel is titled 'SOLIDITY COMPILER' and is divided into several sections:

- COMPILER:** Shows the version '0.7.6+commit.7338295f' and a checkbox for 'Include nightly builds'.
- LANGUAGE:** Set to 'Solidity'.
- EVM VERSION:** Set to 'compiler default'.
- COMPILER CONFIGURATION:** Includes checkboxes for 'Auto compile', 'Enable optimization' (set to 200), and 'Hide warnings'.
- CONTRACT:** Shows the contract name 'Owned (1_Barajas-JFK_02-08-21_7am.sol)' and buttons for 'Publish on Swarm', 'Publish on Ipfs', and 'Compilation Details'.

At the bottom of the sidebar, there are links for 'ABI' and 'Bytecode'. The right panel displays the compilation results for the contract 'SmartContractTravel':

- METADATA:** Shows compiler, language, output, settings, sources, and version information.
- BYTECODE:** Displays the generated bytecode object.
- ABI:** Shows the contract's ABI, including inputs, outputs, and state mutability.

- Despliegue del contrato: Desde esta opción *Deploy & run transactions* podremos desplegar y ejecutar nuestro contrato para comprobar su funcionamiento. En nuestro caso lo desplegaremos en una máquina virtual de JS que emula el entorno de la blockchain de Ethereum y donde podremos comprobar el funcionamiento de nuestro contrato innumerables veces sin coste alguno, dado que se nos proporcionan también una serie de cuentas con 100 ETH cada una para estos fines de prueba; y también lo desplegaremos en el entorno de *Injected Web3*, conectando nuestra cartera de Ethereum de MetaMask y comprobaremos el funcionamiento en una verdadera testnet, dado que no tenemos fondos para desplegarla en la red real de Ethereum, todo esto lo veremos en los puntos posteriores.



- Por último, tenemos cuatro opciones menos usadas que las anteriores, pero igualmente importantes, como es la parte de testing para realización de test unitarios, un linter de Solidity añadido en estas últimas versiones, los enlaces a la documentación y workshops para el desarrollo de aplicaciones descentralizadas y por último un gestor de extensiones de Remix.

4.2.1. Desarrollo y análisis del código

Vamos a analizar parte por parte el código desarrollado para el correcto funcionamiento del Smart Contract Travel, que como ya sabemos tiene como objetivo asegurar la veracidad de la prueba del covid19 de todos los pasajeros de un vuelo de una compañía específica, por lo que la compañía sería la encargada de desplegar un Smart Contract Travel por cada vuelo que oferten.

I. Licencia:

Como en todo contrato debemos comenzar definiendo en un comentario de línea el tipo de licencia que posee, lo que significa a grandes rasgos que si es una licencia privada no se podrá modificar ni copiar y si es una licencia de código abierto se podrá tanto modificar como copiar. Podemos encontrar una lista de licencias en el siguiente enlace <https://spdx.org/licenses/preview/> . Por defecto, en Ethereum prácticamente todo es *open source*, lo que quiere decir que todo el mundo podría modificarlo y copiarlo, lo que hace que sea un entorno mucho más competitivo y en continua evolución.

```
//SPDX-License-Identifier: GPL-3.0
```

II. Versión pragma del código:

Como hemos visto anteriormente, en Solidity es necesario comenzar indicando la versión pragma del código fuente para evitar la compilación con versiones futuras del compilador que podrían introducir cambios e incompatibilidades. Por tanto, con la siguiente regla estamos habilitados para compilar este código con cualquier compilador de Solidity en su versión 0.7, la última estable. Cabe decir que entre versiones no hay cambios que rompan el código, por lo que podremos trabajar con la 0.7.1, 0.7.2, etc...

```
pragma solidity >= 0.7.00 < 0.8.0;
```

III. Contratos:

Nuestra aplicación está conformada por dos contratos bien definidos, Owned y Smart Contract Travel aunque la lógica de la misma reside sobretudo en el segundo de ellos, veámoslos primero por separado.

```
contract Owned{
    constructor(){
        owner = msg.sender;
    }
    address payable owner;

    modifier onlyOwner{
        require(msg.sender == owner, "Solo la aerolinea puede realizar esta funcion");
        _;
    }
}
```

Este contrato únicamente define un modificador, aunque se va a usar en el contrato heredado llamado Smart Contract Travel. Es habitual encontrarse en la mayoría de contratos estos modificadores, y se usan para cambiar el funcionamiento de funciones de forma declarativa. En este caso usamos un modificador para comprobar la condición de que las funciones con este modificador sólo puedan ser ejecutadas por la dirección creadora del contrato, es decir, la aerolínea.

Este contrato consta de un constructor que inicializa la variable *owner* a la dirección que crea el contrato, es decir, a la dirección de la aerolínea. Posteriormente vemos el modificador llamado *onlyOwner*, donde comprobamos gracias a un *require* que la dirección que llama a la función es la misma que creó el contrato y en caso contrario se lanzará la excepción con el mensaje “Sólo la aerolínea puede realizar esta función”. El cuerpo de la función con este modificador será insertado donde aparecen los caracteres especiales “_” en la definición de *onlyOwner*.

Sabiendo cómo funciona el contrato Owned su modificador, pasamos a analizar por partes el código del contrato del Smart Contract Travel para entender el posterior funcionamiento de la aplicación:

```
contract SmartContractTravel is Owned{

    struct Pasajero{
        string nombre;
        string apellidos;
        string dni;

        string nif_csantario;
        string nombre_csantario;
        address dir_csantario;

        string resultado_test;

        uint id_npasajero;
    }

    uint public n_pasajero = 0;

    mapping(uint=>Pasajero) private Pasajeros;

    ...
}
```

Lo primero que vemos es que nuestro SmartContractTravel hereda del contrato Owned que como ya hemos comentado, será en funciones de nuestro contrato donde se usará su modificador.

En la zona de declaración de variables de nuestro contrato tenemos:

- *struct Pasajero*: Usado para almacenar los datos referentes a cada pasajero del vuelo de nuestro contrato, estos son: su nombre, apellidos y dni del pasajero; el nif del centro sanitario, su nombre y la dirección de su cuenta que usará para firmar las transacciones, así como, el propio resultado del

test del pasajero y un identificador de pasajero usado como resguardo de que un pasajero se ha realizado la prueba, que será único por cada pasajero de un vuelo y que podrá ser facilitado al pasajero por el centro sanitario, después veremos cómo lo usamos.

Nota: En Solidity no es posible para un *struct* contener un miembro de su propio tipo, aunque el *struct* puede ser el tipo “valor” de un miembro *mapping* (como veremos posteriormente), esta restricción es necesaria ya que el tamaño del *struct* tiene que ser finito, por como sabemos que funciona la blockchain de Ethereum y su lenguaje.

- *uint public n_pasajero*: Número entero sin signo que actuará como contador de pasajeros que se han realizado la prueba del covid19 en un centro sanitario del vuelo del contrato. Al declararlo como *public* se crea su método *get* automáticamente.

Nota: También cabe destacar que, aunque aquí nosotros inicialicemos la variable a 0, en Solidity todas las variables son autoinicializadas con el valor por defecto de ese tipo de variable, por lo que nunca nos encontraremos una variable nula o indefinida.

- *mapping(uint=>Pasajero) private Pasajeros*: Un mapa en Solidity es muy parecido a los diccionarios en otros lenguajes donde el primer parámetro es la clave y el segundo es el valor de esa clave. En nuestro caso, a través de un número entero sin signo correspondiente al valor de la posición en la que están guardados los datos en el mapa, obtenemos los datos guardados en la estructura de cada pasajero. Aquí se almacenan todos los pasajeros del vuelo correspondiente al contrato que hayan pasado por el centro sanitario para la realización de la prueba del covid19 y hemos decidido definirlo como *private* dado que no queremos que esos datos sean accesibles ni modificables por otros contratos.

IV. Funciones:

Nuestro contrato cuenta con cinco funciones, y vamos a analizar cada una por separado comenzando por la primera de ellas.

```
function Registrar_datos_pasajero(string calldata _nombre, string calldata _apellidos, string calldata _dni, string calldata _nif_csantario, string calldata _nombre_csantario, address _dir_csantario, string calldata _resultado_test) public{

    require(msg.sender == _dir_csantario, "La direccion introducida no coincide con el centro sanitario que envia la transaccion");

    n_pasajero++;

    Register(_nombre, _apellidos, _dni, _nif_csantario, _nombre_csantario, _dir_csantario, _resultado_test, n_pasajero);
}
```

Esta primera función pública llamada *Registrar_datos_pasajero* será el primer paso para registrar los datos de los pasajeros en la blockchain por parte del centro

sanitario, incluyendo el resultado de la prueba realizada al pasajero y recibe una serie de parámetros como el nombre, apellidos, dni del pasajero, resultado del test y los datos referentes al centro sanitario de la estructura del pasajero comentados anteriormente. Será la encargada de llamar a la siguiente función llamada *Register* pasándole estos datos por parámetros más el valor incrementado del número de pasajeros que se han realizado la prueba de ese vuelo (variable *n_pasajero*). También se comprueba con un *require* que la dirección que llama a esa función coincida con la introducida por el centro sanitario que se corresponda, para posteriormente la aerolínea pueda comprobar con los datos que posee de dicho vuelo que se trata de un centro sanitario válido y habilitado para este fin.

Nota: Cabe destacar que cada tipo de datos complejo en Solidity (vectores, cadenas de texto, mapas, estructuras...) tienen anotaciones adicionales como vemos en los parámetros de las funciones. Estas hacen referencia a la ubicación de los datos del contrato, dependiendo del contexto, siempre hay un valor por defecto, pero puede ser reemplazado añadiendo o bien *storage* o *memory* al tipo de dato. Por ejemplo, para tipos parámetros de función (incluyendo parámetros de retorno) es *memory*, por defecto para variables locales es *storage* y la ubicación es forzosamente *storage* para variables de estado. Existe una tercera ubicación de datos llamada *calldata*, se trata de un área que no es modificable ni persistente donde argumentos de funciones son almacenados, entonces los parámetros (no los de retorno) de funciones llamadas externamente son forzosamente “*calldata*” y se comportan casi como si fuesen *memory*.

```
function Register(string calldata _nombre, string calldata _apellidos, string calldata _dni,
    string calldata _nif_csantitario, string calldata _nombre_csantitario,
    address _dir_csantitario, string calldata _resultado_test, uint _i) internal {

    require(msg.sender == _dir_csantitario, "La direccion introducida no coincide con el
centro sanitario que envia la transaccion");

    Pasajeros[_i] = Pasajero({
        nombre: _nombre,
        apellidos: _apellidos,
        dni: _dni,

        nif_csantitario: _nif_csantitario,
        nombre_csantitario: _nombre_csantitario,
        dir_csantitario: _dir_csantitario,

        resultado_test: _resultado_test,
        id_npasajero: _i
    });
}
```

Esta función llamada *Register* sólo podrá ser llamada desde el propio contrato debido a que es una función interna del mismo, por lo que el usuario no podrá interactuar con ella directamente. Como hemos visto anteriormente recibe los mismos parámetros que la función *Registrar_datos_pasajero* más el identificador de la posición del pasajero para asociarla a dicho pasajero. En definitiva, esta función almacena los datos personales del pasajero (en la estructura) en una posición determinada del conjunto de pasajeros (mapa).

```
function Registrar_cambio_test(uint _id_npasajero, address _dir_csanitario) public{

    require(msg.sender == _dir_csanitario, "La direccion introducida no coincide con el
centro sanitario que envia la transaccion"); //requiere que el que registra los datos del pa
sajero sea el centro sanitario que llama al contrato

    Pasajeros[_id_npasajero].resultado_test = "POSITIVO";

}
```

Esta función está concebida para que en el caso de que un pasajero se contagie o realice otro test con resultado positivo en las 72 horas antes de un vuelo después de un primer resultado negativo, su centro sanitario pueda modificar los datos del primer test a un test positivo. Es por ello que esta función recibe como parámetro el número identificador del pasajero (donde se encuentran almacenados sus datos dentro del mapa de pasajeros de un vuelo) que se le proporcionó por parte del centro sanitario en la realización de su primer test y la dirección de dicho centro sanitario. Así, con esta función *Registrar_cambio_test* el centro sanitario podrá modificar los datos del pasajero en cuestión en caso de que no pudiese volar finalmente.

```
function Datos_Pasajero_Registrado(uint _id) public onlyOwner view returns (string memory no
mbre, string memory, string memory, string memory, string memory, address dir_csanitario, st
ring memory, uint id_npasajero){

    Pasajero memory p = Pasajeros[_id];

    return (p.nombre, p.apellidos, p.dni, p.nif_csanitario, p.nombre_csanitario, p.dir_csa
nitario, p.resultado_test, p.id_npasajero);

}
```

El nombre de esta función, *Datos_Pasajero_Registrado*, también nos da una pista de cuál es su finalidad, recuperar los datos de los pasajeros cuyos datos han sido registrados por los centros sanitarios de cada uno. Y justamente para esta función y la siguiente se ha hecho que el contrato Smart Contract Travel herede de *Owned*, dado que es aquí donde queremos que los datos de todos los pasajeros de un vuelo solo puedan ser accesibles por la aerolínea (dirección creadora del contrato) para lo cual usamos el modificador de la función *onlyOwner*. También la declaramos como una vista *view*, dado que esta función es únicamente de lectura y no escribe ningún dato, solo queremos devolverlos.

Nota: Esto último es importante dado que como ya sabemos, modificar datos requiere de minar un bloque, por lo que conlleva un gasto, en cambio, como consultar no requiere de minado, es gratis e instantáneo.

Nota2: Dado que el compilador de Solidity lanza un error si se intenta devolver un dato de tipo estructura (o array) se pensó en utilizar un mapping público que generase su propio getter de la estructura de datos de los pasajeros del vuelo (lo que internamente funciona similar al código de la función anterior, “desenrollando” el struct en tuplas), pero como se quería que únicamente fuese llamado por la aerolínea y por nadie más, pasándole como parámetro el identificador del pasajero en cuestión, este método del mapa público no satisfacía nuestras necesidades.

Por último, tenemos la función encargada de llamar a la función *selfdestruct* con el objetivo de eliminar el contrato cuando el vuelo ya haya sido realizado, con esto se consigue destruir el contrato siendo imposible volver a acceder a cualquiera de sus funciones nunca más. Esta función requiere de una dirección como parámetro, a donde se le reembolsarán los fondos restantes del contrato, por lo que esta función es la de la propia aerolínea creadora del contrato y sólo podrá ser llamada por su dirección, ya que está protegida con el modificador *onlyOwner*.

```
...  
function destroy() public onlyOwner{  
    selfdestruct(owner);  
}  
}
```

Una vez comprendido el código de nuestros contratos y cómo interactúan entre ellos vamos a desplegarlos con el propio IDE de Ethereum que ya hemos visto para comprobar su correcto funcionamiento. Lo haremos de dos maneras, con la máquina virtual de Java Script que nos provee Remix y con *Injected Web3* para comprobar con *Metamask* su funcionamiento real en una red de pruebas de Ethereum.

Para comenzar, nos situamos en la pestaña del menú de Remix en la opción del compilador y con las opciones vistas en apartados anteriores pulsamos en compilar el contrato.

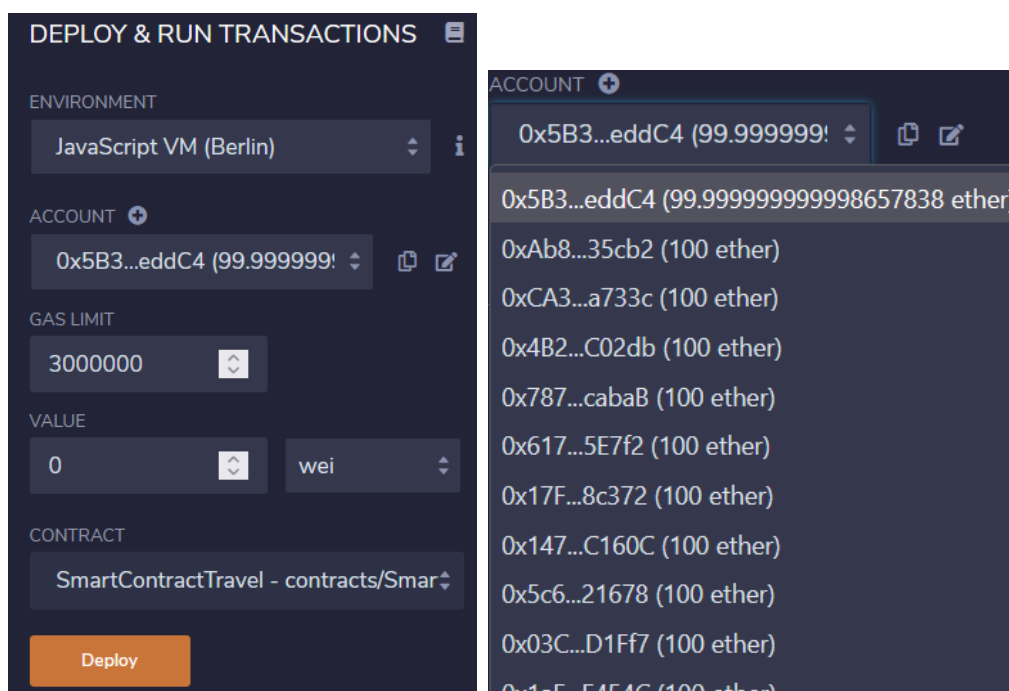


Una vez que lo hemos compilado nos dirigimos a la opción del menú de Remix desde donde podremos elegir el entorno de despliegue de nuestro contrato.

4.2.2. Despliegue con Remix I: JavaScript VM

Esta es la opción más sencilla para comprobar el funcionamiento de nuestro contrato dado que con la *JavaScriptVM* podremos compilar en memoria y desplegar únicamente en memoria, por lo que podríamos entenderlo como una especie de despliegue en un entorno simulado, en donde sólo nosotros podremos probar nuestras funciones de una manera rápida, sencilla y eficaz.

Antes de nada, nos fijamos en la cantidad de Ethereum (virtual) que poseen las cuentas que nos proporciona esta máquina virtual, en este caso, todas tienen 100 ETH excepto la primera que la consideraremos la cuenta de la aerolínea, encargada de desplegar el contrato. Por lo que, la seleccionamos y elegimos el contrato llamado *SmartContractTravel_Madrid-NewYork_idvuelo*, el cual deberá ser desplegado uno por cada vuelo. En este entorno, el límite de gas y el valor del mismo no nos importa demasiado dado que es un entorno de pruebas y no consumirá ningún valor real. Por lo tanto, pulsamos en el botón *Deploy* con estas opciones como si fuéramos la aerolínea para desplegar el contrato de este vuelo.



Una vez desplegado, nos aparecerá en la parte de abajo de este mismo menú nuestro contrato, desde donde podremos interactuar con las funciones del mismo. Y también vemos en el terminal los detalles de las acciones que se están realizando. En este caso, el despliegue del contrato es casi instantáneo debido a que estamos en un entorno simulado y nadie debe supervisar las transacciones como en un entorno real.



Una vez se ha desplegado el contrato, nos fijamos que la dirección de la aerolínea, `0x5B38Da6a701c568545dCfcB03FcB875f56beddC4`, tiene menos ETH por el coste de esta operación.

Antes de desplegar las opciones de nuestro contrato, y encontrarnos con el siguiente menú, vamos a cambiar la dirección en la parte de arriba, a la segunda, `0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2`, con 100 ETH, que será la dirección de un centro sanitario válido llamado "Centro Sanitario 1", por ejemplo.



Aquí nos encontramos con las posibilidades que brinda nuestro contrato para interactuar con él, vamos a repasarlas rápidamente y a comprobar su funcionamiento:

- *Registrar_datos_pasajero*: Función que registrará los datos de cada pasajero de un vuelo y el resultado de su test de Covid19 por parte de su centro sanitario.

The screenshot shows a form titled 'Registrar_datos_pasajero'. It contains several input fields with labels and values: '_nombre:' with value 'Juan', '_apellidos:' with value 'Garcia Perez', '_dni:' with value '11111111X', '_nif_sanitario:' with value '11111111', '_nombre_sanitario:' with value 'Centro1', '_dir_sanitario:' with value '33F64d9C6d1EcF9b849Ae677dD3315835cb2', and '_resultado_test:' with value 'NEGATIVO'. At the bottom right, there is a blue 'transact' button and a copy icon.

En el momento que pulsamos en *Transact* se ejecutará dicha función con los parámetros elegidos, donde deberá coincidir la dirección introducida por el centro sanitario con la dirección que llama a la función, es decir, la misma que la del “Centro Sanitario 1” que hemos comentado anteriormente.

```
transact to SmartContractTravel.Registrar_datos_pasajero pending ...

[vm] from: 0xAb8...35cb2
to:
SmartContractTravel.Registrar_datos_pasajero(string,string,string,string,string,address,
ress,string) 0xD7A...F771B
value: 0 wei data: 0x6ac...00000 logs: 0 hash: 0x49b...cd38a
```

Arrojándonos este resultado positivo y donde podemos observar los detalles de que la transacción se ha efectuado correctamente. En caso de que ambas direcciones no coincidiesen, se revertiría la transacción y se mostraría el siguiente mensaje:

```
transact to SmartContractTravel.Registrar_datos_pasajero pending ...

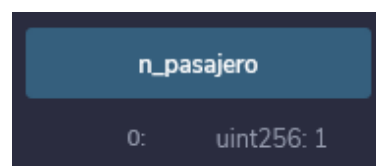
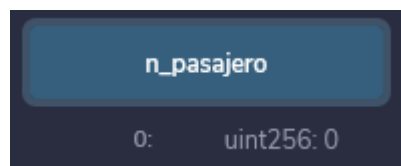
[vm] from: 0x5B3...eddC4
to: SmartContractTravel.Registrar_datos_pasajero(string,string,string,string,string,address,string) 0xD7A...F771B
value: 0 wei data: 0x6ac...00000 logs: 0 hash: 0x26f...d70cf

transact to SmartContractTravel.Registrar_datos_pasajero errored: VM error: revert.

revert
The transaction has been reverted to the initial state.
Reason provided by the contract: "La direccion introducida no coincide con el centro sanitario que envia la transaccion".
Debug the transaction to get more information.
```

- *n_pasajero*: En esta variable pública se guardaba un contador de pasajeros que han sido registrados por su centro sanitario en este vuelo, la cual justo antes del punto anterior tenía el valor 0 (por defecto, ya que no había ningún pasajero registrado), y al registrar los datos del primer pasajero ha cambiado a 1. Esta variable la puede consultar sin coste alguno cualquiera con acceso al contrato (dado que es un getter de una variable pública generado por Solidity automáticamente) e irá incrementando a medida que se vayan registrando los pasajeros en los tres días previos al mismo.

Deberá ser proporcionada al pasajero por parte de su centro sanitario a la realización de la primera prueba a modo de identificador de su prueba, por si posteriormente el mismo u otro centro sanitario necesita modificar el resultado de esta de “negativo” a “positivo”, ya que el pasajero podría contraer la enfermedad después de la realización de la primera prueba e ir al centro sanitario para la realización de una segunda.



```
call to SmartContractTravel.n_pasajero
```

```
CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  
to: SmartContractTravel.n_pasajero() data: 0x81e...63de8
```

Debug

- **Datos_pasajero_registrado:** Con esta función la aerolínea recuperará para la posterior comprobación antes de la salida del vuelo los datos de cada pasajero del mismo, introduciendo su identificador (cuyo máximo será el número total de pasajeros de ese vuelo).

En nuestro caso particular, actuando como la aerolínea, recordemos, desde la dirección “0x5B38...”, introducimos el primero de una larga lista de pasajeros un día de comprobación.

Nota: Esto se podría programar posteriormente desde el frontend para poder ver de un vistazo los datos de todos los pasajeros de un vuelo sin necesidad de ir uno por uno.

Datos_Pasajero_Registrado

_id: 1

call

0: string: nombre Juan

1: string: Garcia Perez

2: string: 111111111X

3: string: 111111111

4: string: Centro1

5: address: dir_csantario 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2

6: string: NEGATIVO

7: uint256: id_npasajero 1

Aquí podemos ver los detalles de esta transacción más en profundidad:

```
call to SmartContractTravel.Datos_Pasajero_Registrado
```

```
CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: SmartContractTravel.Datos_Pasajero_Registrado(uint256) data: 0xc8d...00001
```

Debug

transaction hash	0x27aa22a6173afb23bbe2fa9272816164d5cec9842688280752d676fc68026411
from	0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
to	SmartContractTravel.Datos_Pasajero_Registrado(uint256) 0xD7ACd2a9FD159E69Bb102Alca21C9a3e3A5F771B
execution cost	46156 gas (Cost only applies when called by a contract)
hash	0x27aa22a6173afb23bbe2fa9272816164d5cec9842688280752d676fc68026411
input	0xc8d...00001
decoded input	{ "uint256 _id": "1" }
decoded output	{ "0": "string: nombre Juan", "1": "string: Garcia Perez", "2": "string: 111111111X", "3": "string: 111111111", "4": "string: Centro1", "5": "address: dir_csantario 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2", "6": "string: NEGATIVO", "7": "uint256: id_npasajero 1" }

En el caso de que el que la dirección “*from*” no coincidiese con la dirección que llama a esta función, por ejemplo, si la llamamos desde la dirección del “Centro Sanitario 1”, los datos no serían retornados y se abortaría la transacción con el error:

```
CALL [call] from: 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2 to: SmartContractTravel.Datos_Pasajero_Registrado(uint256) data: 0xc8d...00001 Debug ▼

call to SmartContractTravel.Datos_Pasajero_Registrado errored: VM error: revert.

revert
    The transaction has been reverted to the initial state.
Reason provided by the contract: "Solo la aerolinea puede realizar esta funcion".
Debug the transaction to get more information.
```

- **Registrar_cambio_test:** Esta función sólo puede ser llamada por un centro sanitario para cambiar los datos de un pasajero a través de su identificador visto anteriormente. Por lo que deberemos introducirlo junto con la dirección del centro sanitario actual para cambiar el resultado de la prueba anterior, es decir, siempre de una prueba negativa a una prueba positiva.

Comprobamos su funcionamiento, como si del “Centro sanitario 1” y del pasajero 1 se tratase, para así posteriormente poder corroborar los cambios llamando a la función *Datos_Pasajero_Registrado* en el lugar de la aerolínea.

Registrar_cambio_test

_id_npasajero:

1

_dir_csantitario:

0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2

transact

Registrar_dato...

"Juan","Garcia Perez","11" ▼

Datos_Pasajer...

"1" ▼

0: string: nombre Juan

1: string: Garcia Perez

2: string: 111111111X

3: string: 111111111

4: string: Centro1

5: address: dir_csantitario 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2

6: string: NEGATIVO

7: uint256: id_npasajero 1

Datos_Pasajer...

"1" ▼

0: string: nombre Juan

1: string: Garcia Perez

2: string: 111111111X

3: string: 111111111

4: string: Centro1

5: address: dir_csantitario 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2

6: string: POSITIVO

7: uint256: id_npasajero 1

```
transact to SmartContractTravel.Registrar_cambio_test pending ...

[vm] from: 0xAb8...35cb2
to: SmartContractTravel.Registrar_cambio_test(uint256,address) 0xD7A...F771B value: 0 wei
data: 0x204...35cb2 logs: 0 hash: 0x4de...63abf
```

Si, por ejemplo, intentásemos llamarla desde una dirección diferente a la introducida por parámetros obtendríamos el siguiente error “La dirección introducida no coincide con el centro sanitario que envía la transacción”:

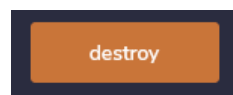
```
transact to SmartContractTravel.Registrar_cambio_test pending ...

[vm] from: 0x5B3...eddC4
to: SmartContractTravel.Registrar_cambio_test(uint256,address) 0xD7A...F771B value: 0 wei
data: 0x204...35cb2 logs: 0 hash: 0x511...53913

transact to SmartContractTravel.Registrar_cambio_test errored: VM error: revert.

revert
The transaction has been reverted to the initial state.
Reason provided by the contract: "La direccion introducida no coincide con el centro sanitario que envia la transaccion".
Debug the transaction to get more information.
```

- **Destroy:** Como ya sabemos, esta función sólo puede ser llamada por la creadora del contrato, por lo que, con la cuenta de la aerolínea, en el momento que pulsemos el botón los datos serán destruidos y las funciones de consulta del contrato dejarán de ser accesibles, aunque la dirección del contrato se mantenga.



```
transact to SmartContractTravel.destroy pending ...

[vm] from: 0x5B3...eddC4 to: SmartContractTravel.destroy() 0xD7A...F771B value: 0 wei
data: 0x831...97ef0 logs: 0 hash: 0x158...7153e

call to SmartContractTravel.Datos_Pasajero_Registrado
```

Si posteriormente llamamos a, por ejemplo, la función *Datos_Pasajero_Registrado* comprobamos que no nos devuelve nada.

```
call to SmartContractTravel.Datos_Pasajero_Registrado

CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
to: SmartContractTravel.Datos_Pasajero_Registrado(uint256) data: 0xc8d...00001

transaction hash      0x68c3b8d0e07089b7a3c94a099d9371c462680bca69170de561b4e0c271ae854b
from                  0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
to                    SmartContractTravel.Datos_Pasajero_Registrado(uint256)
                     0xD7ACd2a9FD159E69Bb102A1ca21C9a3e3A5F771B
execution cost        0 gas (Cost only applies when called by a contract)
hash                  0x68c3b8d0e07089b7a3c94a099d9371c462680bca69170de561b4e0c271ae854b
input                  0xc8d...00001
decoded input          { "uint256 _id": "1" }
decoded output          { "0": "string: nombre ", "1": "string: ", "2": "string: ", "3": "string: ",
                        "4": "string: ", "5": "address: dir_csantitario
                        0x00000000000000000000000000000000", "6": "string: ", "7": "uint256:"
```

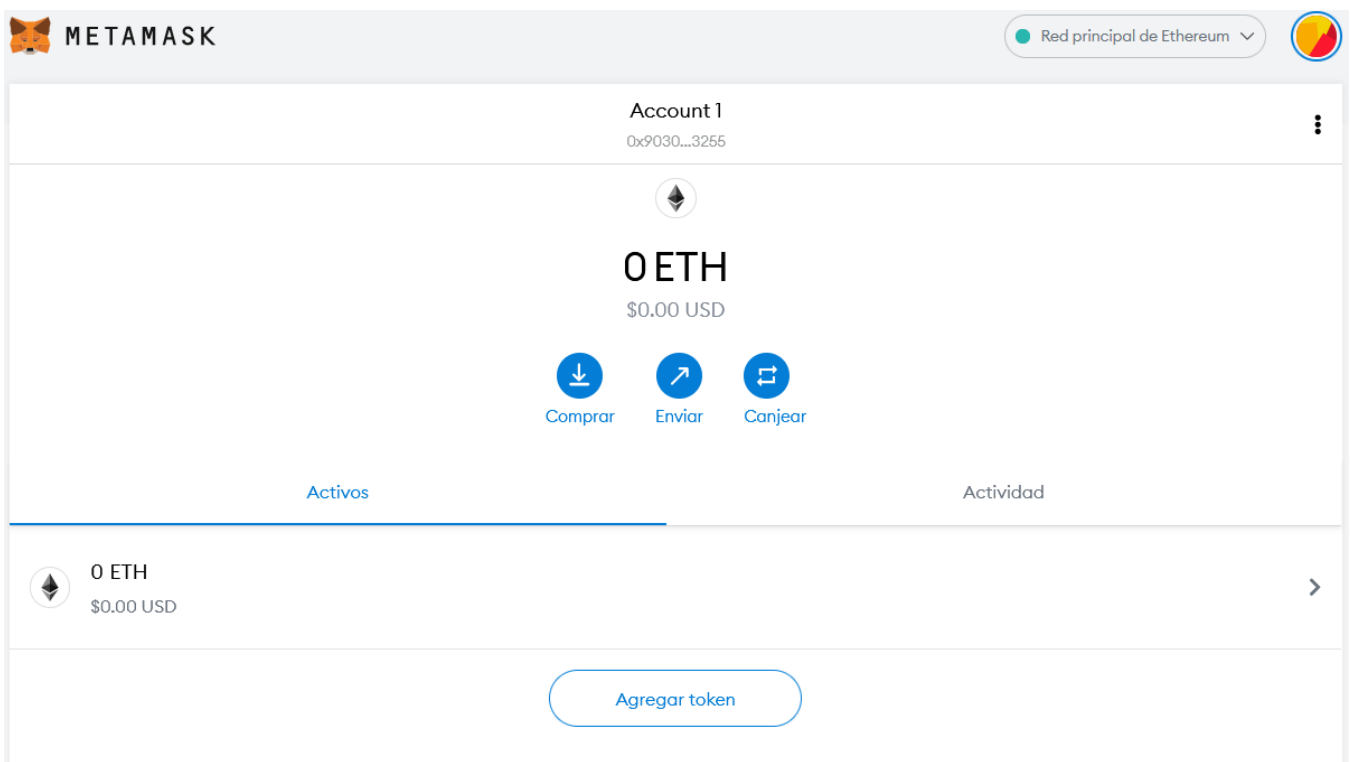
4.2.2. Despliegue con Remix II: Injected Web3 & Metamask

Seleccionando este entorno llamado *Injected Web3* podremos desplegar los contratos tanto en la red real como en una red de prueba siempre y cuando tengamos conectada nuestra cartera, en este caso usaremos Metamask. Entonces lo que vamos a hacer será instalar el plugin en nuestro navegador y crearnos una cuenta en Metamask para posteriormente poder desplegar nuestro contrato en una testnet de Ethereum.



Nos pedirá que creamos una contraseña para dicha cuenta. Tras esto se nos proporcionará nuestra frase de recuperación, esta frase está compuesta de 12 palabras en inglés que sirven para recuperar nuestra cuenta en caso de olvido de contraseña, por lo que es muy importante guardarla tanto en papel, como aprenderla de memoria o utilizar algún software de control de contraseñas. Tras guardar nuestra frase, pasaremos a la siguiente pantalla donde nos pedirá que introduzcamos la frase en el orden correcto pulsando palabra por palabra.

Con esto ya tendremos nuestra cartera a la vista, pero aún si fondos, en los pasos posteriores veremos cómo cargarla. Ahora vamos a ver de qué se compone nuestra cartera:



Lo primero que nos encontramos en la esquina superior derecha de la imagen anterior es con la red de Ethereum donde nos encontramos, ahora mismo la red principal, la cual cambiaremos a una de las testnets disponibles para poder probar nuestro desarrollo, por ejemplo, la red de prueba Rinkeby.

Lo siguiente que vemos en la parte central de la pantalla es el nombre de la cuenta en la que estamos, así como su dirección, que como ya sabemos, son una serie de caracteres únicos para cada dirección de Ethereum. Cada persona puede tener infinitas direcciones y con un contenido distinto en cada una de ellas, pero tenemos que tener claro que la misma cartera puede almacenarlas todas y el usuario ir cambiando de una a otra, lo cual es idóneo para realizar pruebas. En nuestro caso, usaremos el mismo supuesto anterior, y tendremos una dirección para la aerolínea (creadora del contrato) y otra dirección para un centro sanitario.

En la pestaña de activos podemos ver la lista de los diferentes tokens que posee esta dirección, es decir, aquí aparecerá cada token que agreguemos y el valor del que disponemos.

En la pestaña de actividad podremos encontrar el historial de esta cuenta, las veces que ha recibido y enviado cualquier token.

Con el botón de agregar token se pueden añadir monedas que Metamask no reconoce nativamente, lo cual de momento no nos interesa mucho.

Por último, tenemos los botones de comprar, enviar e intercambiar tokens, que se encargan de justamente eso.

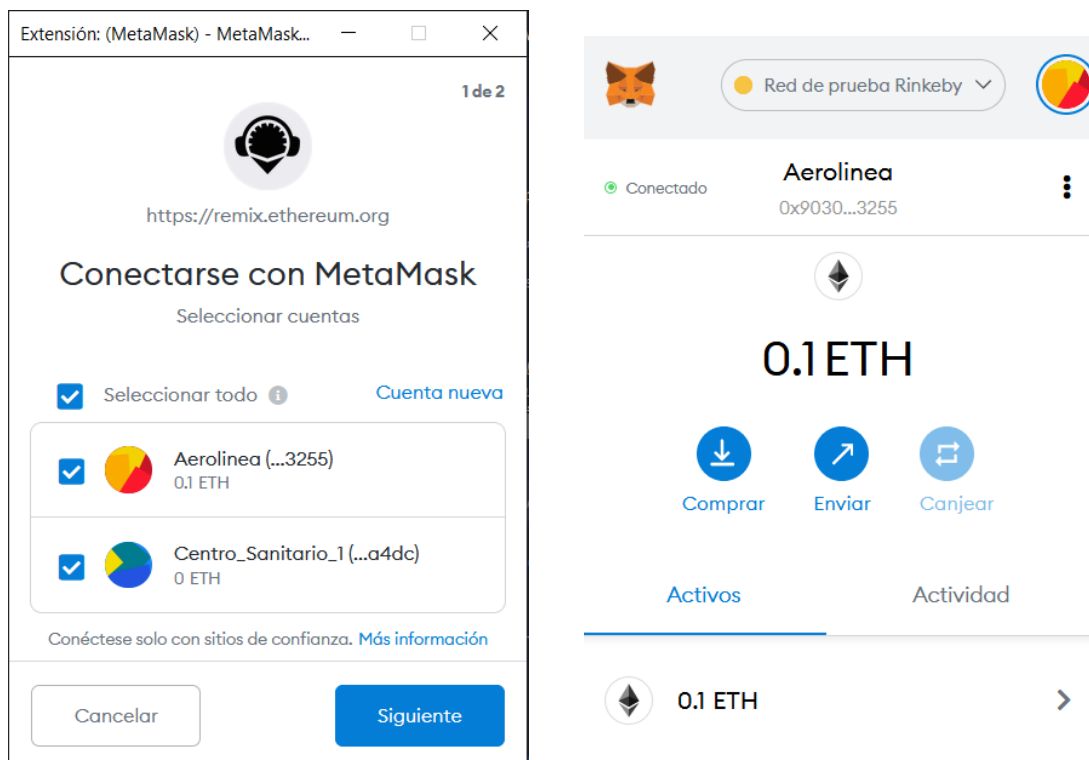
Como ya sabemos, para poder desplegar nuestros contratos inteligentes e interactuar con ellos necesitaremos que nuestra cuenta tenga saldo en ETH, dado que es la moneda utilizada para pagar a los mineros que confirman las transacciones. Por lo que lo primero que debemos hacer es recargar nuestra cuenta con un poco de Ether desde la siguiente página web: <https://www.testnet.help/en/ethfaucet/rinkeby> . Reclamar nuestro Ether para la red de pruebas es sencillo, sólo debemos rellenar nuestra dirección, aceptar una serie de puntos, rellenar el captcha y cerciorarnos que el indicador de batería es superior al 4% (esto es así para que nadie pueda colapsar la red al reclamar Ether infinito, al igual, que solo podremos realizar esta operación una vez cada 24h). Tras esto, una transacción es creada y podemos visualizarla pulsando sobre ella o copiando su hash en el buscador del siguiente enlace: <https://rinkeby.etherscan.io/> desde donde podremos visualizar gran cantidad de información sobre la red de pruebas Rinkeby usada en nuestro caso.

Si ahora refrescamos la página de Metamask y nos situamos en la red de prueba Rinkeby, veremos que tenemos nuestro 0'1 ETH de la transacción anterior. Evidentemente, estos Ethers son ficticios únicamente válidos para esta red de pruebas.

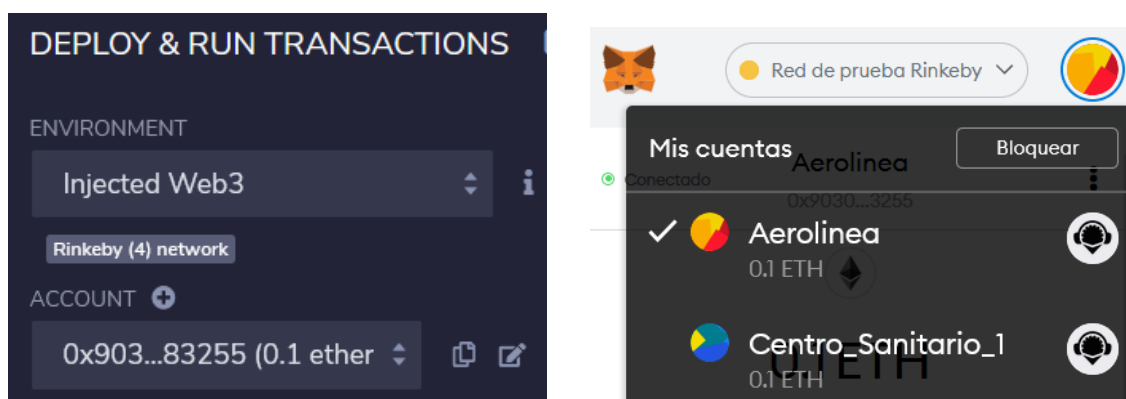
Vamos a realizar el proceso anterior para una cuenta nueva que crearemos en nuestra cartera para poder proseguir con el despliegue en *Injected Web3*. Así, tendremos dos cuentas, la primera que llamamos "Aerolínea" cuya dirección es la siguiente: "0x90306bA64747EDc16649288BFBD314943E283255" y 0'1ETH y la segunda, llamada "Centro_Sanitario_1", "0x324D4bE56BeF52FCBe71C22EbD8fe7f480c3a4DC" con 0'1 ETH también, cantidad que será suficiente para el caso que nos ocupa.

Ahora debemos conectar nuestra cartera de Metamask con el IDE de Ethereum. Para

ellos nos vamos a Remix y seleccionamos en la sección de *Deploy and Run Transactions*, el entorno con el que vamos a trabajar, *Injected Web3*. En el momento que lo seleccionamos, nos salta automáticamente la extensión de Metamask, mostrándonos las cuentas recién creadas de nuestra cartera. Seleccionamos ambas, ya que queremos conectar las dos. Cuando terminamos, ya tenemos conectada nuestra cartera con Remix para hacer el despliegue en la red de prueba Rinkeby. Para corroborarlo, si pulsamos en el icono de la extensión del navegador, vemos la imagen de la derecha, donde podemos comprobar que la cuenta de la Aerolínea está conectada:



Desde este último lugar, podemos cambiar de la cuenta de la aerolínea a la cuenta del centro sanitario, y se verá reflejado instantáneamente en el menú de las cuentas en Remix.



Una vez que tenemos la cuenta de la aerolínea seleccionada, podemos proceder a desplegar el contrato como en el caso anterior. Veremos que la cuenta de la aerolínea tendrá menos de 0.1 ETH en el momento que se despliegue el contrato, debido al coste del despliegue (que es bastante bajo al estar en una red de pruebas). Algo parecido veremos cuando la aerolínea registre los datos de un pasajero, dado que se

le restará al valor de la cuenta el coste por cada transacción.

Nota: Es muy importante cambiar el nombre al nombre de nuestro contrato del fichero autogenerado por Remix `deploy_web3.js` del directorio `scripts` ya que este fichero será el encargado del despliegue de nuestro contrato. En este fichero podemos ver que se indica la cuenta que le proporcionamos desde Remix, el bytecode de nuestro contrato, también indica la cuenta que lo despliega y el coste de despliegue del contrato si estuviese en un entorno real.

```
(async () => {
  try {
    console.log('Running deployWithWeb3 script...')

    const contractName = 'SmartContractHavel' // Change this for other contract
    const constructorArgs = [] // Put constructor args (if any) here for your contract

    // Note that the script needs the ABI which is generated from the compilation artifact.
    // Make sure contract is compiled and artifacts are generated
    const artifactsPath = `browser/contracts/artifacts/${contractName}.json` // Change this for different path

    const metadata = JSON.parse(await remix.call('fileManager', 'getFile', artifactsPath))
    const accounts = await web3.eth.getAccounts()

    let contract = new web3.eth.Contract(metadata.abi)

    contract = contract.deploy({
      data: metadata.data.bytecode.object,
      arguments: constructorArgs
    })

    const newContractInstance = await contract.send({
      from: accounts[0],
      gas: 1500000,
      gasPrice: '30000000000'
    })
    console.log('Contract deployed at address: ', newContractInstance.options.address)
  } catch (e) {
    console.log(e.message)
  }
})()
```


Una vez pulsado el botón de *Deploy* tendremos que confirmar la información del despliegue mostrada desde la ventana emergente de Metamask con la cuenta de la aerolínea. Y al contrario que en el caso anterior, el despliegue en este entorno tarda unos segundos más debido a que es una red real, al igual que la validación de las transacciones generadas por las funciones que escriben en la red.

Una vez desplegado, obtenemos la dirección de nuestro contrato en la red de pruebas Rinkeby: “0xd2fE7127C3D6756AD2d69648166Cb904a4aBA2ea”

```
creation of SmartContractTravel pending...

[block:9057244 txIndex:26] from: 0x903...83255 to: SmartContractTravel.(constructor) value: 0 wei
data: 0x608...60033 logs: 0 hash: 0xfbc...fc54e
```

Para interactuar con el contrato desde Remix, podremos hacerlo prácticamente igual que en el caso anterior, con la única diferencia de que por cada transacción que ejecutemos nos aparecerá una ventana emergente de Metamask parecida a la anterior para que firmemos la transacción generada con la cuenta que corresponda, por lo que debemos cerciorarnos desde la extensión de Metamask qué cuenta estamos utilizando según la función a la que estemos llamando.

También en este caso, cualquier usuario de la red con la dirección del contrato podrá interactuar con él, es decir, siempre que su dirección sea la creadora del contrato (la aerolínea) o que posea la dirección del contrato al recibirla por correo electrónico (el centro sanitario). Entonces, una vez que el contrato está desplegado en la blockchain

podemos acceder a él a nivel de usuario desde cualquier lugar utilizando, por ejemplo, el enlace anterior a la página de *etherscan* e introduciendo en el buscador la dirección de nuestro contrato como vemos a continuación:

Etherscan Rinkeby Testnet Network

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

Home Blockchain Tokens Misc Rinkeby

Contract 0xd2fE7127C3D6756AD2d69648166Cb904a4aBA2ea

Contract Overview

Balance: 0 Ether

More Info

My Name Tag: Not Available

Contract Creator: 0x90306ba64747edc166... at txn 0xfbc3dd2fd07ee047bc1...

Transactions Contract Events

Latest 2 from a total of 2 transactions

Txn Hash	Method	Block	Age	From	To	Value	Txn
0xe0b3f0b6967f2757850...	0x6ac9b0b1	9057296	4 mins ago	0x324d4be56bef52fcbe7...	IN 0xd2fe7127c3d6756ad2...	0 Ether	0.00
0xfbc3dd2fd07ee047bc1...	0x60806040	9057244	18 mins ago	0x90306ba64747edc166...	IN Contract Creation	0 Ether	0.00

Ahora debemos de verificar y publicar el contrato, esto lo haremos desde la pestaña *Contract* y sólo podrá hacerlo el creador del contrato introduciendo la dirección del contrato, el tipo del compilador, la versión del código utilizada y la licencia utilizada:

Verify & Publish Contract Source Code

COMPILER TYPE AND VERSION SELECTION

Source code verification provides **transparency** for users interacting with smart contracts. By uploading the source code, Etherscan will match the compiled code with that on the blockchain. Just like contracts, a "smart contract" should provide end users with more information on what they are "digitally signing" for and give users an opportunity to audit the code to independently verify that it actually does what it is supposed to do.

Please enter the Contract Address you would like to verify

0xd2fE7127C3D6756AD2d69648166Cb904a4aBA2ea

Please select Compiler Type

Solidity (Single file)

Please select Compiler Version

v0.7.6+commit.7338295f

☒ Un-Check to show all nightly Commits also

Please select Open Source License Type

5) GNU General Public License v3.0 (GNU GPLv3)

☒ I agree to the terms of service

Continue Reset

En la siguiente página deberemos copiar el código del contrato en el cuadro de texto habilitado para ello y seleccionamos el tipo de optimización, que en nuestro caso no hemos utilizado. Continuamos y comprobamos que se haya generado correctamente el ByteCode y el ABI del contrato:

```
Contract Source Code      Compiler Output

Compiler debug log:
✔ Note: Contract was created during TxHash# 0xfbc3dd2fd07ee047bc1e9188acd04413d62fed9939946697711949c0356fc54e
👍 Successfully generated ByteCode and ABI for Contract Address [0xd2f7E7127C3D6756AD2d69648166Cb904a4aBA2ea]

Compiler Version: v0.7.6+commit.7338295f
Optimization Enabled: 0
Runs: 200

ContractName:

SmartContractTravel

ContractBytecode:

6080604052600060015534801561001557600080fd5b5033600080610100a81548173fffffffffffffffffffffffffffffffffffffffff02191
100656000396000f3fe08060405234801561001057600080fd5b50600436106100575760003560e01c806320453a881461005c5780636ac9b0
14610314575b600080fd5b6100a86004803603604081101561007257600080fd5b8101908080359060200190929190803573ffffffffffffff
a600480360360e08110156100c057600080fd5b8101908080359060200190640100000000811156100dd57600080fd5b820183602082011156101445760008
61011157600080fd5b9091929391929390803590602001906401000000008111561013257600080fd5b820183602082011156101445760008
```

```

1Type": "uint256", "name": "_id", "type": "uint256"}}, "name": "Datos_Pasajero_Registrado", "outputs": [{"internalType": "string", "name": "nombre", "type": "string"},
ng", "name": "", "type": "string"}, {"internalType": "string", "name": "", "type": "string"}, {"internalType": "string", "name": "", "type": "string"},
ng", "name": "", "type": "string"}, {"internalType": "address", "name": "dir_csantario", "type": "address"}, {"internalType": "string", "name": "", "type": "string"},
256", "name": "id_npasajero", "type": "uint256"}], "stateMutability": "view", "type": "function"}, {"inputs":
256", "name": "_id_npasajero", "type": "uint256"}, {"internalType": "address", "name": "dir_csantario", "type": "address"}], "name": "Registrar_cambio_test", "outputs":
"nonpayable", "type": "function"}, {"inputs": [{"internalType": "string", "name": "_nombre", "type": "string"},
ng", "name": "_apellidos", "type": "string"}, {"internalType": "string", "name": "_dni", "type": "string"},
ng", "name": "_nif_csantario", "type": "string"}, {"internalType": "string", "name": "_nombre_csantario", "type": "string"},
ess", "name": "dir_csantario", "type": "address"}, {"internalType": "string", "name": "_resultado_test", "type": "string"}], "name": "Registrar_datos_pasajero", "outputs":
"nonpayable", "type": "function"}, {"inputs": [], "name": "destroy", "outputs": [], "stateMutability": "nonpayable", "type": "function"}, {"inputs":
, "outputs": [{"internalType": "uint256", "name": "", "type": "uint256"}], "stateMutability": "view", "type": "function"}]}

```

Ahora que ya sabemos que es capaz de leer las variables del contrato y el ByteCode, volvemos a la dirección del contrato y comprobamos que el contrato está verificado con un check en verde. Por tanto, ya tenemos accesibles las funciones del contrato, tanto las de lectura como las de escritura. Para usar estas últimas será necesario conectar a través de Web3 nuestras direcciones tanto de la aerolínea como del centro sanitario de nuestra cartera en Metamask como hicimos anteriormente al conectarla con Remix.

Una vez hecho esto, comprobamos que el pasajero introducido anteriormente desde Remix, se encuentra en la información del contrato, corroborando que el contador $n_pasajero$ se encuentra a 1, lo que quiere decir que un centro sanitario ya registro al primer pasajero de un vuelo:

The screenshot shows the 'Contract' tab of the application. At the top, there are three tabs: 'Transactions', 'Contract' (which is selected and has a green checkmark), and 'Events'. Below the tabs are three buttons: 'Code', 'Read Contract' (which is highlighted in a darker grey), and 'Write Contract'. Underneath these buttons is a section titled 'Read Contract Information' with a document icon. This section contains a list of variables: '1. Datos_Pasajero_Registrado', '2. n_pasajero', and '1 uint256'.

Ahora vamos a comprobar que desde aquí también podemos registrar a un pasajero de un vuelo con la cuenta del centro sanitario. Introducimos los datos referentes al pasajero en cuestión y firmamos la transacción con la cuenta del centro sanitario en Metamask.

Transactions

Contract ✓

Events

Code

Read Contract

Write Contract

Connected - Web3 [0x324D...a4DC]

1. Registrar_cambio_test

2. Registrar_datos_pasajero

_nombre (string)

Jose

_apellidos (string)

Vazquez Garcia

_dni (string)

22222222X

_nif_csantitario (string)

111111111

_nombre_csantitario (string)

Centro1

_dir_csantitario (address)

0x324D4bE56BeF52FCBe71C22EbD8fe7f480c3a4DC

_resultado_test (string)

NEGATIVO

Write

Extensión: (MetaMask) - MetaMask ...

Red de prueba Rinkeby

Centro_Sanita... → 0xd2fE...A2ea

https://rinkeby.etherscan.io

INTERACCIÓN CON EL CONTRATO

0

DETAILS DATA

GAS FEE

0.000317

No hay tasa de conversión disponible

Precio de gas (GWEI)

1,000000008

Límite de gas

317326

AMOUNT + GAS FEE

TOTAL

0.000317

No hay tasa de conversión disponible

Rechazar

Confirmar

Y comprobamos otra vez que se pueden leer los datos del pasajero y que el contador se ha incrementado en uno, por lo tanto, *n_pasajero* es igual a 2, funcionando perfectamente nuestro contrato inteligente desplegado en Ethereum:

Transactions

Contract ✓

Events

Code

Read Contract

Write Contract

Read Contract Information

1. Datos_Pasajero_Registrado

2. n_pasajero

2 uint256

4.3. Integración web

En esta sección, una vez visto el objetivo y funcionamiento del contrato, vamos a tratar de desarrollar un proyecto real con el ya comentado framework Truffle y así profundizar más en el conocimiento del mismo para dotar de una interfaz web básica a las funcionalidades de nuestro Smart Contract Travel.



Para ello son necesarias una serie de instalaciones previas de las herramientas que vamos a usar, como son *node.js*, *web3.js*, *Truffle*, *Git*, *Metamask* y nuestro IDE offline de preferencia, en mi caso es *Visual Studio Code* dado que implementa una consola dentro del propio IDE y los pluggins de Solidity son muy útiles a la hora de desarrollar.

Tanto *node.js* como el *Visual Studio Code* los descargaremos desde su página oficial, haciendo una instalación por defecto. Una vez tengamos la última versión del *Visual Studio Code*, accedemos y desde la pestaña de extensiones buscamos “Solidity”, e instalaremos los dos pluggins comentados anteriormente: el primero se llama “Solidity” y su autor es Juan Blanco, esta extensión sirve para colorear correctamente el código .sol; el segundo, de tintinweb se llama “Solidity Visual Developer” y se encargará de la detección de errores.

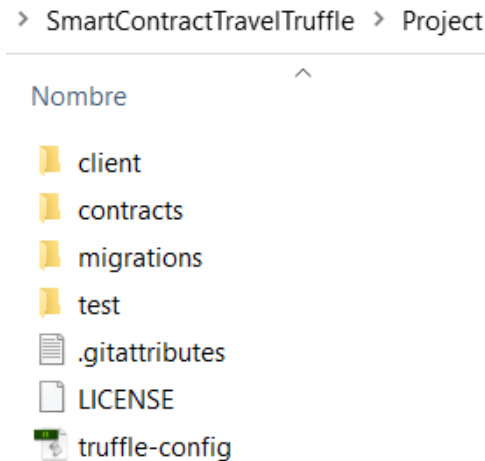
Para la instalación de *web3.js* y *Truffle*, nos situaremos en la carpeta del proyecto con ***cd SmartContractTravelTruffle*** y ejecutaremos ***npm install web3*** y ***npm install -g truffle***, una vez aceptados todos los mensajes que salen, tendremos *Truffle* instalado en este directorio, para comprobarlo ejecutamos ***truffle version*** y vemos lo siguiente:

```
Truffle v5.4.3 (core: 5.4.3)
Solidity v0.5.16 (solc-js)
Node v14.17.4
Web3.js v1.5.0
```

Una vez descargadas y configuradas nuestras herramientas, vamos a crear un proyecto en el que utilizaremos una **Truffle box** para obtener una estructura de código ya creada con **React** y añadiremos nuestro contrato para dotarlo de la funcionalidad deseada.

Estas “Truffle box” son plantillas creadas por su propia comunidad que implementan un lenguaje o funcionalidad diferente y son usadas para no perder tiempo diseñando y estructurando un proyecto al principio, ya que si vamos a crear una aplicación web siempre tendremos la misma estructura o si por el contrario fuésemos a crear una aplicación Android, la estructura inicial sería la misma. Para ello nos dirigimos a la página oficial de Truffle y buscamos la plantilla llamada “react” que ya contiene una aplicación web básica que modificaremos con nuestros propios contratos y

funcionalidad. Para descargarla e instalarla nos situamos en nuestro directorio de trabajo, donde queremos que esté la raíz de nuestro proyecto, en este caso con los comandos ***cd Project*** y ***truffle unbox react***. Con esto, descargamos el proyecto completo en el directorio anterior, donde la estructura de carpetas quedará como sigue:



El directorio *client* contiene la aplicación React y será desde donde lanzaremos la aplicación web.

Dentro de los *contracts* nos vienen dos contratos Solidity por defecto con la aplicación base, eliminaremos todos a excepción del llamado *Migrations.sol*.

El directorio *migrations* contiene una parte fundamental de Truffle, equivalente a la compilación en otros lenguajes ya que en Solidity necesitamos compilar y migrar los contratos cada vez que se modifique algo en los mismos, desplegándolos en la blockchain que hayamos configurado (en este caso en localhost) para poder hacer correctamente las llamadas a la misma. Por lo tanto, los archivos de migraciones se usan para desplegar contratos en la blockchain y su nombre empieza con un número que indica el orden en el que debe ejecutarse.

La carpeta *test* contiene los test unitarios de uno de los contratos de ejemplo, el cual borraremos y si tenemos tiempo, será donde guardaremos el test unitario para nuestro contrato.

```
Unbox successful, sweet!
Commands:
  Compile:          truffle compile
  Migrate:          truffle migrate
  Test contracts:   truffle test
  Test dapp:        cd client && npm test
  Run dev server:   cd client && npm run start
  Build for production: cd client && npm run build
```

Truffle I: Configuración del entorno de trabajo

Lo primero que debemos hacer para poder compilar nuestro contrato inteligente con la suite de Truffle es editar su archivo de configuración *truffle-config.js* y especificar la versión del compilador que queremos que use, dado que por defecto está configurado con una versión bastante anticuada, la 0.5. Para ello, el fichero de configuración debe quedar como sigue, habiéndole añadido la sección *compiler*:

```
const path = require("path");

module.exports = {
  // See <http://truffleframework.com/docs/advanced/configuration>
  // to customize your Truffle configuration!
  contracts_build_directory: path.join(__dirname, "client/src/contracts"),
  networks: {
    develop: {
      port: 8545
    }
  },
  compilers: {
    solc: {
      version: "0.7.6", // A version or constraint - Ex. "^0.5.0"
    }
  }
};
```

Hemos especificado la misma versión que la utilizada al probar nuestro contrato en Remix para asegurarnos de su correcta compilación y funcionamiento.

Una vez hecho esto, podemos iniciar la consola de Truffle con el comando **truffle develop** el cual es el encargado de iniciar una blockchain local en *localhost:8545*, con costo computacional cero, por lo que las transacciones tampoco nos costarán nada y también nos inicia 10 cuentas con 100 ETH por defecto y las claves privadas de cada una de ellas. En esta consola podremos consultar el saldo de estas cuentas con **web3.eth.getBalance("<dirección>")**, realizar transacciones entre ellas con **web3.eth.sendTransaction({from:"",to:"",value:"..."})** y visualizar los detalles de las mismas, así como obtener el número de bloque actual de la cadena con **web3.eth.getBlockNumber()**, y otras muchas opciones que veremos a continuación.


```

velTruffle\Project>truffle develop
Truffle Develop started at http://127.0.0.1:8545/

Accounts:
(0) 0xb7bb99eb910095bb78c0aa65d3d56c08ca912007
(1) 0xb6768c994ad8bc4bfd2f0ecfd45ae864f021ef7b
(2) 0x2cdb018be97c0f581c15d3a56f6d2d5d554d67bc
(3) 0xf99e131e3f910e3bfdfaafbc80043ce8008bd6
(4) 0x7cc59bed9fc2915e82a416183fba7e6aebcec22f
(5) 0xe92d86bd6b69184b7ae48f00299588e4b6847411
(6) 0x55ccf7ec78d050148b91f18a1ce67bfaab8ec627
(7) 0xc15be20af39e251ac81d7cecde39250fc1d4a7e1
(8) 0x7cf9ef8e2fcf3522f55a5002ff70d448791d6e84
(9) 0xbbca2bb9355a1e06070da3347379ff785d0ea386

Private Keys:
(0) b996beab75faff64ef8f9d598c7e178736b53a7bc98a17bb4e007770b9dca31a
(1) f50039c982110bf8a1947b46c78aa22fc9013ee50d631914d09176af42446e26
(2) 6e0695ad55fee3aef2671f49b18f052237a4d08e6df2acee63fae64c0f7cb849
(3) 90d315822c23a491066e740184016fb7af4ea85233ec99bec910853c1ff151
(4) 3385f5c9a1e561f94ba42411b268e32fae7e3be735b1434f2b4f3b64be123292
(5) d2839e2083fb5f7188b9a02b4d4af88953973d689f544388e350f8d2b2c5bbe1
(6) e55b4ebcd44a4eddf5c6251625d15a34ee6beb31eaa4c160168a3a0f0bef0138
(7) efd0b94239346c580e8cc51eb34190d325b1165add7ca99308593eadd62c5a80
(8) d81f8ae5cef34c8f44a5c0bd7edc69ef939ce0bc936e9a670c5e24907b00f0ba
(9) 2c2b13555a4ef169354a45bb9fa57d461c0d365eb590e2fb2d621acea07f75f8

Mnemonic: wrestle despair relax believe bone pulse shallow detect choice olympic shuffle buffalo

⚠ Important ⚠ : This mnemonic was created for you by Truffle. It is not secure.
Ensure you do not use it on production blockchains, or else you risk losing funds.

truffle(develop)>

```

Truffle II: Compilación y despliegue de contratos inteligentes

Lo siguiente que debemos hacer es crear nuestro contrato inteligente llamado “SmartContractTravel.sol” en el directorio *contracts*, copiando y pegando el código desarrollado y probado anteriormente.

Ahora que ya tenemos nuestro contrato implementado, deberemos compilarlo para que una vez compilado lo podamos subir a nuestra red local para ejecutarlo y probarlo.

Entonces, para continuar, creamos el script que desplegará nuestro contrato en la blockchain. En un principio, este script es bastante simple dado que Truffle nos facilita mucho las cosas. Si nos fijamos en el apartado anterior “Remix II”, el script generado por el IDE online (*deploy_web3.js*) para este fin era algo más complejo. En este caso, lo llamamos *2_deploy_contracts* dado que será el segundo en ejecutarse del directorio *migrations* por detrás de *1_initial_migration.js*.

```

migrations > JS 2_deploy_contracts.js > ...
1   var SmartContractTravel = artifacts.require("./SmartContractTravel.sol");
2
3   module.exports = function(deployer) {
4     ...
4     deployer.deploy(SmartContractTravel);
5   };

```

Una vez hecho esto, debemos compilar y migrar los contratos. Por lo que nos dirigimos a la consola de Truffle inicializada anteriormente y compilamos los contratos para ver que lo hace sin errores ejecutando el comando **compile**:


```
truffle(develop)> compile

Compiling your contracts...
=====
> Compiling .\contracts\Migrations.sol
> Compiling .\contracts\SmartContractTravel.sol
> Artifacts written to C:\Users\Josele\Desktop\Año 2020-2021\Documentos TFG\TFG-SmartContractTravel\TFG\SmartContractTravelTruffle\Project\client\src\contracts
> Compiled successfully using:
  - solc: 0.7.6+commit.7338295f.Emscripten.clang
```

Para migrar los contratos ejecutamos **migrate**. Este comando ejecuta todos los archivos de la carpeta *migrations* y antes de esto, los recompila de nuevo:

```
truffle(develop)> migrate

Compiling your contracts...
=====
> Compiling .\contracts\Migrations.sol
> Compiling .\contracts\SmartContractTravel.sol
> Artifacts written to C:\Users\Josele\Desktop\Año 2020-2021\Documentos TFG\TFG-SmartContractTravel\TFG\SmartContractTravelTruffle\Project\client\src\contracts
> Compiled successfully using:
  - solc: 0.7.6+commit.7338295f.Emscripten.clang

Starting migrations...
=====
> Network name:      'develop'
> Network id:        5777
> Block gas limit: 6721975 (0x6691b7)

1_initial_migration.js
=====

Replacing 'Migrations'
-----
> transaction hash: 0x03c7b9c777f2d4fdaa801ee5f9440066f2fe00f57d0fcf2cb5b31bb3e70a49f3
> Blocks: 0        Seconds: 0
> contract address: 0x2Ad0E2d9002ae0aaf75a8B33dDB243229Cb35F4A
> block number:     1
> block timestamp:  1628271997
> account:          0xB7bb99EB910095BB78c0AA65d3D56C08Ca912007
> balance:          99.99968161
> gas used:         159195 (0x26ddb)
> gas price:        2 gwei
> value sent:       0 ETH
> total cost:       0.00031839 ETH

> Saving migration to chain.
> Saving artifacts
-----
> Total cost:       0.00031839 ETH

2_deploy_contracts.js
=====

Replacing 'SmartContractTravel'
-----
> transaction hash: 0x1e7054e2e59c965bdd766e9ec2a8abffdd49c5f36a46055a14e19d13fd296da1
> Blocks: 0        Seconds: 0
> contract address: 0xEF2f61dCd7BF1BF08Ac8f86952b33c499Ae5f10D
> block number:     3
> block timestamp:  1628271998
> account:          0xB7bb99EB910095BB78c0AA65d3D56C08Ca912007
> balance:          99.997434582
> gas used:         1081176 (0x107f58)
> gas price:        2 gwei
> value sent:       0 ETH
> total cost:       0.002162352 ETH

> Saving migration to chain.
> Saving artifacts
-----
> Total cost:       0.002162352 ETH

Summary
=====
> Total deployments: 2
> Final cost:       0.002480742 ETH

- Blocks: 0        Seconds: 0
- Saving migration to chain.
- Blocks: 0        Seconds: 0
- Saving migration to chain.
```


Una vez comprobados los datos del pasajero, vamos a llamar a la función de modificar el test de un pasajero, pero primero lo haremos introduciendo por parámetros una dirección que no coincida con la que está llamando a la función de nuestro contrato para comprobar que nos devuelve error.

[illegible]

Ahora vamos a registrar correctamente el cambio en el test del pasajero y a comprobar que efectivamente pasa de ser negativo a ser positivo para el primer pasajero registrado.

[illegible]

Por último, debemos comprobar que en este entorno el destructor del contrato funciona correctamente, y solo puede ser ejecutado por la aerolínea. Por lo que primero, probamos a llamar a la función con la cuenta del centro sanitario (*accounts[1]*):

[illegible]

Al comprobar que reporta el error correctamente, destruimos nuestro contrato de la red actual con la dirección de la aerolínea (*accounts[0]* en este caso) y comprobamos que queda inaccesible:

[illegible]

Truffle II: Test unitarios

Podemos automatizar esta serie de pruebas realizadas en el apartado anterior a través de los test unitarios que podremos desarrollar tanto en el lenguaje de programación Solidity como en JavaScript o TypeScript, en nuestro caso, lo haremos con la segunda opción.

Truffle se construye sobre el marco de prueba *Mocha* para proporcionar un marco de prueba sólido desde el cual escribir nuestras pruebas en JavaScript. Mocha es un marco de prueba de JavaScript rico en funciones que se ejecuta en *Node.js* y en el navegador, lo que hace que las pruebas asíncronas sean simples. Las pruebas se ejecutan en serie, lo que permite informes flexibles y precisos, al tiempo que asigna las excepciones no detectadas a los casos de prueba correctos.

Estos test debemos almacenarlos en el directorio *test* de nuestro espacio de trabajo y para ejecutarlos con Truffle deberemos ejecutar desde la terminal de comandos situada en nuestro directorio de trabajo el comando **truffle test**, que será el encargado de ejecutar todos los archivos de prueba de dicho directorio.

En este caso, hemos desarrollado siete test básicos para que probar algunas de las funcionalidades de nuestro contrato se haga más sencillo y automático.

Este archivo de test, llamado "*smartContractTravel.js*" comienza declarando la hebra de trabajo principal, que será la encargada de ejecutar los test e importando el contrato a probar almacenándolo en la variable *SmartContractTravel*, desde donde podremos acceder a las funciones del contrato:

```
const { isMainThread } = require("worker_threads");

const SmartContractTravel = artifacts.require("./SmartContractTravel.sol");

contract("SmartContractTravel", accounts => {
  ...
});
```

En el último bloque de código anterior, la palabra *contract* se define como una variable global la cual nos proporcionará todo el marco de pruebas, levantando por detrás una blockchain contra la que se ejecutarán los test definidos dentro de dicho bloque de código. También se le especifican la variable donde se encuentra el contrato a testear (almacenada anteriormente) y el vector de direcciones asociado a dicha blockchain, cuyo funcionamiento ya vimos anteriormente.

```
it("El numero de pasajeros debe ser 0 al iniciar el contrato", async () => {
  const smartContractTravel = await SmartContractTravel.deployed({from: accounts[0]});
  const n_pasajero = await smartContractTravel.n_pasajero({from: accounts[0]});
  assert.equal(n_pasajero, 0);
});
```

En este primer test, se comprueba que la variable *n_pasajero* está inicializada correctamente al desplegar el contrato. Lo primero que se hace es establecer el mensaje que mostrará el resultado del test y posteriormente se crea una llamada asíncrona a dicho test. Como veremos en los test posteriores, lo primero que debemos hacer en todos y cada uno de ellos es desplegar el contrato en cuestión (en

este caso se hace desde la primera dirección) y almacenarlo en una variable. Lo segundo que hacemos es almacenar la variable `n_pasajero` cuyo valor debería ser 0, dado que todavía no se ha interactuado de ninguna forma con el contrato. Por último, con `assert` nos fijamos que sean iguales el valor de la variable del contrato con el valor esperado. Esta es la estructura básica de un test, veamos los siguientes:

```
it("El numero de pasajeros debe ser 1 al registrar un pasajero", async () => {
  const smartContractTravel = await SmartContractTravel.deployed();
  const dir_csanitario = accounts[1];

  await smartContractTravel.Registrar_datos_pasajero("Juan", "Garcia Lopez", "11111111X", "11111111", "Centro1", dir_csanitario, "NEGATIVO", {from: accounts[1]});

  const n_pasajero = await smartContractTravel.n_pasajero({from: accounts[0]});
  assert.equal(n_pasajero, 1);
});
```

En este caso, comprobamos que cada vez que se introduzca un pasajero, la variable `n_pasajero` se incrementa correctamente, para así asegurarnos que el identificador de cada pasajero funciona correctamente. Por lo que, desplegamos el contrato y establecemos la dirección del centro sanitario a la segunda de la lista de direcciones, para posteriormente llamar a la función que registra la información de los pasajeros por parte del centro sanitario. Por último, comprobamos que sean iguales el valor de la variable y el valor esperado.

```
it("La direccion que llama a la funcion 'Registrar_datos_pasajero' coincide siempre con la direccion introducida por parametros", async () => {
  const smartContractTravel = await SmartContractTravel.deployed();

  const dir_csanitario = accounts[1];
  const dir_aerolinea = accounts[0];

  try {
    await smartContractTravel.Registrar_datos_pasajero("Juan", "Garcia Lopez", "11111111X", "11111111", "Centro1", dir_csanitario, "NEGATIVO", {from: dir_aerolinea});
  } catch (e) {
    assert.equal(e, "Error: Returned error: VM Exception while processing transaction: revert La direccion introducida no coincide con el centro sanitario que envia la transaccion -- Reason given: La direccion introducida no coincide con el centro sanitario que envia la transaccion.");
  }
});
```

Con este test, nos aseguramos que siempre saltará la excepción en nuestro contrato si tratamos de llamar a la función de registro de los datos del pasajero con una dirección diferente a la introducida por parámetros. Para ello, hacemos que falle introduciendo por parámetros la dirección del centro sanitario y llamando a la función con la dirección de la aerolínea. Rescatamos la excepción y comprobamos que lanza el mensaje correcto para la misma comprobando que es igual al que esperamos.

```

it("La direccion que llama a la funcion 'Registrar_cambio_test' coincide siempre con la direcc
ión introducida por parametros", async () => {
    const smartContractTravel = await SmartContractTravel.deployed();

    const dir_csanitario = accounts[1];
    const dir_aerolinea = accounts[0];

    try {
        await smartContractTravel.Registrar_datos_pasajero("Juan","Garcia Lopez","111111
111X","111111111","Centro1", dir_csanitario, "NEGATIVO", {from: dir_aerolinea});
        await smartContractTravel.Registrar_cambio_test(1, dir_csanitario, {from: dir_ae
rolinea});
    } catch (e) {
        assert.equal(e, "Error: Returned error: VM Exception while processing transactio
n: revert La direccion introducida no coincide con el centro sanitario que envia la transac
cion --
Reason given: La direccion introducida no coincide con el centro sanitario que envia la tra
nsaccion.");
    }
});

```

En este caso, hacemos algo parecido al caso anterior, pero llamando con direcciones diferentes (en la llamada y en el parámetro) a la función que registraría un cambio en el resultado del test de un pasajero (para ello, primero registramos a ese pasajero al que posteriormente le cambiaremos el resultado).

```

it("La direccion que llama a la funcion 'Datos_Pasajero_Registrado' coincide siempre con la
dirección creadora del contrato", async () => {
    const smartContractTravel = await SmartContractTravel.deployed({from: accounts[0]});
    const dir_csanitario = accounts[1];

    try {
        await smartContractTravel.Datos_Pasajero_Registrado(0, {from: dir_csanitario});
    } catch (e) {
        assert.equal(e, "Error: Returned error: VM Exception while processing transactio
n: revert Solo la aerolinea puede realizar esta funcion");
    }
});

```

A través de este test, comprobamos que la dirección que llama a la función que recupera los datos de un pasajero es siempre la correspondiente a la que despliega el contrato. Como en este caso lo despliega la primera dirección de la lista, hacemos que falle llamando a la función desde la segunda, es decir, saltará la excepción siempre que no coincidan ambas direcciones.


```

it("La direccion que llama a la funcion 'destroy' coincide siempre con la dirección creadora del contrato", async () => {
  const smartContractTravel = await SmartContractTravel.deployed({from: accounts[0]});
  const dir_csantario = accounts[1];

  try {
    await smartContractTravel.destroy({from: dir_csantario});
  } catch (e) {
    assert.equal(e, "Error: Returned error: VM Exception while processing transaction: revert Solo la aerolinea puede realizar esta funcion -- Reason given: Solo la aerolinea puede realizar esta funcion.");
  }
});

```

En este caso hacemos exactamente lo mismo que en el test anterior, pero comprobando que la dirección que llama a la función que destruye el contrato es siempre la misma que el que lo crea.

```

it("El resultado del test covid asociado al id del pasajero siempre cambia correctamente al llamar a la función 'Registrar_cambio_test'", async () => {
  const smartContractTravel = await SmartContractTravel.deployed();

  const dir_csantario = accounts[1];
  const dir_aerolinea = accounts[0];

  await smartContractTravel.Registrar_datos_pasajero("Juan","Garcia Lopez","111111111X","111111111","Centro1", dir_csantario, "NEGATIVO", {from: dir_csantario});
  await smartContractTravel.Registrar_cambio_test(1, dir_csantario, {from: dir_csantario});

  await smartContractTravel.Datos_Pasajero_Registrado(1, {from: dir_aerolinea}).then(function(result){
    assert.equal(result[6], "POSITIVO");
  });
});

```

Por último, comprobamos que el resultado del test de covid19 asociado al identificador del pasajero siempre cambia correctamente al llamar a la función cuya finalidad es esta. Para ello, desplegamos el contrato y registramos a un nuevo pasajero y su resultado negativo en la prueba, cuyo identificador de pasajero será 1 al ser el primero en registrarse. Luego, llamamos a la función que cambia el resultado del test pasándole este identificador por parámetros y comprobamos, llamando desde la dirección de la aerolínea a la función que nos devuelve los datos de dicho pasajero, que el resultado del test ha cambiado de negativo a positivo correctamente para dicho pasajero, dado que *result[6]* corresponde a la variable séptima devuelta por dicha función, la variable *resultado_test* del contrato la cual debería cambiar correctamente.

Para comprobar el correcto funcionamiento de estos test, como hemos comentado antes, bastará con ejecutar en una terminal situada en nuestro directorio de trabajo el comando **truffle test** como vemos a continuación.

Nota: Es imprescindible parar la blockchain local anterior saliéndonos de *truffle develop* pulsando dos veces Ctrl+C o escribiendo **.exit**. Esto implica que se borren todos los datos, dado que en el momento que la volvamos a levantar, volverá a crear todo el entorno de nuevo con los datos por defecto, por lo que nuestros contratos deberán volver a ser compilados, migrados, etc. Para evitar esto, podemos usar otras blockchains como Ganache, también comentada en este trabajo.

```
Using network 'test'.

Compiling your contracts...
=====
> Compiling .\contracts\Migrations.sol
> Compiling .\contracts\SmartContractTravel.sol
> Artifacts written to C:\Users\Josele\AppData\Local\Temp\test--15796-gVlXtN5Qumij
> Compiled successfully using:
   - solc: 0.7.6+commit.7338295f.Emscripten.clang

Contract: SmartContractTravel
  ✓ El numero de pasajeros debe ser 0 al iniciar el contrato (160ms)
  ✓ El numero de pasajeros debe ser 1 al registrar un pasajero (707ms)
  ✓ La direccion que llama a la funcion 'Registrar_datos_pasajero' coincide siempre con la dirección introducida por parametros (1015ms)
  ✓ La direccion que llama a la funcion 'Registrar_cambio_test' coincide siempre con la dirección introducida por parametros (369ms)
  ✓ La direccion que llama a la funcion 'Datos_Pasajero_Registrado' coincide siempre con la dirección creadora del contrato (134ms)
  ✓ El resultado del test covid asociado al id del pasajero siempre cambia correctamente al llamar a la funcion 'Registrar_cambio_test' (1012ms)

  ✓ La direccion que llama a la funcion 'destroy' coincide siempre con la dirección creadora del contrato (271ms)

7 passing (4s)
```

Lo primero que hace este comando es usar la red de pruebas y compilar de nuevo los contratos según la versión del compilador especificada en el archivo de configuración de Truffle. Posteriormente, ejecuta secuencialmente los test desarrollados para nuestro contrato.

Como podemos ver, el resultado de los siete test básicos desarrollados para probar nuestro SmartContractTravel son satisfactorios, por lo que podemos afirmar que el contrato funciona según lo deseado.

También nos indica el tiempo que ha requerido cada uno de los test para ejecutarse contra el contrato, lo que nos da una idea de la complejidad de los mismos y lo que tardan por ende en realizarse las llamadas a las funciones de cada uno de ellos en una red de pruebas como esta.

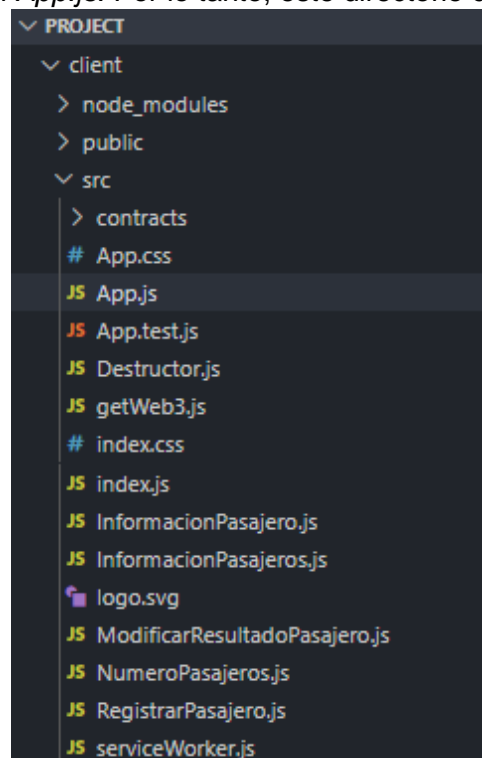
Truffle III: React-FrontEnd

En este tercer apartado, vamos a tratar de desarrollar una interfaz gráfica básica para operar a través de las funcionalidades de nuestro contrato inteligente. Esto como vimos anteriormente, lo haremos con React, que es una biblioteca de JavaScript de código abierto diseñada para crear interfaces de usuario con el objetivo de facilitar el desarrollo de aplicaciones web.

Antes de comenzar, es bueno recordar algunos conceptos claves que nos serán indispensables para desarrollar esta interfaz:

- Las operaciones de consulta, las cuales obtienen datos de la red tienen coste cero, dado que no se necesita validar transacciones.
- Las operaciones de escritura, las cuales insertan o modifican datos en la red tienen un coste transaccional, por lo que el usuario deberá pagarlas con su billetera (en este caso se seguirá usando Metamask).
- Cualquiera con la dirección del contrato podrá usarlo, aunque en la mayoría de operaciones de nuestro contrato se definen una serie de restricciones como ya hemos visto anteriormente.
- Los nodos tienen un *endpoint* al que conectarse, en nuestro caso, Truffle lo crea localmente en: 'http://127.0.0.1:8545/"/>.
- La API de los contratos se representa con su ABI, el cual nos permitirá interactuar con ellos. El ABI de nuestro contrato lo encontramos en un archivo JSON en el directorio *src/contracts* dentro de *client*.
- Los contratos, al igual que las cuentas, tienen un identificador único, su dirección.

Para comenzar, en nuestro directorio *client/src* nos crearemos un archivo de JavaScript por cada una de las funcionalidades de nuestro sistema, es decir, creamos un componente por cada función del contrato. Estos poseerán su propio estado y funciones de renderización, que serán usados para construir la interfaz de la aplicación en el archivo principal *App.js*. Por lo tanto, este directorio contiene lo siguiente:



Estos componentes deberán tener definidos tanto el ABI del contrato, como la dirección del mismo y el *endpoint* a donde se conectarán, por lo que, al comienzo de cada uno de ellos, después de importar tanto la hoja de estilos, como la biblioteca de React y el cliente Web3 (previamente instalado con ***npm add web3***), deberemos declararlos como sigue:

```
import './App.css';
import React, { Component } from "react";

import Web3 from 'web3';

const nodeEndpoint = 'http://127.0.0.1:8545/';
const contractAddress = '0xEF2f61dCd7BF1Bf08Ac8f86952b33c499Ae5f10D';

const abi = [
  {
    "inputs": [],
    "name": "n_pasajero",
    "outputs": [
      {
        "internalType": "uint256",
        "name": "",
        "type": "uint256"
      }
    ],
    "stateMutability": "view",
    "type": "function",
    "constant": true
  },
  ...
...];
```

Una vez visto lo necesario para desarrollar nuestros componentes, veámoslos cada uno de ellos por separado, para comprender el funcionamiento de los mismos y cómo interactúan con el contrato inteligente SmartContractTravel.

El primero que abordaremos será *NumeroPasajero.js* el cual será el encargado de consultar a través de un botón cuantos pasajeros han sido registrados en nuestro contrato por parte de los centros sanitarios:

```
class NumeroPasajeros extends Component {
  constructor(props){
    super(props);
    this.state = {
      numeroPasajeros:0,
    }
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick(){
```

```

const web3 = new Web3(nodeEndpoint);

const smartContractTravel = new web3.eth.Contract(abi, contractAddress);

smartContractTravel.methods.n_pasajero().call().then(numeroPasajeros => this.setState(
({numeroPasajeros})));
}

render(){
  const {numeroPasajeros} = this.state;

  return(
    <div className="Container pasajeros">
      <span>Numero de Pasajeros: {numeroPasajeros}</span>
      <button onClick={this.handleClick}>Obtener Numero de pasajeros</button>
    </div>
  );
}
}
export default NumeroPasajeros;

```

A través del constructor del componente creamos su estado, el cual, en este caso, únicamente contendrá la variable, *numeroPasajeros*, donde se almacenará el número de pasajeros actualmente registrados en el sistema.

Además tenemos una función llamada *handleClick()* que será la encargada de obtener el número de pasajeros del contrato. Lo primero que hacemos es instanciar el cliente Web3 importado anteriormente, pasándole por parámetros el *endpoint* al que debe conectarse (donde está desplegado nuestro contrato). Esto nos permitirá realizar numerosas operaciones dentro de la red (por ejemplo, consultar el número de bloque actual). En este caso lo usaremos para instanciar nuestro contrato y como vemos, necesita del ABI y la dirección del mismo. Una vez que ya tenemos definido el objeto de nuestro contrato, ya podemos acceder a sus métodos. En nuestro caso accedemos al método de consulta *n_pasajero()* que a través de *call()* le especificamos que no existirá transacción alguna ya que solo deberá consultar datos. Una vez rescatado el dato que necesitábamos, actualizamos el estado de nuestro componente.

La función *render()* es la encargada de mostrar el dato del estado de nuestro componente en nuestra interfaz gráfica. Esta simplemente, almacena el dato en una constante y devuelve un contenedor con el número de pasajeros y un botón. Cuando pulsamos dicho botón se llama a la función anteriormente descrita para almacenar el valor devuelto por el contrato y se muestra actualizado en nuestra interfaz de la siguiente manera:

Numero de Pasajeros: 8

Obtener Numero de pasajeros

Seguidamente analizamos [RegistrarPasajero.js](#) que al contrario que el caso anterior, en este, se encargará de escribir en la blockchain los datos introducidos por los centros sanitarios para registrar sus pasajeros, llamando a la función del contrato [Registrar_datos_pasajero](#) que emitirá una transacción, por lo que deberá ser firmada (a través de Metamask) por la dirección de la cuenta correspondiente al centro sanitario.

```
class RegistrarPasajero extends Component {
  constructor(props){
    super(props);
    this.state = {
      nombreVariables: ['Nombre', 'Apellidos', 'Dni', 'Centro', 'NifCentro', 'DireccionCuenta'],
      datos: new Map(),
      enviando: false,
      transactionHash: null,
      numeroPasajeros: null
    }
    this._renderFields = this._renderFields.bind(this);
    this._renderButtons = this._renderButtons.bind(this);
    this._handleChange = this._handleChange.bind(this);
    this._handleClick = this._handleClick.bind(this);
    this._handleResult = this._handleResult.bind(this);
    this._handleReset = this._handleReset.bind(this);
  }
  ...
}
```

En este caso, el estado del componente define: un vector de variables *nombreVariables* (que serán los parámetros de nuestra función a llamar en el contrato), por la forma como está implementado el componente, simplemente añadiendo el nombre de una variable nueva a la lista, esta se renderizaría automáticamente como veremos a continuación; un diccionario de *datos*, cuyas claves serán nuestras variables y sus valores correspondientes; una variable *enviando*, que utilizamos para desactivar el botón que envía la información introducida mientras se está minando la transacción; una variable *transactionHash* donde se mostrará el hash de la transacción creada para cada operación y una variable *numeroPasajeros* que mostrará el identificador del pasajero una vez se hayan registrado sus datos.

```
...
_handleChange(e){
  let {datos} = this.state;

  let value = e.target.value ;
  let variable = e.target.attributes.placeholder.value;

  datos.set(variable,value);
}

_handleResult(e){
  let {datos} = this.state;
```

```

    let value = e.target.attributes.value.nodeValue;
    let variable = e.target.attributes.name.nodeValue;

    datos.set(variable, value);
  }
  ...

```

Con estas funciones manejamos los cambios en cada uno de los inputs. La primera de ellas maneja todos los inputs menos el correspondiente al resultado del test que es de tipo “radio-button”, por lo que funciona de manera ligeramente diferente y es por lo que se ha implementado la segunda función distinta a la primera. Con *e.target.value* rescatamos el valor del input y con el *e.target.attributes.placeholder* rescatamos la clave para dicho valor, para posteriormente almacenarlos en nuestro diccionarios de *datos*. Lo mismo en el segundo caso, aunque en vez de utilizar el *placeholder* de cada campo, usaremos directamente el *name* asignado a cada uno de los “radio-buttons”.

```

...
_handleClick(){
    const {datos} = this.state;

    if(datos.get('Nombre') == null || datos.get('Apellidos') == null || datos.get('Dni') == null || datos.get('Centro') == null || datos.get('NifCentro') == null || datos.get('DireccionCuenta') == null || datos.get('Resultado') == null){
        alert("Los campos no pueden estar vacíos");
    }else{
        this.setState({enviando: true, transactionHash:null});
        window.ethereum.enable().then(accounts => {
            const web3 = new Web3(window.ethereum);
            const smartContractTravel = new web3.eth.Contract(abi, contractAddress);

            smartContractTravel.methods.Registrar_datos_pasajero(datos.get('Nombre'), datos.get('Apellidos'), datos.get('Dni'), datos.get('Centro'), datos.get('NifCentro'), datos.get('DireccionCuenta'), datos.get('Resultado')).send({from:accounts[0]})
            .on('transactionHash', transactionHash => this.setState({transactionHash}))
            .on('receipt', _ => this.setState({enviando: false}));

            smartContractTravel.methods.n_pasajero().call().then(numeroPasajeros => this.setState({numeroPasajeros}));
        });
    }
}
...

```

Esta función es la encargada de registrar los datos del pasajero introducidos por el centro sanitario. Para ello, lo primero que hacemos es rescatar los *datos* del estado del componente y posteriormente comprobamos que ninguno de ellos tiene un valor nulo, es decir, nos aseguramos que se hayan introducido todos los datos antes de llamar a la función de nuestro contrato. En caso de que alguno de los campos no se haya introducido, nos aparecerá una alerta al pulsar el botón advirtiéndonos de que los campos no pueden estar vacíos. En caso contrario, establecemos el valor de la

variable del estado *enviando* a “verdadero” para así indicar que se ha iniciado la transacción (aún sin el hash de la misma, por lo que *transactionHash* lo establecemos a un valor nulo).

En la siguiente línea, lo que hacemos es solicitar acceso a la cartera del usuario, para así poder obtener las direcciones del usuario y poder operar con la cartera, en este caso, Metamask. Una vez hecho esto, instanciamos el cliente Web3 con *window.ethereum* que nos proporciona un proveedor para Metamask así conectarnos con la blockchain. También instanciamos nuestro contrato para poder acceder a sus funcionalidades.

Una vez hecho esto, procedemos a llamar a la función de nuestro contrato *Registrar_datos_pasajero* con los parámetros rescatados del estado de la aplicación. Y a diferencia del componente anterior, al ser una función que genera una transacción (escribe en la blockchain) debemos usar *send* en vez de *call*. En los parámetros del *send* especificamos la primera dirección de la lista de cuentas con *accounts[0]*, la cual siempre hará referencia a la dirección con la que se llama a esta función. Si la dirección introducida por parámetros por el centro sanitario no coincide con la que llama a la función, la transacción fallará al firmarla con Metamask, mostrando el error definido en el contrato. Ahora, definimos dos pasos para nuestra transacción:

- Primer paso: Cuando se envía la transacción, recuperamos el hash de la misma antes de ser minada (antes de que sea confirmada) y establecemos la variable del estado de la aplicación *transactionHash* al hash devuelto.
- Segundo paso: Una vez se ha minado la transacción anterior recuperamos el recibo de que se ha minado correctamente y establecemos nuestra variable de estado *enviando* a “falso” (su estado original), lo que servirá para volver a habilitar el botón que veremos posteriormente para registrar otro pasajero.

Por último, llamamos a la función descrita anteriormente para que nos devuelva el número de pasajeros registrados, que coincidirá con el identificador de cada pasajero, y lo almacenamos en la variable de estado *numeroPasajeros* definida para este fin. Esto lo hacemos, para poder mostrar el identificador del pasajero una vez que ha sido registrado en el sistema junto con el hash de la transacción.

```
...
_handleReset(){
  const {nombreVariables} = this.state;
  let datos = new Map();
  this.setState({datos});

  nombreVariables.forEach(value =>document.getElementById(value).value= "");
}
...
```

El cometido de esta función es únicamente limpiar el valor de los inputs y será llamada como veremos con un botón encargado de resetear los campos del registro.


```

...
_renderFields(){
    const {nombreVariables} = this.state;
    return (nombreVariables.map((value) =>
        <input className="input-
informacion" type='text' id={value} placeholder={value} onChange={this._handleChange}></input>
    ));
}

_renderResultado(){
    return(
        <div>
            <input type="radio" name="Resultado" value="Positivo" onChange={this._handle
Result}/>
            <label>Positivo</label>
            <input type="radio" name="Resultado" value="Negativo" onChange={this._handle
Result}/>
            <label>Negativo</label>
        </div>
    );
}

_renderButtons(){
    const{enviando} = this.state;
    return(
        <div className="Botones">
            <button onClick={this._handleClick} disabled={enviando}> Registrar pasajero<
/button>
            <button onClick={this._handleReset} disabled={enviando}>Resetea
datos</butt
on>
        </div>
    );
}
...

```

La primera función es la encargada de renderizar los campos donde se introducen los datos del pasajero a registrar. Es un bucle que devuelve un input por cada valor que hay dentro de *nombreVariables*. La función *map* itera por cada uno de los elementos de *nombreVariables* y *value* coge un valor de la lista cada vez, creando así un input por cada uno de los valores.

La segunda, simplemente renderiza los “radio-buttons” con los dos posibles resultados del test.

La última, es simplemente la encargada de renderizar los botones y como hemos visto antes hacemos depender la propiedad *disabled* de cada uno de ellos con el valor booleano de la variable de estado *enviando*, para así desactivar la interacción con esta función de la aplicación mientras se esté procesando una transacción.

```

...
render(){
  let {numeroPasajeros} = this.state;
  const {transactionHash} = this.state;
  if(numeroPasajeros > 0){
    numeroPasajeros++;
  }

  return(
    <div className="Container registrar-pasajero">
      <div className="fields">
        {this._renderFields()}
        {this._renderResultado()}
      </div>
      {this._renderButtons()}
      <span><br /><hr />Identificador del pasajero: {numeroPasajeros} </span>
      <span><hr />
        <label>TransactionHash: {transactionHash}</label>
      </span>
    </div>
  );
}
}
export default RegistrarPasajero;

```

Como última función de este componente, tenemos la función *render* principal, que haciendo uso de las anteriores crea una interfaz para esta funcionalidad parecida a la siguiente:

El siguiente componente a describir es *ModificarResultadoPasajero.js* que como su propio nombre indica será el encargado de llamar a la funcionalidad del contrato cuya tarea es modificar el resultado del test de un pasajero introduciendo su identificador y la dirección de la cuenta que realiza la transacción.

```
class ModificarResultadoPasajero extends Component {
  constructor(props){
    super(props);
    this.state = {
      nombreVariables: ['id', 'DireccionCuenta'],
      datos: new Map(),
      enviando: false,
      transactionHash: false
    }

    this._renderFields = this._renderFields.bind(this);
    this._renderButtons = this._renderButtons.bind(this);
    this._handleChange = this._handleChange.bind(this);
    this._handleClick = this._handleClick.bind(this);
    this._handleReset = this._handleReset.bind(this);
  }
  ...
}
```

En este caso, el estado del componente define: un vector de variables *nombreVariables* (que serán los parámetros de nuestra función a llamar en el contrato), por la forma como está implementado el componente, simplemente añadiendo el nombre de una variable nueva a la lista, esta se renderizaría automáticamente como vimos en el caso anterior, por lo que únicamente cambiamos los nombre de las variables para este componente; un diccionario de *datos*, que al igual que el caso anterior las claves serán nuestras variables y sus valores correspondientes; una variable *enviando*, que utilizamos para desactivar el botón que envía la información introducida en los campos mientras se está minando la transacción y una variable *transactionHash* donde se mostrará el hash de la transacción creada para cada operación.

Este componente tiene funciones comunes al anterior, como son: *_handleChange()*, *_handleReset()*, *_renderFields()* y *renderButtons()* por lo que no volveremos a explicarlas y nos centraremos en la función donde se define la verdadera funcionalidad del componente.

```
...
_handleClick(){
  const {datos} = this.state;

  if(datos.get('id') <= 0 || datos.get('DireccionCuenta') == null){
    alert("Los campos no pueden estar vacíos");
  }else{
    this.setState({enviando: true, transactionHash:null});
    window.ethereum.enable().then(accounts => {
      const web3 = new Web3(window.ethereum);
      const smartContractTravel = new web3.eth.Contract(abi, contractAddress);
    });
  }
}
```

```

        smartContractTravel.methods.Registrar_cambio_test(datos.get('id'), datos.get(
('DireccionCuenta')).send({from:accounts[0]})
        .on('transactionHash', transactionHash => this.setState({transactionHash
        .on('receipt', _ => this.setState({enviando: false})));
    });
}
}
...

```

En este caso, la función *handleClick()* funciona idénticamente a la del componente anterior, con la evidente diferencia que la función del contrato llamada por el mismo es *Registrar_cambio_test* y que en este caso no proporcionamos al final de la transacción el identificador del pasajero, dado que ya lo tiene y es uno de los parámetros. El resto del funcionamiento es exactamente el mismo que el caso anterior.

```

...
render(){
    const{transactionHash} = this.state;
    return(
        <div className="Container registrar-pasajero">
            <div className="fields">
                {this._renderFields()}
            </div>
            {this._renderButtons()}
            <span><br /><hr />
                <label>TransactionHash: {transactionHash}</label>
            </span>
        </div>
    );
}
}
export default ModificarResultadoPasajero;

```

Por último, como toda función *render* principal de los componentes, creará la interfaz de este haciendo uso de las anteriores, quedando similar a la siguiente imagen:

El siguiente componente a analizar es *InformaciónPasajero.js* que es el encargado de llamar a la función del contrato que, siendo la dirección de la aerolínea la que la llama, recupera los datos de los pasajeros introduciendo el identificador de cada uno de ellos.

```
class InformacionPasajero extends Component {
  constructor(props){
    super(props);
    this.state = {
      id:'',
      info_pasajero: []
    }

    this._renderDatos = this._renderDatos.bind(this);
    this._renderButtons = this._renderButtons.bind(this);
    this._handleChange =this._handleChange.bind(this);
    this._fetchInformacion = this._fetchInformacion.bind(this);
    this._clearData = this._clearData.bind(this);
  }
  ...
}
```

Como vemos, en el constructor que define el estado del componente tenemos una variable *id* que se utilizará para identificar al pasajero del que queremos obtener la información y un vector *info_pasajero* donde se almacenará esta información una vez recuperada de la blockchain.

También en este componente usamos funciones comunes a los anteriores, como son: *_handleChange()*, *_handleReset()*, *_renderFields()* y *renderButtons()* por lo que no volveremos a explicarlas y nos centraremos en la función donde se define la verdadera funcionalidad del componente.

```
...
_fetchInformacion(){
  const {id} = this.state;

  if(id > 0){
    let info_pasajero1 = [];
    const web3 = new Web3(nodeEndpoint);
    const smartContractTravel = new web3.eth.Contract(abi, contractAddress);

    window.ethereum.enable().then(accounts => {
      smartContractTravel.methods.Datos_Pasajero_Registrado(id).call({from:accounts[
0]}).then(function(result){

        for (let i=0; i < 7; i++){
          info_pasajero1.push(result[i]);
        }
        return info_pasajero1;

      }).then(result => this.setState({info_pasajero: info_pasajero1}));
    });
  }
}
```

```

    }else{
      alert("Los campos no pueden estar vacíos");
    }
  }
  ...

```

En este caso, en vez de llamar a la función *handleClick* la hemos denominado *fetchInformacion* y es la encargada de recuperar la información del usuario con el identificador introducido por la aerolínea. Para ello, lo primero que comprobamos es que el identificador sea mayor que 0, por lo que siempre nos llegará un valor válido del identificador. Nos declaramos un vector auxiliar donde almacenaremos la información devuelta por la función del contrato *Datos_pasajero_registrado* para posteriormente establecer el estado a este vector, si no lo hacemos así, una vez que recuperemos los datos de un pasajero, si introducimos otro identificador, la función nos seguirá mostrando la información del primero, es decir, no cambiaría correctamente de estado.

Como ya sabemos, para este tipo de funciones de consulta, debemos instanciar el cliente Web3 y el contrato como vemos, y en este caso, habilitaremos las cuentas de Metamask para poder comprobar cuál de ellas está llamando a la función, dado que siempre debería ser la aerolínea. Posteriormente, llamamos a la función del contrato, con el método *call* y especificando en sus parámetros la cuenta desde la que se llama a la función (aunque esta no genere transacción alguna). Gestionamos el resultado, copiándolo a nuestra variable auxiliar a través de un bucle hasta el número de parámetros devueltos por nuestra función del contrato, siete en este caso. Una vez tenemos nuestro vector auxiliar con todos los datos del pasajero introducido, establecemos el estado con esta información.

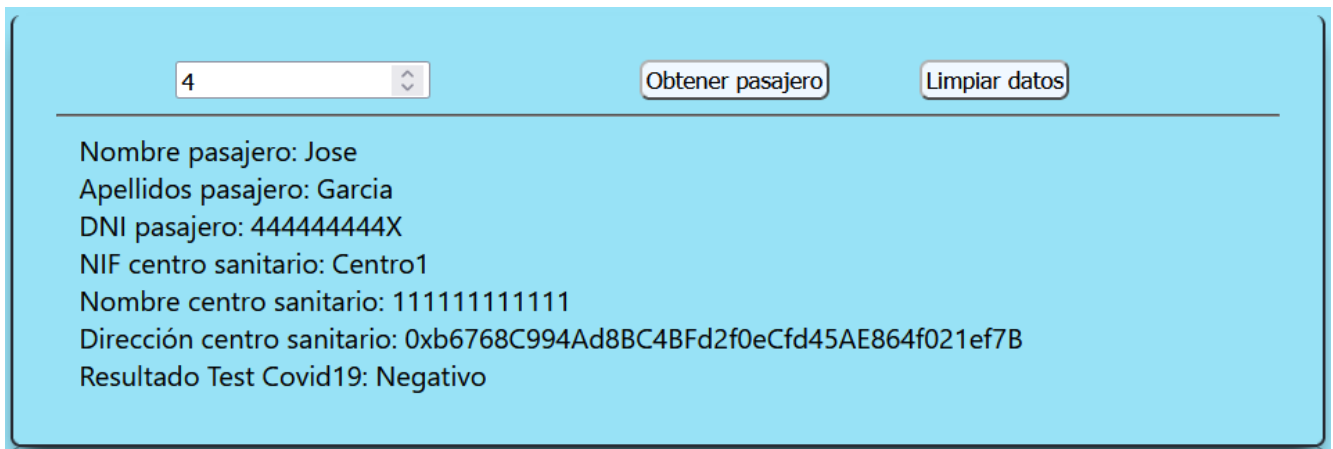
```

...
_renderDatos(){
  const {info_pasajero} = this.state; //aquí tenemos los datos cargados
  if (info_pasajero.length !== 0) {
    return (
      <div>
        <hr />
        <label> Nombre pasajero: {info_pasajero[0]} </label> <br />
        <label> Apellidos pasajero: {info_pasajero[1]} </label> <br />
        <label> DNI pasajero: {info_pasajero[2]} </label> <br />
        <label> NIF centro sanitario: {info_pasajero[3]} </label> <br />
        <label> Nombre centro sanitario: {info_pasajero[4]} </label> <br />
        <label> Dirección centro sanitario: {info_pasajero[5]} </label> <br />
        <label> Resultado Test Covid19: {info_pasajero[6]} </label> <br />
      </div>
    );
  }
}
...

```

Para renderizar estos datos, simplemente cargamos la variable de estado donde se encuentran los datos rescatados anteriormente y uno a uno vamos accediendo y mostrando los mismos. Esta función es llamada desde la función *render* principal del componente, prácticamente igual a las anteriores, en donde se renderiza el input del

identificador, los dos botones, tanto el que llama a la función *fetchInformacion* al clicar sobre él, como el que limpia la información del input, quedándonos una interfaz de la funcionalidad como la que sigue:



El penúltimo componente a analizar es *InformacionPasajeros.js* que como ya comentamos en capítulos anteriores (en concreto, en la página 82), es el componente encargado de crear y mostrar una lista con todos los pasajeros registrados en el contrato, por lo que el funcionamiento es parecido al anterior con algunos matices que veremos a continuación.

```
class InformacionPasajeros extends Component {
  constructor(props){
    super(props);
    this.state = {
      datos: new Map(),
      info_pasajero: [],
      numeroPasajeros: 0
    }
    this._renderDatos = this._renderDatos.bind(this);
    this._renderButtons = this._renderButtons.bind(this);
    this._fetchInformacion = this._fetchInformacion.bind(this);
    this._clearData = this._clearData.bind(this);

    const web3 = new Web3(nodeEndpoint);
    const smartContractTravel = new web3.eth.Contract(abi, contractAddress);
    smartContractTravel.methods.n_pasajero().call().then(numeroPasajeros => this.setState({numeroPasajeros}));
  }
  ...
}
```

En este caso, el constructor del componente define las siguientes variables de estado: *datos* que como siempre es un diccionario, pero en este caso almacenará la información a devolver por el componente, es decir, una lista de la información de todos los pasajeros, su clave será el identificador del pasajero y el valor del mismo es nuestra siguiente variable de estado, *info_pasajero* que es un vector que almacenará cada uno de los campos de información de un pasajero, por lo tanto, por cada clave del diccionario, accedemos a la información de ese pasajero. Por último, tenemos *numero_pasajeros* que, como vemos, cargamos su valor de la blockchain como ya

hemos visto, pero en este caso, esto lo hacemos en el propio constructor, llamando a la función del contrato `n_pasajero()` para que nos devuelva el número total de pasajeros que se encuentran registrados en dicho vuelo.

```
...
_fetchInformacion(){
  const {numeroPasajeros} = this.state;
  //si hay pasajeros:
  if(numeroPasajeros > 0){
    const web3 = new Web3(nodeEndpoint);
    const smartContractTravel = new web3.eth.Contract(abi, contractAddress);
    let datos = new Map();

    window.ethereum.enable().then(accounts => {
      for(let id=1; id<=numeroPasajeros; id++){
        smartContractTravel.methods.Datos_Pasajero_Registrado(id).call({from:accounts[0]}).then(function(result){
          let info_pasajero1 = [];

          for (let i=0; i < 7; i++){
            info_pasajero1.push(result[i]);
          }
          datos.set(id,info_pasajero1);

        }).then(result => this.setState({datos: datos}));
      }
    });
  }
}
...
```

Esta función, también parecida a la del componente anterior, comienza comprobando que existan pasajeros registrados, si es así, instanciamos el cliente Web3 y nuestro contrato, también nos declaramos un diccionario auxiliar donde almacenaremos los datos de todos los pasajeros. Habilitamos la conexión con nuestra cartera Metamask y por cada uno de los pasajeros existentes, consultamos a la función `Datos_pasajero_registrado` (con `call` y enviando la dirección que consulta para asegurarnos que solo lo hace la creadora del contrato). Luego, almacenamos como en el componente anterior la información de cada uno de los pasajeros, y posteriormente, la vamos añadiendo a nuestro diccionario de `datos`, con su clave correspondiente al identificador del pasajero y su valor con todos los datos del mismo para terminar estableciendo esta estructura de datos en el estado de nuestra aplicación.

```
...
_renderDatos(){
  const {datos} = this.state;
  let items=[];
  for(let [key, value] of datos){
    items.push(<span>Nombre pasajero: {value[0]}<br /></span>);
    items.push(<span>Apellidos pasajero: {value[1]}<br /></span>);
  }
}
```

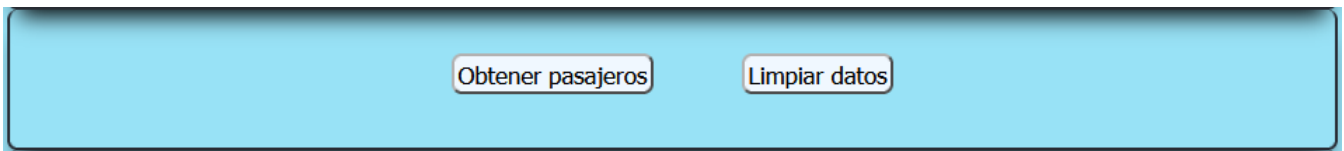


```

        items.push(<span>DNI pasajero: {value[2]}<br /></span>);
        items.push(<span>NIF centro sanitario: {value[3]}<br /></span>);
        items.push(<span>Nombre centro sanitario: {value[4]}<br /></span>);
        items.push(<span>Dirección centro sanitario: {value[5]}<br /></span>);
        items.push(<span>Resultado Test Covid19: {value[6]}<br /><hr /></span>);
    }
    return items;
}
...

```

Finalmente, para renderizar los datos se llamará desde la función *render* principal del componente a la función anterior, donde rescatamos del estado los datos de todos los pasajeros para posteriormente recorrerlo en un bucle específico para diccionarios, y así simplemente ir añadiendo a la lista llamada *items* los diferentes contenedores con los datos de cada pasajero (accedemos a ellos mediante su valor, que corresponde con el vector de los datos de cada uno de los pasajeros). En la siguiente sección, veremos su funcionamiento, pero por el momento podemos visualizar el componente renderizado a continuación:



Por último, comentaremos en esta sección el simple componente encargado de renderizar y dotar de funcionalidad al botón encargado de destruir el contrato llamando a la función del SmartContractTravel *destroy*. El estado de dicho componente únicamente se compone de las variables *enviando* y *transactionHash* explicadas en componentes anteriores y que dotará a este de la misma funcionalidad, inhabilitar el botón mientras la transacción se mine y mostrar el hash de la transacción una vez esta se haya validado.

```

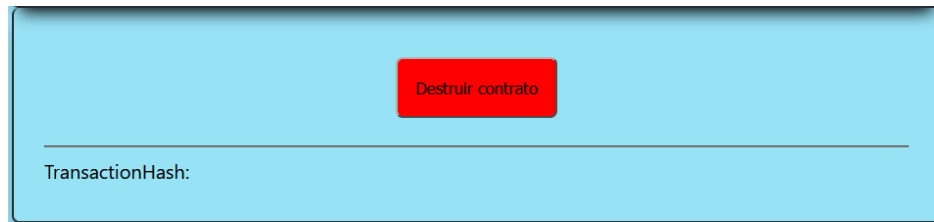
...
_handleDestroy(){
    this.setState({enviando: true, transactionHash:null});
    window.ethereum.enable().then(accounts => {
        const web3 = new Web3(window.ethereum);
        const smartContractTravel = new web3.eth.Contract(abi, contractAddress);

        smartContractTravel.methods.destroy().send({from:accounts[0]})
        .on('transactionHash', transactionHash => this.setState({transactionHash}))
        .on('receipt', _ => this.setState({enviando: false}));
    })
}
...

```

También como hemos comentado en componentes anteriores, esta función simplemente establece la variable de estado *enviando* a “verdadero” para que se inhabilite el botón mientras se comunica con la red. Y previa solicitud de acceso a la billetera de Metamask para firmar la transacción, creamos la instancia de nuestro cliente Web3 y del contrato para poder acceder a la función *destroy* del mismo, llamada con el método *send* especificando la cuenta que llama a dicha función.

Establecemos el hash de la transacción en nuestra variable de estado y posteriormente, una vez validada, volvemos a habilitar el botón.



Finalmente, vemos el archivo principal de la aplicación *App.js* encargado de renderizar todos los componentes de la misma anteriormente explicados, importando cada uno de ellos y llamando a cada componente en la función *render* principal de toda la aplicación, quedando la visualización de la página completa como podemos observar en la siguiente y última página de esta sección.

```
import React, { Component } from "react";
import NumeroPasajeros from './NumeroPasajeros';
import RegistrarPasajero from './RegistrarPasajero';
import InformacionPasajero from './InformacionPasajero';
import InformacionPasajeros from './InformacionPasajeros';
import ModificarResultadoPasajero from './ModificarResultadoPasajero';
import Destructor from './Destructor';
import './App.css';

class App extends Component {
  render(){
    return (
      <div className="App">
        <div>
          <h1>Smart Contract Travel:</h1>
        </div>
        <NumeroPasajeros></NumeroPasajeros>
        <h2>Centro sanitario:</h2>
        <RegistrarPasajero></RegistrarPasajero>
        <ModificarResultadoPasajero></ModificarResultadoPasajero>
        <h2>Aerolínea:</h2>
        <InformacionPasajero></InformacionPasajero>
        <InformacionPasajeros></InformacionPasajeros>
        <Destructor></Destructor>
      </div>
    );
  }
}

export default App;
```

Smart Contract Travel:

Numero de Pasajeros: 0

Obtener Numero de pasajeros

Centro sanitario:

Nombre

Apellidos

Dni

Centro

NifCentro

DireccionCuenta

☐ Positivo ☐ Negativo

Registrar pasajero

Resetear datos

Identificador del pasajero:

TransactionHash:

id

DireccionCuenta

Cambiar resultado

Resetear datos

TransactionHash:

Aerolínea:

Id del pasajero

Obtener pasajero

Limpiar datos

Obtener pasajeros

Limpiar datos

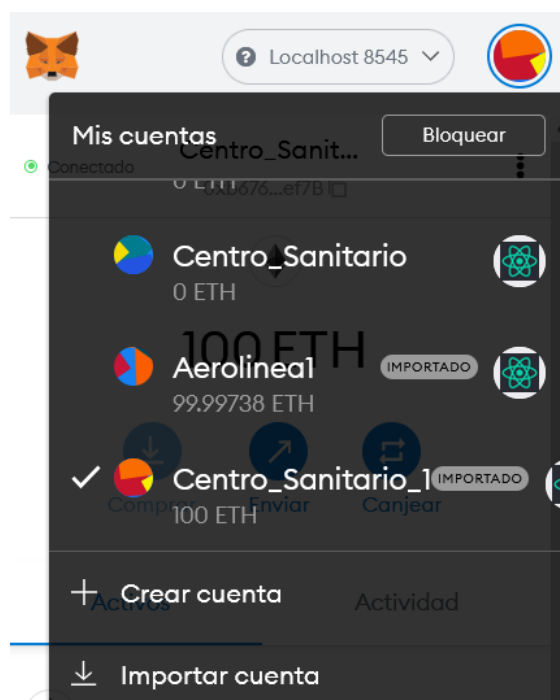
Destruir contrato

TransactionHash:

4.4. Ejemplo de uso DApp

Para probar nuestra aplicación debemos iniciar dos consolas de comandos. En la primera de ellas nos dirigimos al directorio de nuestro proyecto, ejecutamos Truffle y desplegamos nuestro contrato en la blockchain local como hemos visto anteriormente con los comandos **truffle develop** y **migrate**. En la segunda de ellas tendremos que dirigirnos al directorio *client* de nuestro proyecto e iniciar el servidor *node*, ejecutando el comando **npm start** nos abrirá nuestro navegador por defecto (Firefox en nuestro caso) en *localhost:3000* y vemos la página de nuestra aplicación mostrada en la página anterior.

Una vez conectados con Metamask a nuestra red local (*localhost:8545*), nos damos cuenta de que no tenemos ninguna de las cuentas con fondos que Truffle nos crea cuando ejecutamos **develop**, por lo que agregamos un par de ellas copiando la clave privada que se nos proporciona al iniciar Truffle. Abrimos Metamask y pulsando en el icono de nuestra cuenta, nos aparece la opción de importar cuentas, introducimos las claves privadas de las dos primeras, nos aseguramos que esté conectada y ya tendríamos ambas listas para usar. Supondremos la primera como la de la aerolínea, ya que es la que despliega el contrato y es por ello que ya tiene menos de 100 ETH en la imagen inferior ('0xB7bb99EB910095BB78c0AA65d3D56C08Ca912007'), y la segunda, supondremos que es la de un centro sanitario llamado "Centro_Sanitario1" con 100ETH ('0xb6768C994Ad8BC4BFd2f0eCfd45AE864f021ef7B').



Nota: Cada vez que reiniciemos Truffle deberemos de resetear nuestra cuenta de Metamask dado que, en caso contrario, al intentar volver a usarla nos devolverá un error relacionado con el nonce ("*Transaction Nonce too low or too high*"), ya que al reiniciar Truffle se reinicia también el estado de la cuenta en Metamask y este no encuentra las transacciones realizadas en la anterior sesión de Truffle lanzando el error comentado, y es por esto que debemos reiniciarla (evidentemente en una blockchain real esto no ocurrirá). Para reestablecer la cuenta de Metamask debemos pulsar en el círculo de nuestra cuenta, dirigirnos a *Configuración/Avanzado* y pulsamos en *Restablecer cuenta* para evitar este error.

Una vez sabido esto, proseguimos con el ejemplo de uso de nuestra aplicación descentralizada. Para ello, lo primero que haremos será registrar algunos pasajeros con la cuenta del centro sanitario importada anteriormente. Introducimos los datos del primer pasajero, en la billetera de Metamask seleccionamos la cuenta del centro sanitario y copiamos su dirección para pegarla en el campo correspondiente:

Smart Contract Travel:

Numero de Pasajeros: 0
 Obtener Numero de pasajeros

Centro sanitario:

☐ Positivo
 ☒ Negativo

Identificador del pasajero: 1

TransactionHash:

Extensión: (MetaMask) - MetaMask ...

Localhost 8545

Centro_Sanita...
 →
 0xEF2f...f10D

http://localhost:3000

INTERACCIÓN CON EL CONTRATO

0

DETALLES
 DATA

EDITAR

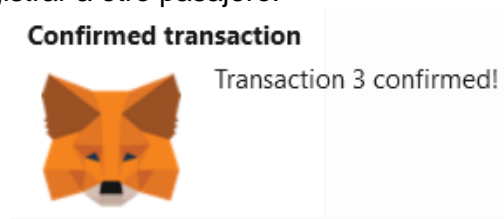
localhost suggested gas fee
 0.000651
 0.000651 ETH

Max fee: 0.000651 ETH

Total
 0.000651
 0.000651 ETH

Amount + gas fee
 Max amount: 0.000651 ETH

Una vez pulsemos en registrar los datos del pasajero se desactivarán los botones y deberemos firmar la transacción desde la ventana emergente de Metamask con la cuenta del Centro_Sanitario1. Una vez firmada podemos ver el identificador del pasajero y el hash de la transacción anterior, y ya tendremos los botones activados de nuevo para poder registrar a otro pasajero.



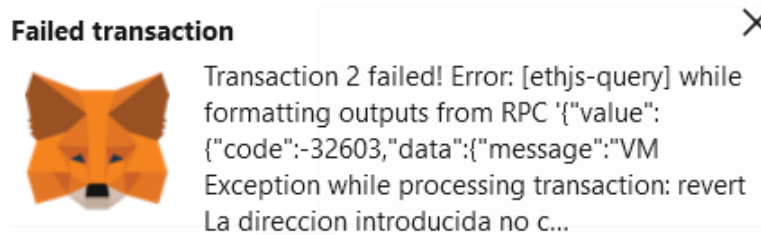
Identificador del pasajero: 1

TransactionHash:
 0x9607ab822ad4686414811175664e005aac49b33e65c17e0be8123a370a4705c9

Podemos comprobar que se haya añadido correctamente pulsando el botón que nos da el total de pasajeros inscritos como vemos a continuación:

Obtener Numero de pasajeros

En el caso de que la dirección que firma la transacción no coincida con la introducida en el campo de los datos del pasajero, en vez de obtener la confirmación de la transacción, obtendremos el siguiente error al firmarla y evidentemente no se añadirá el pasajero:



```
Transaction 2 failed! Error: [ethjs-query] while formatting outputs from RPC '{"value":{"cod":{"error":"revert","program_counter":3529,"return":"0x08c379a0000000000000000000000000 direccion introducida no coincide con el centro sanitario que envia la transaccion"},"stack'
```

Si alguno de los campos está vacío al pulsar el botón de registrar los datos del pasajero nos aparecerá la siguiente ventana emergente (este mensaje aparecerá en cualquiera de las funcionalidades cuando alguno de los campos esté vacío):

Aceptar

Una vez comprobado que se registran los pasajeros correctamente, proseguimos insertando varios pasajeros más para hacer este ejemplo de uso un poco más realista. En este ejemplo introducimos 5 pasajeros:

Obtener Numero de pasajeros

Es el momento de comprobar por parte de la aerolínea los datos de los pasajeros introducidos por los centros sanitarios. Para ello, debemos cambiar en Metamask la cuenta a la de la aerolínea, dado que es la única habilitada para consultar dichos datos a través de la aplicación.

Como ya sabemos, tenemos dos opciones para comprobar dichos datos por parte de la aerolínea, la primera de ellas será introduciendo el identificador de cada uno de los pasajeros para obtener sus datos, pasajero a pasajero:

Aerolínea:

Nombre pasajero: Juan
Apellidos pasajero: Garca Lopez
DNI pasajero: 11111111X
NIF centro sanitario: Centro Sanitario1
Nombre centro sanitario: 11111111
Dirección centro sanitario: 0xb6768C994Ad8BC4BFd2f0eCfd45AE864f021ef7B
Resultado Test Covid19: Negativo

La segunda opción es obtener la lista de los datos de todos los pasajeros registrados en el sistema como vemos a continuación, al pulsar el botón podremos visualizarla:

Nombre pasajero: Juan
Apellidos pasajero: Garca Lopez
DNI pasajero: 11111111X
NIF centro sanitario: Centro Sanitario1
Nombre centro sanitario: 11111111
Dirección centro sanitario: 0xb6768C994Ad8BC4BFd2f0eCfd45AE864f021ef7B
Resultado Test Covid19: Negativo

Nombre pasajero: Fernando
Apellidos pasajero: Ruiz Gutierrez
DNI pasajero: 22222222X
NIF centro sanitario: Centro Sanitario1
Nombre centro sanitario: 11111111
Dirección centro sanitario: 0xb6768C994Ad8BC4BFd2f0eCfd45AE864f021ef7B
Resultado Test Covid19: Negativo

Nombre pasajero: Marco
Apellidos pasajero: Romero Aguado
DNI pasajero: 33333333X
NIF centro sanitario: Centro Sanitario1
Nombre centro sanitario: 11111111
Dirección centro sanitario: 0xb6768C994Ad8BC4BFd2f0eCfd45AE864f021ef7B
Resultado Test Covid19: Negativo

Nombre pasajero: Francisco
Apellidos pasajero: Ortiz Terrn
DNI pasajero: 44444444X
NIF centro sanitario: Centro Sanitario1
Nombre centro sanitario: 11111111
Dirección centro sanitario: 0xb6768C994Ad8BC4BFd2f0eCfd45AE864f021ef7B
Resultado Test Covid19: Negativo

Nombre pasajero: Maria Amparo
Apellidos pasajero: Rodriguez Romn
DNI pasajero: 55555555X
NIF centro sanitario: Centro Sanitario1
Nombre centro sanitario: 11111111
Dirección centro sanitario: 0xb6768C994Ad8BC4BFd2f0eCfd45AE864f021ef7B
Resultado Test Covid19: Negativo

Como vemos, los datos están registrados correctamente y podemos consultarlos únicamente desde la cuenta de la aerolínea, dado que, si lo intentamos desde cualquier otra cuenta, en producción no nos mostrará absolutamente nada, aunque podemos ver el error que devuelve el contrato desde la consola del navegador como vemos a continuación:

```
Uncaught (in promise) Error: Returned error: VM Exception while processing transaction: revert Solo la aerolinea puede realizar esta funcion
ErrorResponse web3.min.js:15410
send web3.min.js:17813
onreadystatechange web3.min.js:33569
send web3.min.js:33558
```

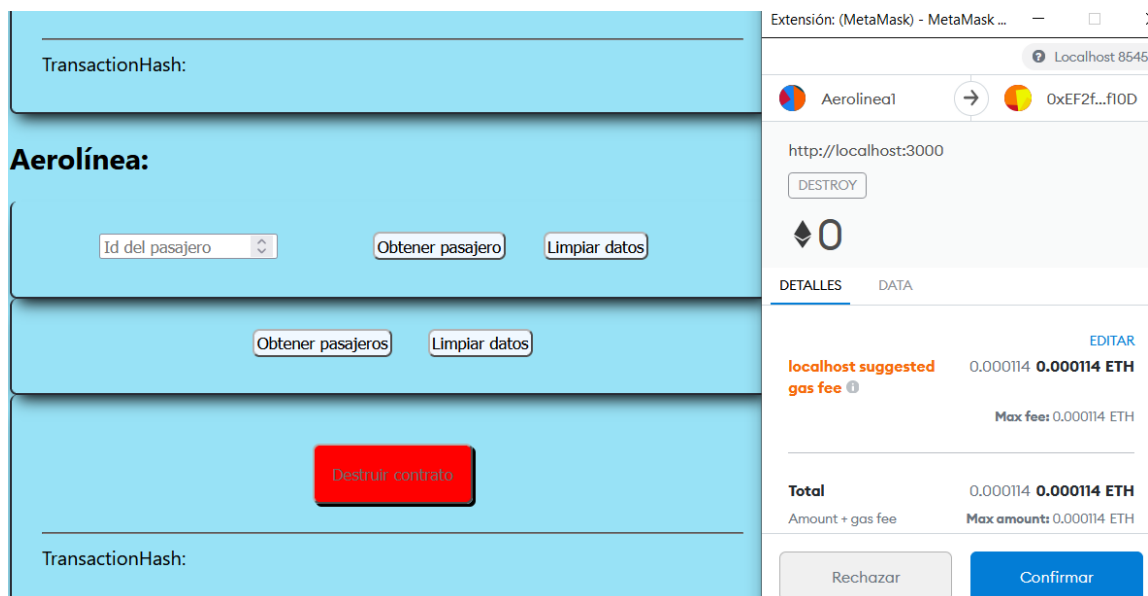
Una vez introducidos una serie de pasajeros con resultado del test negativo en nuestro sistema por parte de un centro sanitario y comprobada esta información por parte de la aerolínea, mostraremos cómo modificar el resultado del test de uno de estos pasajeros. Esto es simple, desde la cuenta del centro sanitario introduciremos el identificador del pasajero cuya prueba del Covid19 haya cambiado a positiva después de la realización de la primera e introduciremos también la dirección de la cuenta de este centro sanitario (dado que en caso de no coincidir la cuenta que firma la transacción generada con la dirección introducida, nos volverá a dar un error en la transacción como el anteriormente comentado y no se producirá cambio alguno en la información del pasajero cuyo identificador es el introducido):

Como vemos en este caso, también se desactivan ambos botones hasta que la transacción no haya sido firmada y minada.

Ahora es el momento de comprobar por parte de la aerolínea que el resultado del test de dicho pasajero ha cambiado a positivo. Para ello, elegimos la primera forma de consultar dicha información, es decir, introduciendo el identificador de dicho pasajero como vemos a continuación:

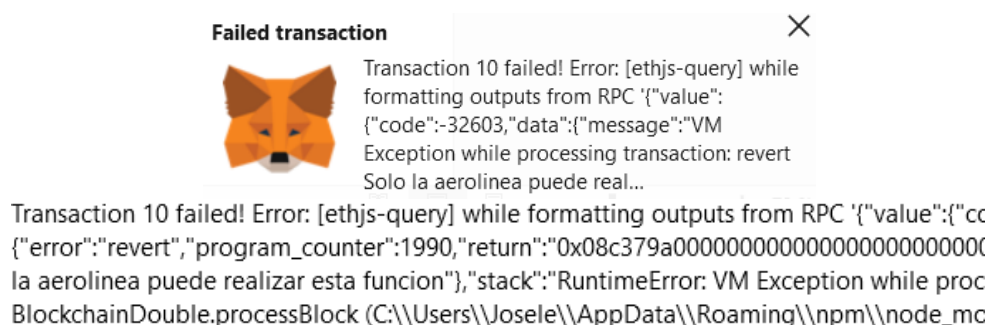
Como vemos el resultado del test del primer pasajero ha cambiado correctamente y cuando la aerolínea revise estos datos antes de la salida del vuelo correspondiente, deberá vetar la entrada al mismo a este pasajero. En esta revisión de datos, la aerolínea también debe asegurarse de que los centros sanitarios que han registrado a los pasajeros son válidos, comprobando su NIF sanitario y cerciorándose de que coincide con la dirección de la cuenta en su base de datos de centros sanitarios validados previamente para este fin.

Por último, una vez que se realice el vuelo, la aerolínea tiene la capacidad de destruir este contrato de la red (aunque todas las transacciones realizadas anteriormente queden almacenadas permanentemente en la misma). Para ello, seleccionamos la cuenta de la aerolínea, pulsamos en el botón rojo y firmamos la transacción con la cuenta de la aerolínea como vemos a continuación:



Una vez confirmada la transacción ya no será posible consultar este contrato y veremos el hash de la última transacción del mismo.

Si en vez de utilizar la cuenta de la aerolínea, tratamos de destruir este contrato con cualquier otra cuenta, al firmar la transacción con Metamask, se nos devolverá el error del contrato “Solo la aerolínea puede realizar esta función” y revertirá la transacción:



Como hemos visto el funcionamiento de la aplicación es bastante sencillo y funciona a la perfección, ya que los datos se escriben y se leen correctamente de la blockchain siempre y cuando se esté autorizado para ello.

4.5. Especificaciones del equipo de desarrollo

En esta sección se muestran tanto las especificaciones del equipo de desarrollo como las especificaciones mínimas, tanto hardware como software.

Hardware utilizado: Ordenador Personal MSI GE62VR 7RF

- Procesador: Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz (8 CPUs), ~2.8GHz
- Memoria RAM: 16 GB DDR4-2400
- Capacidad de almacenamiento: 3.5 TB
- Tarjeta gráfica: NVIDIA GeForce GTX 1060
- Memoria gráfica: 3 GB GDDR5
- Pantalla: 15.6"
- Conectividad: Killer Gb LAN 802.11 ac Wi-Fi + Bluetooth v4.2
- Sistema Operativo: Microsoft Windows 10 Home (64 bits)

Hardware mínimo requerido:

Para que la aplicación pueda desarrollarse correctamente es necesario contar con unas características básicas del equipo informático que permitan el adecuado procesamiento del código, así como una rápida ejecución de las herramientas usadas para la implementación y prueba de los contratos inteligentes, que no suponga una ralentización en el proceso de desarrollo. En este sentido sería necesario contar con, al menos:

- Procesador a 1.5GHz o superior
- Memoria RAM 2GB o superior
- Disco duro: 250 GB o superior
- Tarjeta gráfica integrada o dedicada

Los anteriores requisitos mínimos, si bien no son imprescindibles, si son recomendables a fin de conseguir una fluidez en el trabajo que permita un procesamiento rápido en la ejecución de las distintas herramientas usadas en el mismo.

Software:

- Windows 10 Home
- Truffle v5.4.3
- Solidity 0.7.6
- Node v14.17.4
- Web3.js v1.5.0
- Metamask: Extensión de Firefox
- Visual Studio Code

4.6. Planificación temporal

Para la ejecución de este trabajo ha sido necesario seguir una metodología acorde con el contenido del mismo para planificar y optimizar los tiempos para la consecución de los objetivos previamente definidos.

Podemos diferenciar tres fases bien definidas para la realización del mismo como son: el estudio e investigación de la tecnología Blockchain, el diseño y el desarrollo de la aplicación de nombre Smart Contract Travel desarrollada y la redacción de la memoria y documentación necesaria de todo el proyecto. En la siguiente tabla podemos ver con más detalle esta planificación temporal por semanas y una estimación del número de horas dedicadas a cada tarea:

Descripción de la tarea/fase	Tiempo empleado	Semanas
Planteamiento y propuesta de la temática del trabajo de fin de grado al tutor académico (estudio de viabilidad y propuesta del tema, reuniones previas con el tutor, ...).	30 horas	Semana1
Fase previa de establecimiento de objetivos y planteamiento de la estructura del proyecto.	35 horas	Semana2
Primera fase de investigación sobre la tecnología Blockchain y la cadena de bloques de Ethereum.	75 horas	Semana3 - Semana4
Creación de la documentación referente a la tecnología usada en el proyecto.	60 horas	Semana5 - Semana6
Segunda fase de investigación de las herramientas posteriormente utilizadas para el desarrollo del proyecto (investigación y aprendizaje sobre diferentes herramientas y su uso, investigación y aprendizaje de los lenguajes de programación usados, lecturas intensivas de documentación, cursos, tutoriales, etc.).	100 horas	Semana7 - Semana8 - Semana9
Tercera fase de diseño, implementación, testing y despliegue del contrato inteligente de nuestra aplicación Smart Contract Travel mediante diversas herramientas.	60 horas	Semana10 - Semana11
Cuarta fase de diseño e implementación del front-end de nuestra aplicación web Smart Contract Travel.	45 horas	Semana12
Quinta fase de demostración del funcionamiento real de la aplicación en un entorno simulado.	25 horas	Semana13
Creación de la documentación referente al desarrollo y pruebas de funcionamiento de la aplicación descentralizada Smart Contract Travel.	30 horas	Semana14
Total:	460 horas aprox.	3-4 meses aprox.

4.7. Líneas futuras

Se podría mejorar la implementación del Smart Contract Travel en varias vías, una de ellas sería que, en vez de crear un contrato para cada vuelo por parte de la aerolínea, lo que conlleva un coste alto de despliegue en la red de los mismos, se gestionase internamente dentro de este contrato, un número finito de vuelos; por ejemplo programados en un rango de fechas, para que así todos estos utilicen y registren a los pasajeros a través del mismo contrato y poder así reducir costes para la aerolínea, lo que se vería directamente reflejado en el precio de los billetes para cada pasajero, ya que estos costes de despliegue se dividirán entre todos los pasajeros con billete.

Otra vía de mejora respecto al contrato y futura aplicación se podría desarrollar tomando en consideración que la aparición de la pandemia COVID 19 es muy reciente en el tiempo, así como las grandes incógnitas que los científicos están tratando de resolver sobre su velocidad de transmisión y durabilidad. Por ello no es ilógico pensar que, a medida que las líneas de investigación vayan ampliándose y haciéndonos comprender más datos sobre la forma de transmisión, así como los periodos en los que el individuo resulta contagioso, podamos aplicar igualmente todos estos descubrimientos a los programas creados al efecto.

Por ejemplo, si se llegara a descubrir que un resultado positivo de la variante delta resulta contagioso durante un periodo de 15 días, resultaría posible que el programa permitiera que, una vez transcurrido ese tiempo, el pasajero pudiera ingresar en otro vuelo de la misma compañía sin necesidad de realización de una nueva prueba al efecto, pues sabiendo que su periodo infeccioso ha sido rebasado, podría utilizarse el primitivo Smart Contract Travel a efectos ulteriores.

Para ello la ciencia tiene que avanzar hasta extremos que permitan concluir con la suficiente certeza qué variante concreta infecta al individuo, tanto con un test de antígenos como con una P.C.R, así como durante cuánto tiempo esta variante resulta contagiosa a partir del resultado positivo, y durante cuando tiempo se asegura que, una vez superada la enfermedad, persiste la inmunidad en el individuo, con lo que también sería necesario fijar un tiempo máximo de utilización del primitivo Smart Contract Travel, para que no pudiera ser empleado una vez transcurrido ese periodo en que la inmunidad estaría garantizada según los avances científicos.

Por ejemplo si conocemos que hoy una persona arroja un resultado positivo en Covid-19 asociado a la variante “guatemalteca”, que esta variante “guatemalteca” tiene un periodo en el que resulta contagiosa de 10 días, y que según los avances científicos, transcurridos esos 10 días se garantiza una inmunidad del individuo al menos de 30 días, el contrato en el futuro podría gestionar todos estos nuevos datos para que la persona infectada hoy, sin necesidad de nuevas pruebas, pudiera tomar un vuelo de la misma compañía dentro de 11 días, cuando ya se sabe que no es contagiosa, y con un límite máximo de 30 días, en los que está garantizada su inmunidad y que por tanto no es susceptible de volver a reinfectarse ni de contagiar a terceros.

Respecto a la aplicación web podría mejorarse notoriamente creando un sistema de registros y login de centros sanitarios validados por la compañía aérea para así no tener que comprobar por parte de la aerolínea todas y cada una de las direcciones y NIF de los centros sanitarios de cada vuelo.

5. Conclusiones

En este proyecto hemos visto detalladamente la historia, características y funcionamiento de la tecnología Blockchain para así poder comprender correctamente los fundamentos técnicos que la sustentan. También hemos profundizado en el conocimiento de la plataforma Ethereum, su funcionamiento y algunas de las herramientas fundamentales de su ecosistema, para posteriormente desarrollar un prototipo de aplicación descentralizada que resuelve el problema de la falsificación de los test del Covid19 de realización previa a efectuar un vuelo comercial.

Como ya sabemos, esta tecnología tiene un potencial enorme de aplicación en multitud de sectores y servicios debido a su naturaleza descentralizada, la cual proporciona una seguridad sin precedentes y elimina la necesidad, en la mayoría de los casos, de una entidad central supervisora. Esto tiene un gran potencial en sectores tan amplios como pueden ser las finanzas, seguros, registro de la propiedad, el área de la salud, trazabilidad de productos, gestión empresarial e incluso en el arte digital, y muchos más.

Tras comprender el funcionamiento de la tecnología de la cadena de bloques y de una de las plataformas más importantes en el mundo blockchain como es Ethereum, con sus contratos inteligentes, se ha desarrollado satisfactoriamente el proyecto llamado Smart Contract Travel, el cual cumple el objetivo de asegurar la integridad de los pasajeros de un vuelo mediante la realización de las pruebas del Covid19 en los tres días previos a la salida del vuelo correspondiente, facilitando al pasajero estos trámites y poniendo en contacto a la aerolínea que oferta dicho vuelo con los centros sanitarios donde realizará dicha prueba cada pasajero del mismo, quedando la actuación de este pasajero relevada a un segundo plano.

En dicho proyecto se ha implementado de forma satisfactoria un contrato inteligente, con la lógica de nuestra aplicación, a través de diversas herramientas y el lenguaje de programación Solidity de Ethereum, así como una interfaz web desarrollada con React para interactuar con la misma, que será usada únicamente por la aerolínea y los centros sanitarios encargados de registrar los datos de los pasajeros de un vuelo.

También se ha desplegado el Smart Contract Travel de diferentes maneras y en distintas cadenas de bloques, tanto simuladas como redes de test de Ethereum, para mostrar el funcionamiento de las herramientas más relevantes para el desarrollo de contratos inteligentes. El despliegue no se ha realizado en la red principal de Ethereum debido al coste económico que conlleva, pero con lo aprendido hasta ahora, sabríamos como realizar dicho proceso y cómo analizar estos costes en la producción real de la aplicación.

Por último, cabe destacar que el principal objetivo de este trabajo era profundizar en el aprendizaje de esta tecnología debido a su actual impacto en el mercado laboral y a su enorme futuro, y tras comenzar el proyecto con unos conocimientos realmente básicos pero con mucho interés en su aprendizaje, después de mucho leer e investigar, se ha conseguido tener una visión mucho más amplia y clara acerca de este mundo y las diversas plataformas, tecnologías y herramientas que lo conforman, lo que ha desembocado en que este interés crezca aún más si cabe para así continuar formándome, aprendiendo y creciendo dentro de este sector tan llamativo y lleno de posibilidades.

6. Referencias bibliográficas

- [1] (2021, 30 julio) *Blockchain: definición y ámbitos de aplicación*. IONOS (consultado el 10/05/2021) <https://www.ionos.es/digitalguide/online-marketing/vender-en-internet/blockchain/>
- [2] Preukscha, A. (2017). *Blockchain: la revolución industrial de internet*. España: Grupo Planeta. (consultado el 15/04/2021)
- [3] Cointelegraph. *¿Qué es blockchain y cómo funciona la cadena de bloques?* (consultado el 10/05/2021) <https://es.cointelegraph.com/bitcoin-for-beginners/how-does-blockchain-work>
- [4] Molero, P. A. (2021, 25 enero). *Blockchain, así funciona la tecnología que va a cambiar el mundo y los negocios*. Emprendedores.es. (consultado el 03/05/2021) <https://www.emprendedores.es/gestion/blockchain-bitcoin-cambiar-mundo-negocios/>
- [5] Illescas, M. (2018, 19 enero). *Nos guste o no, las monedas digitales, las criptomonedas y toda la jerga asociada: Blockchain, Tokens, cadenas de bloques, peer . . .* Broker De Forex. (consultado el 18/05/2021) <https://www.brokerdeforex10.com/criptomonedas/que-son-los-tokens/>
- [6] Por: OroyFinanzas.com. (2015, 17 noviembre). *¿Qué es un token en Bitcoin?* OroyFinanzas.com. (consultado el 20/05/2021) <https://www.oryofinanzas.com/2014/10/que-token-bitcoin-criptomonedas/>
- [7] Por: OroyFinanzas.com. (2015, 26 noviembre). *¿Qué son los contratos inteligentes o smart contracts?* OroyFinanzas.com. (consultado el 20/05/2021) <https://www.oryofinanzas.com/2015/11/que-son-contratos-inteligentes-smart-contracts/>
- [8] Academy, B. (2020, 27 mayo). *Cuántos tipos de blockchain hay*. Bit2Me Academy. (consultado el 22/05/2021) <https://academy.bit2me.com/cuantos-tipos-de-blockchain-hay/>
- [9] A. (2018, 19 diciembre). *Blockchain Privada vs. Pública: ¿Cuál es la mayor diferencia?* NGL Latam Spain. (consultado el 22/05/2021) <https://nemespanol.io/blockchain-privada-vs-publica-cual-es-la-mayor-diferencia/>
- [10] Ramon Millan. *Características de las redes P2P* (consultado el 25/05/2021) http://www.arianamillan.com/libros/librodistribucionlibrosredesp2p/distribucionlibrosrede-sp2p_caracteristicasp2p.php
- [11] Academy, B. (2020, 20 octubre). *¿Qué son los nodos?* Binance Academy. (consultado el 19/05/2021) <https://academy.binance.com/es/articles/what-are-nodes>
- [12] Cash, M. (2019, 3 marzo). *Los nodos de blockchain y su influencia en la minería* | Mercury Cash Blog. Mercury Cash Blog | El Blog de Mercury Cash en idioma español. (consultado el 19/05/2021) <https://blog.mercury.cash/es/2019/01/25/los-nodos-de-blockchain-y-su-influencia-en-la-mineria/>
- [13] Maldonado, J. (2019, 11 julio). *¿Qué son Nodos y Supernodos?* Criptotendencias - Noticias de bitcoin, criptomonedas y blockchain. (consultado el 19/05/2021) <https://www.criptotendencias.com/base-de-conocimiento/que-son-nodos-y-supernodos/>

- [14] Blockchain, M. (2019, 6 febrero). *¿Qué es el P2P?* Máster en Blockchain, Smart Contracts y CriptoEconomía. (consultado el 21/05/2021) <https://masterethereum.com/que-es-p2p/>
- [15] Academy, B. (2021, 18 agosto). *¿Qué es un Algoritmo de Consenso?* Binance Academy. (consultado el 26/08/2021) <https://academy.binance.com/es/articles/what-is-a-blockchain-consensus-algorithm>
- [16] Gómez, B. (2019, 1 noviembre). *¿Quién manda en una red blockchain?* Paradigma. (consultado el 25/05/2021) <https://www.paradigmigital.com/techbiz/quien-manda-en-una-red-blockchain/>
- [17] Academy, B. (2020, 10 noviembre). *¿Qué es Prueba de trabajo / Proof of Work (PoW)?* Bit2Me Academy. (consultado 25/05/2021) <https://academy.bit2me.com/que-es-proof-of-work-pow/>
- [18] Academy, B. (2020, 10 noviembre). *¿Qué es Prueba de participación / Proof of Stake (PoS)?* Bit2Me Academy. (consultado el 29/05/2021) <https://academy.bit2me.com/que-es-proof-of-stake-pos/>
- [19] *Tipos de cuentas y transacciones*. (2018, 18 febrero). APRENDE BLOCKCHAIN. (consultado el 29/05/2021) <https://aprendeblockchain.wordpress.com/fundamentos-tecnicos-de-blockchain/cuentas-y-transacciones/>
- [20] (2021, 10 marzo). *Blockchain: bloques, transacciones, firmas digitales* | *CriptoNoticias*. CriptoNoticias - Noticias de Bitcoin, Ethereum y criptomonedas. (consultado el 29/05/2021) <https://www.cryptonoticias.com/criptopedia/blockchain-bloques-transacciones-firmas-digitales-hashes/>
- [21] Curran, B. (2018, 23 julio). *Comparing Bitcoin & Ethereum: UTXO vs Account Based Transaction Models*. Blockonomi. (consultado el 29/05/2021) <https://blockonomi.com/utxo-vs-account-based-transaction-models/>
- [22] *Comparing Bitcoin & Ethereum: UTXO vs Account Based Transaction Models*. (2018, 23 julio). Reddit. (consultado el 29/05/2021) https://www.reddit.com/r/ethereum/comments/915hmu/comparing_bitcoin_ethereum_utxo_vs_account_based/
- [23] *El modelo UTXO*. (s. f.). Horizen Academy. (consultado el 30/05/2021) <https://academy.horizen.io/es/technology/advanced/the-utxo-model/>
- [24] Rhodes, D. (2021, 20 agosto). *Cryptographic Hash Functions Explained: A Beginner's Guide*. Komodo Academy | En. (consultado el 02/06/2021) <https://komodoplatfrom.com/en/academy/cryptographic-hash-function/>
- [25] Vazquez, A. (2019, 24 septiembre). *El minado en Blockchain ¿Quiénes son y qué hacen los mineros?* | *CYSAE LEGAL*. CYSAE- Abogados Legaltech Madrid. (consultado el 02/06/2021) <https://www.cysae.com/el-minado-en-blockchain/>
- [26] Academy, B. (2021, 24 agosto). *¿Qué es la minería de criptomonedas?* Binance Academy. (consultado el 02/06/2021) <https://academy.binance.com/es/articles/what-is-cryptocurrency-mining>

- [27] Jimenez, D. (2019, 23 septiembre). *¿Cuántos algoritmos de consenso existen para las Blockchain?* Cointelegraph. (consultado el 29/05/2021)
<https://es.cointelegraph.com/news/cuantos-algoritmos-de-consenso-existen-para-las-blockchain>
- [28] Activity Report 2015-2016. (consultado el 02/06/2021)
<https://www.fidefundacion.es/attachment/729944/>
- [29] Preukschat, A. (2018, 29 enero). *Hyperledger: la Blockchain privada que todos tenemos que conocer.* elEconomista.es. (consultado el 02/06/2021)
<https://www.eleconomista.es/economia/noticias/8899454/01/18/Hyperledger-la-Blockchain-privada-que-todos-tenemos-que-conocer.html>
- [30] D. (2018, 21 marzo). *Aprende a distinguir PoW, PoS y Poi.* Bitcoin.es tu portal de información de criptomonedas. (consultado el 29/05/2021)
<https://bitcoin.es/mineria/pow-pos-y-poi/>
- [31] Jiménez, A. (2006, 26 agosto). *P versus NP. ¿Nunca lo entendiste?* Xataka Ciencia. (consultado el 02/06/2021)
<https://www.xatakaciencia.com/matematicas/p-versus-np-nunca-lo-entendiste>
- [32] *Conceptos de seguridad y criptografía en blockchain.* (2018, 18 febrero). APRENDE BLOCKCHAIN. (consultado el 04/06/2021)
<https://aprendeblockchain.wordpress.com/fundamentos-tecnicos-de-blockchain/fundamentos-basicos-de-criptografia-en-blockchain/>
- [33] Community, I. S. (2019, 18 septiembre). *Criptografía de curva elíptica - INT Chain Spanish Community.* Medium. (consultado el 04/06/2021)
<https://medium.com/articulos-de-la-comunidad/criptograf%C3%ADa-de-curva-el%C3%ADptica-99b8c8c1657c>
- [34] Academy, B. (2020, 10 noviembre). *¿Qué es la Tolerancia a Fallas Bizantinas (BFT)?* Bit2Me Academy. (consultado el 09/06/2021)
<https://academy.bit2me.com/que-es-tolerancia-fallas-bizantinas-bft/>
- [35] *Qué es el ataque del 51%: seguridad en Blockchain.* (2021, 31 marzo). Criptoblog. (consultado el 09/06/2021)
<https://criptoblog.tutellus.com/el-ataque-del-51-1>
- [36] *Realtime mining hardware profitability | ASIC Miner Value.* (s. f.). ASIC Miner Value. (consultado el 09/06/2021)
<https://www.asicminervalue.com/>
- [37] *NiceHash - Leading Cryptocurrency Platform for Mining and Trading.* (s. f.). nicehash. (consultado el 09/06/2021)
<https://www.nicehash.com/>
- [38] Shanaev, S. (2018, 24 noviembre). *Cryptocurrency Value and 51% Attacks: Evidence from Event Studies.* Papers. (consultado el 09/06/2021)
https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3290016
- [39] Munro, A. (2018, 14 mayo). *Cryptocurrency mining insider: All PoW coins have secret ASICs.* Finder.Com.Au. (consultado el 09/06/2021)
<https://www.finder.com.au/cryptocurrency-mining-insider-all-pow-coins-have-secret-asics>

- [40] Vorick, D. (2018, 21 junio). *The State of Cryptocurrency Mining - Sia and Skynet Blog*. Medium. (consultado el 11/06/2021) <https://blog.sia.tech/the-state-of-cryptocurrency-mining-538004a37f9b>
- [41] Academy, B. (2020, 9 noviembre). *¿Qué es el doble gasto?* Bit2Me Academy. (consultado el 11/06/2021) <https://academy.bit2me.com/que-es-doble-gasto/>
- [42] (2020, 4 agosto). *Ataques a blockchains, minería encubierta y robos a casas de cambio durante 2018*. CriptoNoticias - Noticias de Bitcoin, Ethereum y criptomonedas. (consultado el 11/06/2021) <https://www.criptonoticias.com/seguridad-bitcoin/ataques-blockchains-mineria-encubierta-robos-casas-cambio-2018/>
- [43] (2018, 29 julio). *Seis herramientas utilizadas por los hackers para robar*. (consultado el 11/06/2021) <https://es.cointelegraph.com/news/six-tools-used-by-hackers-to-steal-cryptocurrencyhow-to-protect-wallets>
- [44] Alcaide, J. C. (s. f.). *Blockchain para registro de la propiedad: países pioneros en su uso*. blog.enzymeadvisinggroup. (consultado el 14/06/2021) <https://blog.enzymeadvisinggroup.com/blockchain-registro-propiedad>
- [45] Blázquez, S. (2018, 16 julio). *Comillas y CEU estrenan blockchain universitario*. Blockchain Economía. (consultado el 14/06/2021) <https://www.blockchaineconomia.es/blockchain-universitario/>
- [46] Das, S. (2021, 4 marzo). *Japan Could Place Its Entire Property Registry on a Blockchain*. CCN.Com. (consultado el 14/06/2021) <https://www.ccn.com/japan-place-entire-property-registry-blockchain/>
- [47] Das, S. (2021, 4 marzo). *100%: Dubai Will Put Entire Land Registry on a Blockchain*. CCN.Com. (consultado el 14/06/2021) <https://www.ccn.com/100-dubai-put-entire-land-registry-blockchain/>
- [48] (2017, 27 noviembre). *Nunca pierdas tu llave privada: una odisea para recuperar \$30.000 en bitcoins*. CriptoNoticias - Noticias de Bitcoin, Ethereum y criptomonedas. (consultado el 11/06/2021) <https://www.criptonoticias.com/seguridad-bitcoin/nunca-pierdas-llave-privada-odisea-recuperar-30000-bitcoins/>
- [49] Pastor, J. (2018, 29 enero). *Coincheck sufre el mayor robo de criptodivisas de la historia, 535 millones de dólares en NEM*. Xataka. (consultado el 11/06/2021) <https://www.xataka.com/criptomonedas/coincheck-detiene-operaciones-de-retirada-de-criptodivisas-y-los-usuarios-estan-empezando-a-temer-lo-peor>
- [50] Academy, B. (2021, 3 mayo). *¿Qué es Ethereum Classic (ETC)?* Bit2Me Academy. (consultado el 11/06/2021) <https://academy.bit2me.com/que-es-ethereum-classic-etc-criptomonedas/>
- [51] Academy, B. (2021, 11 mayo). *¿Qué es una transacción coinbase?* Bit2Me Academy. (consultado el 22/06/2021) <https://academy.bit2me.com/que-es-coinbase-transaccion/>
- [52] Financiero, N. (2020, 2 febrero). *Blockchain, la tecnología que revolucionará las finanzas*. Nuevo Financiero. (consultado el 14/06/2021) <https://nuevofinanciero.com/blockchain-tecnologia-finanzas/>

- [53] (2018, 10 septiembre). *Radiografía de los usos futuros de la tecnología blockchain*. (consultado el 22/06/2021) <https://www.ticbeat.com/innovacion/usos-futuros-blockchain/>
- [54] (2018, 5 septiembre). *Cómo la tecnología blockchain cambiará el mundo para siempre*. (consultado el 22/06/2021) <https://www.ticbeat.com/innovacion/como-la-tecnologia-blockchain-cambiara-el-mundo-para-siempre/>
- [55] (2021, 9 marzo). *Qué es Bitcoin (BTC) | CriptoNoticias - blockchains y criptomonedas*. CriptoNoticias - Noticias de Bitcoin, Ethereum y criptomonedas. (consultado el 22/06/2021) <https://www.cryptonoticias.com/criptopedia/que-es-bitcoin-btc/>
- [56] Mitra, R. (2019, 12 junio). *¿Qué es Ethereum? ¡La guía más completa que existe!* Blockgeeks. (consultado el 27/06/2021) <https://blockgeeks.com/guides/es/que-es-ethereum/>
- [57] Ecosystem, A. B. (2018, 19 septiembre). *Paso a paso: así se crea un nodo regular en Alastria*. Medium. (consultado el 27/06/2021) <https://alastria-es.medium.com/paso-a-paso-as%C3%AD-se-crea-un-nodo-regular-en-alastria-e9ef9a47b07f>
- [58] Garatu, G. (2018, 12 diciembre). *¿Qué tipos de Blockchain o Cadenas de Bloques existen?* Grupo Garatu. (consultado el 27/06/2021) <https://grupogaratu.com/cuales-son-los-diferentes-tipos-de-blockchains-o-cadena-de-bloques/>
- [59] (2019, 20 octubre). *Qué son los tokens y cómo se diferencian de las criptomonedas*. CriptoNoticias - Noticias de Bitcoin, Ethereum y criptomonedas. (consultado el 27/06/2021) <https://www.cryptonoticias.com/mercados/que-son-tokens-como-diferencian-criptomonedas/>
- [60] Cecilia, P (2018, 4 septiembre). *Blockchain: qué es, cómo funciona y cómo se está usando en el mercado*. (consultado el 28/06/2021) <https://www.welivesecurity.com/la-es/2018/09/04/blockchain-que-es-como-functiona-y-como-se-esta-usando-en-el-mercado/>
- [61] Python, R. (2021, 1 julio). *Desarrollar una aplicación «blockchain» desde cero en Python*. Recursos Python. (consultado el 01/07/2021) <https://recursospython.com/guias-y-manuales/aplicacion-blockchain-desde-cero/>
- [62] *Cost of a 51% Attack for Different Cryptocurrencies | Crypto51*. (s. f.). PoW 51% Attack Cost. (consultado el 01/07/2021) <https://www.crypto51.app>
- [63] *La Historia de Ethereum | Plus500*. (s. f.). Qué es Ether y cómo funciona. (consultado el 02/07/2021) <https://www.plus500.es/Instruments/ETHUSD/The-History-of-Ethereum%7E4>
- [64] *¿Qué es Ethereum y cómo funciona?* (s. f.). IG. (consultado el 02/07/2021) <https://www.ig.com/es/ethereum-trading/que-es-ether-y-como-functiona>
- [65] Academy, B. (2020, 9 noviembre). *¿Qué es la Ethereum Virtual Machine (EVM)?* Bit2Me Academy. (consultado el 15/07/2021) <https://academy.bit2me.com/que-es-ethereum-virtual-machine-evm/>

- [66] Academy, B. (2020, 9 noviembre). *¿Qué son las DApps?* Bit2Me Academy. (consultado el 18/07/2021) <https://academy.bit2me.com/que-son-las-dapps/>
- [67] Academy, B. (2021, 14 junio). *¿Qué es MetaMask? La forma más fácil de usar DApps.* Bit2Me Academy. (consultado el 26/07/2021) <https://academy.bit2me.com/que-es-metamask-la-forma-mas-facil-de-usar-dapps/>
- [68] Academy, B. (2021, 20 julio). *¿Qué es Ethereum (ETH)?* Bit2Me Academy. (consultado el 27/07/2021) <https://academy.bit2me.com/que-es-ethereum-eth-criptomonedas/>
- [69] Blanco, D. (2021, 14 julio). *Creando Smart Contracts en Ethereum.* Paradigma. (consultado el 30/07/2021) <https://www.paradigmigital.com/dev/creando-smart-contracts-en-ethereum/>
- [70] *Desarrollo con Truffle II.* (2018, 17 abril). APRENDE BLOCKCHAIN. (consultado el 30/07/2021) <https://aprendeblockchain.wordpress.com/desarrollo-en-ethereum/desarrollo-con-truffle-ii/>
- [71] E. (s. f.). *SmartContractTravel* / *0xd2fE7127C3D6756AD2d69648166Cb904a4aBA2ea*. Ethereum (ETH) Blockchain Explorer. (consultado el 09/08/2021) <https://rinkeby.etherscan.io/address/0xd2fE7127C3D6756AD2d69648166Cb904a4aBA2ea>
- [72] Ethereum. (s. f.). *Development Networks.* Ethereum.Org. (consultado el 30/07/2021) <https://ethereum.org/en/developers/docs/development-networks/>
- [73] Ethereum. (s. f.). *Intro to Ethereum.* Ethereum.Org. (consultado el 30/07/2021) <https://ethereum.org/en/developers/docs/intro-to-ethereum/>
- [74] Ethereum. (s. f.). *Networks.* Ethereum.Org. (consultado el 30/07/2021) <https://ethereum.org/en/developers/docs/networks/>
- [75] Ethereum. (s. f.). *Non-fungible tokens (NFT).* Ethereum.Org. (consultado el 30/07/2021) <https://ethereum.org/en/nft/>
- [76] Ethereum. (s. f.). *Transactions.* Ethereum.Org. (consultado el 30/07/2021) <https://ethereum.org/en/developers/docs/transactions/>
- [77] Maldonado, J. (2020, 28 mayo). *¿Qué es la Ethereum Virtual Machine (EVM)? La máquina virtual de Ethereum.* Cointelegraph. (consultado el 02/08/2021) <https://es.cointelegraph.com/explained/what-is-the-ethereum-virtual-machine-evm-the-ethereum-virtual-machine>
- [78] Maldonado, J. (2021, 7 enero). *Truffle, la mayor herramienta de desarrollo para Ethereum.* Cointelegraph. (consultado el 02/08/2021) <https://es.cointelegraph.com/explained/truffle-the-biggest-development-tool-for-ethereum>
- [79] *The official ethereum wallet stopped working. The foundation should make it work.* (2020, 20 febrero). Reddit. (consultado el 02/08/2021) https://www.reddit.com/r/ethereum/comments/f6via3/the_official_ethereum_wallet_stopped_working_the/

- [80] Pastor, J. (2021, 18 marzo). *Qué son los NFT, los activos digitales que están transformando el coleccionismo de arte y bienes digitales*. (consultado el 03/08/2021) <https://www.xataka.com/criptomonedas/que-nft-activos-digitales-que-estan-transformando-coleccionismo-arte-bienes-tangibles-e-intangibles>
- [81] Piedra, U. L. (s. f.). *Cómo poner en marcha tu primer Smart Contract en 3 pasos*. (consultado el 01/08/2021) <https://www.izertis.com/es/-/blog/como-poner-en-marcha-tu-primer-smart-contract-en-3-pasos>
- [82] *¿Qué es Truffle en Ethereum? Una guía básica*. (2020, 30 junio). (consultado el 02/08/2021) <https://es.beincrypto.com/aprende/truffle-ethereum-eth-guia-basica/>
- [83] Quintiliano, R. (2021, 9 abril). *¿Qué es el algoritmo Casper?* (consultado el 29/07/2021) <https://blog.bitnovo.com/que-es-el-algoritmo-casper-ethereum2/>
- [84] Solidity — *documentación de Solidity*. (s. f.). (consultado el 01/08/2021) <https://solidity-es.readthedocs.io/es/latest/>
- [85] *Sweet Tools for Smart Contracts*. (s. f.). (consultado el 20/07/2021) <https://www.trufflesuite.com/>
- [86] M. (2021, 10 marzo). *¿Qué son los tokens no fungibles NFT? Guía completa 2021. Mundo NFT*. (consultado el 27/07/2021) <https://mundonft.com/no-fungibles-nft/>
- [87] Vyper — *Vyper documentation*. (s. f.). (consultado el 01/08/2021) <https://vyper.readthedocs.io/en/latest/>
- [88] Welcome to Remix's documentation! — *Remix - Ethereum IDE 1 documentation*. (s. f.). (consultado el 01/08/2021) <https://remix-ide.readthedocs.io/en/latest/#>
- [89] (2021, 16 abril). *Fusión Ethereum 1.0 y 2.0: la fecha para la unión mínima viable aún no tiene consenso*. (consultado el 29/07/2021) <https://www.criptonoticias.com/tecnologia/fusion-ethereum-10-20-fecha-union-minima-no-consenso/>

