uc3m | Universidad Carlos III de Madrid

Grado en Ingeniería en Tecnologías de la Telecomunicación.
2018-2019

Trabajo Fin de Grado

"Estudio sobre *Smart Contracts* en *Ethereum*"

Alvaro Roco Salas

Tutor/es

Daniel Díaz Sánchez

Leganés, febrero de 2019

RESUMEN

Este trabajo fin de grado consiste en la exploración y el estudio de una tecnología con el objetivo de comprender su funcionamiento y confeccionar una guía que permita a un usuario nivel medio, adentrarse en ella y dar los primeros pasos con las ideas claras.

La tecnología en cuestión es *blockchain* y el elemento central hacia donde se orientan todos los conceptos son los *smart contracts* de la plataforma Ethereum.

Se partirá desde ámbito general, explicando el funcionamiento y las características más relevantes de *blockchain*, ya que es esta la tecnología sobre la que se asientan Ethereum y sus *smart contracts*.

Se mencionarán las características esenciales del protocolo que introdujo la tecnología *blockchain* por su relevancia y porque es el que se lleva toda la fama, Bitcoin.

Por último, se abordará la plataforma Ethereum para explicar sus características, el funcionamiento de su máquina virtual, se profundizará en los *smart contracts* y se mostrarán unos conceptos básicos sobre el lenguaje de programación más utilizado para diseñarlos, *Solidity*.

Palabras clave: *smart contracts*, Ethereum, Bitcoin, blockchain, *Solidity*, Vitalik Buterin.

ÍNDICE DE CONTENIDOS

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN.

1.1 Motivación.

A raíz del interés generado, en el que se profundizará en el impacto socieconómico, y del potencial que ofrece, *Blockchain* (cadena de bloques) puede ser la tecnología más disruptiva desde la creación de internet y creo que puede marcar un nuevo paradigma digital.

Internet comenzó a desarrollarse en el ámbito militar durante la década de los 60 para poder establecer comunicaciones internas en el ejército de los EEUU. Hasta la década de los 90 no llegó a los hogares, pero rápidamente se convirtió en indispensable para nuestro día a día. El internet de la información, transformó nuestras vidas y supuso un nuevo nicho de mercado empresarial, del que surgieron algunos gigantes como Google o Amazon.

Pese a que internet pueda parecer una maravilla por completo, tiene algunos inconvenientes, como la dependencia de un tercero para almacenar nuestros datos, la nula transparencia de cómo se utilizan esos datos o la privacidad de los usuarios. Todos estos problemas comparten un aspecto entre ellos, internet es una tecnología cuya autoridad está centralizada, bien sea una persona o una empresa. Esa autoridad es en la que tenemos que confiar cuando usamos internet.

Blockchain abre una nueva era de internet, la del internet del valor. Esta tecnología permite compartir valor digitalmente sin necesidad de una entidad de confianza que centralice la autoridad de la red.

Dentro de las diferentes redes y protocolos blockchain, el principal protagonista es el archiconocido Bitcoin. Sin embargo, existen muchas otras alternativas. Entre ellas, destaca Ethereum, que pretende ser más que una solución *per se*, una infraestructura en la que multitud de propuestas puedan desplegarse, bajo el mismo soporte.

Debido a que aún faltan muchos años para que la tecnología sea completamente madura y se haya extendido al grueso de los usuarios tecnológicos, son pocos los recursos que permitan realizar una iniciación a nivel medio en este nuevo mundo.

Existen multitud de publicaciones que hablan sobre blockchain o sobre algunos de los protocolos que operan basados en blockchain. Sin embargo, la gran mayoría se remiten únicamente a la idea del concepto, lo que no facilita una comprensión completa del tema que se está tratando. El resto de publicaciones suponen, o el documento con las especificaciones técnicas del tema en cuestión, o una explicación del problema/solución en el ámbito de la programación.

1.2 Objetivos.

Este proyecto pretende ser, por consiguiente, una guía introductoria a la tecnología *blockchain* para aquellos usuarios que no dispongan de conocimientos previos. En concreto, el documento se va a centrar en los *smart contracts* de la plataforma Ethereum.

Para ello, se tratará de ofrecer tanto la idea de los conceptos necesarios para entender su funcionamiento como, en algunos casos, profundizar en ellos para resolver las dudas que le surgía a un servidor cuando trataba de indagar y explorar la tecnología y siempre encontraba la misma información.

Se abordarán inicialmente los conceptos más generales y, poco a poco, se irá poniendo el foco en los contratos inteligentes y como se ejecutan. De la misma forma, se ofrecerá alguna pincelada sobre el lenguaje de programación más utilizado en la implementación de *smart contracts*, *Solidity*.

Cabe destacar que, en ningún caso se pretende realizar una guía formativa de expertos en *smart contracts* de Ethereum por dos motivos:

1. La tecnología está en pleno desarrollo y constante cambio, por lo que se considera más importante comprender y sentar unas bases de conocimiento relacionadas con *blockchain* para facilitar la adopción de conocimientos en el futuro.
2. Los *smart contracts* se pueden programar en, actualmente, 7 lenguajes de programación, cada uno de ellos basado en uno ya existente pero todos ellos creados por y para *blockchain*. Tanto los conceptos *per se*, como cada uno de los lenguajes implican material suficiente para realizar un TFG por separado, por lo que unirlos supondría un documento sumamente extenso.

1.3 Marco regulatorio.

Como se comentará más adelante, *blockchain* es una tecnología muy reciente pero que plantea un nuevo paradigma digital que puede modificar la actividad diaria de las personas, al igual que lo hizo internet en su momento.

Se puede acudir a cualquier aspecto que deba ser regulado para comprobar que cada gobierno aplica un marco legal diferente y, en algunas ocasiones, totalmente contrario. La tecnología *blockchain* no iba a ser la excepción y, en efecto, cada gobierno aplica una regulación diferente.

En el extremo de los gobiernos que se oponen por completo a la adopción de la tecnología encontramos a Marruecos, Nepal o Pakistán, que prohíben toda actividad

relacionada con criptomonedas. De forma parecida, tanto Qatar como Bahréin limitan la actividad con criptomonedas dentro de sus fronteras, no así la actividad de sus ciudadanos en el extranjero.

En algunos países como China, no se prohíbe su uso pero si se tratan de poner trabas. En el caso concreto de China, ninguna entidad financiera puede dar soporte a transacciones, ni siquiera extranjeras, que estén relacionadas con criptomonedas. También se han emitido comunicados en los que se manifiestan los inconvenientes de trabajar con criptomonedas (no están soportadas por ningún gobierno, en caso de pérdida no se admiten denuncias…).

Sin embargo, también existen gobiernos que tratan de facilitar la expansión y adopción de la tecnología. En Canadá, por ejemplo, se acepta el uso de criptomonedas y se permite efectuar pagos con ellas. Aunque no se considere moneda de curso legal, debe incluirse en la declaración de la renta y se aplican tasas impositivas al minado (la acción de crear y añadir nuevos bloques a la cadena) con fines lucrativos.

De forma parecida, en Italia, las ganancias o pérdidas derivadas de transacciones con criptomonedas también deben incluirse en los impuestos relacionados con la renta. En la actualidad, el gobierno italiano está en proceso de aprobar una enmienda que otorga validez legal a la actividad que se produce en la *blockchain*, según la *Proposta di modifica n. 8.0.3 al DDL n. 989*[1].

En España, al igual que en el resto de países, el marco regulatorio es poco concreto. La Comisión Nacional de Valores de España y el Banco de España expresaron que ni las criptomonedas, ni la actividad relacionada con las mismas, están respaldadas por ninguna de las garantías que se aplican en el marco regulatorio del sistema financiero. **(1)**

Respecto a la fiscalidad relacionada con los criptoactivos, se aplican actualmente en España las siguientes tasas impositivas: **(27)**

- Modelo 720. Declaración Informativa. Declaración sobre bienes y derechos situados en el extranjero: Se deben declarar en este modelo todas las criptomonedas que se dispongan en *Exchanges* (plataformas que posibilitan el intercambio de criptoactivos o criptomonedas por dinero fiduciario o viceversa), ya que estos suelen tener su sede fiscal en el extranjero.

[1] La propuesta de enmienda fue publicada el 14 de enero de 2019 por el *Senato della Repubblica* y puede consultarse en: http://www.senato.it/japp/bgt/showdoc/frame.jsp?tipodoc=Emendc&leg=18&id=1096791&idoggetto=1095835 **[consulta 19/02/2019]**

- Impuesto de patrimonio: Sólo afecta a aquellos inversores cuyas inversiones superan un determinado patrimonio neto. Este mínimo depende directamente de la regulación vigente en cada comunidad. Este impuesto graba las posesiones, por lo que la fluctuación del precio de las criptomonedas afecta directamente.
- IRPF: Directamente relacionado con ganancias y pérdidas patrimoniales. Aplica también a aquellos intercambios entre diferentes criptos o *tokens* sin pasar por dinero fiduciario.
- IVA: La actividad relacionada con criptomonedas está exenta de esta tasa impositiva.

1.4 Impacto socioeconómico.

Una de las motivaciones para abordar este trabajo fin de grado residía en la curiosidad que le surgía a un servidor una tecnología que comenzaba a cobrar fuerza en las publicaciones relacionadas con el mundo tecnológico y de la cuál todo parecía muy abstracto y existía poca documentación que te introdujera en ella.

A continuación se van a exponer diferentes comparativas y gráficas, obtenidas con la herramienta "Google Trends"[2], que muestran el número de búsquedas de algunas palabras clave. Con esto se quiere mostrar el creciente interés y cómo, en efecto, está atrayendo cada día a más gente.

En primer lugar se va a mostrar una comparativa del interés que generan en todo el mundo y España los términos "bitcoin", "blockchain", "ethereum" y "Real Madrid Club de Fútbol" desde el 31 de agosto de 2008 (fecha de publicación del *whitepaper* de Bitcoin). El motivo de incluir el último concepto es para que la gráfica contenga un elemento de gran importancia social, como en este caso uno de los mejores equipos de fútbol del mundo.

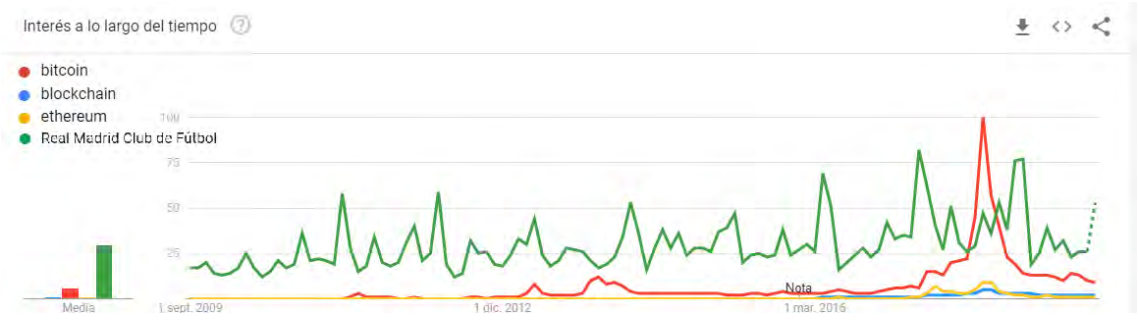


Fig.1. Comparativa mundial - bitcoin, blockchain, ethereum y Real Madrid. 31/8/09 – 20/02/19

[2] La herramienta está disponible en la web: https://trends.google.es/trends/

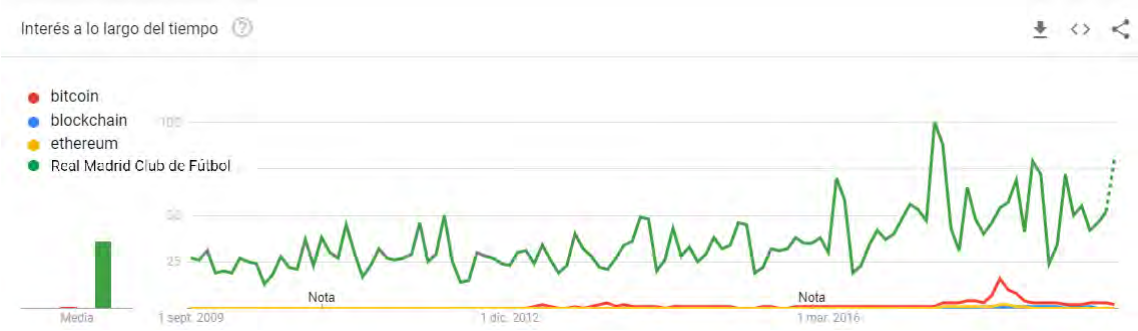


Fig. 2. Comparativa España - bitcoin, blockchain, ethereum y Real Madrid. 31/8/09 – 20/02/19

Se puede comprobar como Bitcoin es quien se lleva la fama en el asunto, quizás por ser la pionera o quizás por la especulación que le rodea. A nivel mundial ha superado en alguna ocasión al interés que despertaba el equipo de fútbol. Coincide de hecho con un periodo en el que el precio de la criptomoneda de Bitcoin alcanzaba su máximo histórico.

A continuación se va a sustituir el espectro de búsqueda, desde enero de 2014 hasta la actualidad (ya que el *whitepaper* de *Ethereum* se publicó en diciembre de 2013), y el elemento comparativo por "Java". En este caso se ha utilizado un lenguaje de programación maduro, asentado y utilizado por muchas empresas para desarrollar sus soluciones.

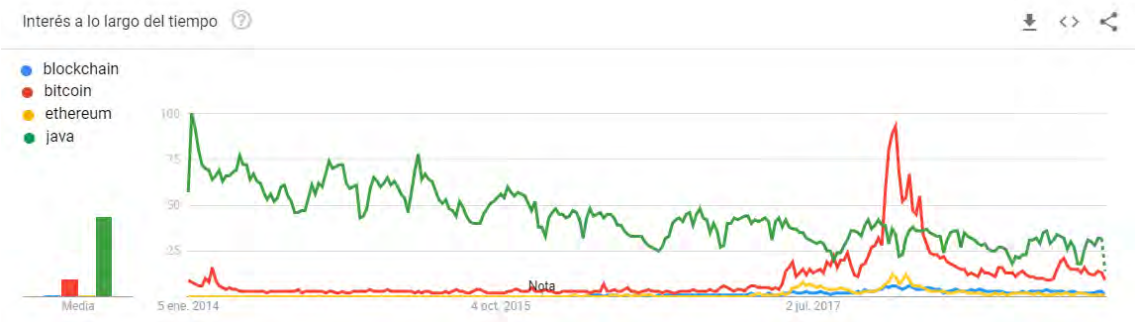


Fig. 3. Comparativa España - bitcoin, blockchain, ethereum y Real Madrid. 1/1/14 – 20/02/19

Se puede comprobar que el interés por la tecnología *blockchain*, así como sus máximos exponentes (Bitcoin y Ethereum), tiene una tendencia ascendente desde su creación. Hay que tener en cuenta que Java apareció en 1996 y está completamente desarrollado, la primera implementación de *blockchain* llegó en 2009 y aún se están definiendo los límites y posibles casos de uso de la tecnología.

Sin embargo, y pese a despertar interés entre la comunidad tecnológica, una de las razones por las cuales su adopción y el interés suscitado no han sido más elevados y no ha trascendido al usuario nivel medio, puede ser la falta de documentación que acerque la tecnología al usuario y le permita comprender las ideas fundamentales.

1.5 Estructura de trabajo.

Este trabajo fin de grado, al consistir de un estudio de una tecnología, se dividirá en diversos capítulos, de la idea más general para contextualizar los conceptos que se expondrán a continuación, hasta los *smart contracts*, el *core* de lo que se considera el gran potencial de la *blockchain* Ethereum. Se aplicará una estructura dividida en 7 capítulos:

- Capítulo 1: Ofrece una visión general del proyecto, las motivaciones y objetivos del autor, el impacto socioeconómico y el marco regulador que atañe a la tecnología *blockchain* en diferentes países del mundo.
- Capítulo 2: Introduce ampliamente la tecnología sobre la que funciona la plataforma Ethereum, haciendo foco sobre algunos conceptos que serán relevantes en capítulos posteriores.
- Capítulo 3: Pone el foco sobre el protocolo Ethereum y su máquina virtual.
- Capítulo 4: Profundiza en todo lo relacionado con los *smart contracts*.
- Capítulo 5: Describe brevemente los objetivos a corto plazo de la plataforma Ethereum, así como algunas de las mejoras en desarrollo.
- Capítulo 6: Contiene una conclusión en la que se valora en qué medida se han cumplido los objetivos y alguna apreciación personal del autor.
- Anexo: Contiene una tabla con algunos comandos de ejecución, una escala de unidades de la criptomoneda de Ethereum y un resumen en inglés del contenido del trabajo.

A lo largo del documento se utilizarán dos tipos de referencias, las notas al pie para referenciar a artículos con los que poder profundizar en algún concepto, visitar páginas web de alguna compañía, etc. y las citas bibliográficas tradicionales que indican la fuente en la que se ha basado alguna sección del presente trabajo fin de grado.

2 BLOCKCHAIN.

2.1 Introducción.

En su esencia, *blockchain* no es más que una base de datos distribuida en forma de cadena de bloques (*block-chain*). Las transacciones que se realizan sobre la red se almacenan como bloques y su resultado se anota en el libro mayor o *"ledger"* de la base de datos. Los bloques tienen una dependencia matemática entre sí y se van anidando en forma de cadena, cumplimentando el algoritmo de consenso de la red, que especifica qué bloques deben ser considerados válidos. La cadena se protege criptográficamente y se almacena parcial o totalmente en los nodos que conforman la red.

Algunas de las aplicaciones transversales de la *blockchain* son las criptomonedas (Bitcoin, Ethereum, etc.), los contratos inteligentes (*smart contracts*), las aplicaciones descentralizadas (*Decentralized Applications – DApps*) o las organizaciones autónomas descentralizadas (*Decentralized Autonomous Organization – DAO*).

Las criptomonedas son la clave para el mantenimiento de una red *blockchain*. Con ellas se aplican incentivos para recompensar a los usuarios encargados de validar los bloques y añadirlos a la cadena o se utilizan como gasolina para ejecutar los *smart contracts*, desplegarlos y almacenar información en la cadena.

Es importante a su vez, diferenciar entre protocolo y criptomoneda. El protocolo supone el código que regula cómo debe comportarse la *blockchain*, mientras que la criptomoneda es el elemento de valor que se utiliza en las transacciones. El protocolo Bitcoin llegó como un sistema financiero alternativo, abierto, descentralizado, transparente y que garantizaba el anonimato de sus usuarios. Sin embargo, este protocolo tiene determinadas limitaciones en lo que a "inteligencia" se refiere.

Por ello surgió Ethereum, como una mejora trascendental del protocolo Bitcoin. Ethereum dotaba a su código de mayor inteligencia y ofrecía una programación más sencilla para convertirse en una plataforma para desarrolladores. A diferencia de Bitcoin, Ethereum surgió con un propósito más allá de las transacciones económicas.

La gran diferencia entre Bitcoin y Ethereum es la implementación de los *smart contracts* en la segunda. Estos elementos, tan complejos como desee el programador, pueden ser totalmente independientes o depender de actividad externa (oráculos) para comprobar una condición o ejecutar su resultado. Por lo tanto, son muy versátiles y pueden emplearse para casi cualquier propósito.

Todo esto hace de Ethereum y, particularmente, de los *smart contracts* que sean muy interesantes para ser estudiados. Aún más cuando los límites de uso aún no se conocen y el conocimiento no trasciende con facilidad, pese a que la tecnología ya permita implementar soluciones.

Dentro del sector de las telecomunicaciones pueden verse multitud de aplicaciones. Entre ellas, mediante *blockchain* se podría implementar un sistema que controlara la identidad de nuestro dispositivo móvil y nos permitiera gestionar de manera casi inmediata la portabilidad de un operador a otro mediante los *smart contracts*. Otra posible herramienta sería el diseño de un controlador de versiones, que un fabricante de dispositivos móviles incluyera en sus terminales y le avisara de determinados comportamientos sospechosos como actualizaciones no oficiales o usos fraudulentos.

En este punto quedan aún muchas definiciones por concretar, muchas preguntas por resolver y muchos otros temas que aún ni siquiera se han comentado. De aquí en adelante se procede a profundizar en los temas que anteriormente se han comentado, desde *blockchain* hasta los *smart contracts*.

2.2 Antecedentes.

Blockchain no es una tecnología que haya surgido de la noche a la mañana. Es más bien una integración de diferentes ideas que nadie antes había logrado unir. Por tanto, antes de hablar de *blockchain per se*, hay que echar la vista atrás y analizar sus orígenes.

En 1985, David Chaum ya hablaba de dinero anónimo digital y del problema del doble gasto (*double spend*). El problema del doble gasto no es más que la necesidad de controlar que una cantidad x de ese dinero digital anónimo, no sea utilizado en dos transacciones simultáneamente. Es decir, evitar que se pueda replicar el valor del activo digital. Estas ideas están consideradas como una de las raíces del movimiento *Cypherpunk*.

El movimiento *Cypherpunk* surgió a finales de la década de los 80 y su objetivo era luchar por la privacidad en las comunicaciones por internet y que esta se garantizara sin la intervención de ningún gobierno, empresa u organización. Uno de los fundadores, Eric Hughes, escribió en 1993:

> *"Privacy is necessary for an open society in the electronic age. … We cannot expect governments, corporations, or other large, faceless organizations to grant us privacy … We must defend our own privacy if we expect to have any".*
>
> *"We the Cypherpunks are dedicated to building anonymous systems. We are defending our privacy with cryptography, with anonymous mail forwarding systems, with digital signatures, and with electronic money".*[3]

Para poder luchar contra el control de las unidades centralizadas, la idea era crear un ruido de fondo, para no llamar la atención y no ser detectado por ningún control basado

[3] Hughes, Eric, 9 de marzo de 1993. A Cypherpunk's Manifesto. https://www.activism.net/cypherpunk/manifesto.html

en patrones de comportamiento, y atacar aquellas vulnerabilidades de privacidad para bloquear los sistemas y que estos fueran revisados.

Por este motivo, los *Cypherpunk* tenían clara la necesidad de conseguir un método/protocolo para realizar transacciones y/o comunicaciones anónimas. El anonimato en las comunicaciones fue el punto de partida. En 1992, Eric Hughes estableció un sistema que eliminaba el remitente de los correos y los enviaba a los destinatarios correspondientes.

Surgió entonces una discusión sobre la censura y la moderación del contenido a enviar. Para no caer en una entidad centralizada que se encargara de ello, se estableció una red de nodos, *Cypherpunks Distributed Remailer* (CDR), que disponían de la lista de correo. Independientemente del nodo que recibiera el mail a retransmitir, el correo llegaría a su destinatario, siempre y cuando no se censurara o moderara su reenvío. Para que esto sucediera, todos los nodos de la red tendrían que estar de acuerdo en impedir su envío.

Sin embargo, el spam que recibía la lista comenzó a ser un inconveniente. Para prevenirlo, se planteó la posibilidad de requerir un micropago o una prueba de trabajo (*Proof of Work - PoW*) por cada envío. Finalmente se impuso la segunda propuesta.

Adam Back propuso en 1997 *Hashcash*[4] como método para llevar a cabo la PoW. El objetivo era procesar el correo con una función de hash hasta conseguir un determinado número de ceros a su inicio. El número de ceros a conseguir indica la dificultad de la prueba. Los argumentos de la función de hash eran el propio correo y un *nonce* aleatorio, que se modificaban hasta conseguir el objetivo.

Cuando se lograba el hash deseado, el correo se añadía. De esta forma el receptor podía comprobar si el *nonce* era válido sencillamente, aplicando la función de hash a la tupla correo-*nonce*, y el emisor demostraba que había invertido recursos en conseguir el objetivo.

En 1998, Wei Dai propuso *b-money*[5], un sistema para intercambiar dinero. Wei Dai proponía que todos los usuarios dispusieran de una copia de las cuentas de cada participante. Las transacciones se enviarían a todos los componentes de la red y los usuarios tendrían que actualizar sus registros con la información recibida.

Nick Szabo propuso tres ideas fundamentales para la implementación y el desarrollo de blockchain. En 1994 planteó el concepto de *smart contract*, que razonaría tres años más tarde, en 1997. **(1)**

El mismo año que propuso servidores *Byzantine Fault Tolerant* (BFT) para poder descentralizar un registro y expuso, inspirado a su vez en la publicación "*How to Time-Stamp a Digital Document*"[6] de Stuart Habert y W. Scott Stornetta, la posibilidad de almacenar información relevante como una cadena de *hashes*:

---

[4] Back, Adam, 1997. Hashcash proof-of-work http://www.hashcash.org/

[5] Dai, Wei, 1998. b-money, http://www.weidai.com/bmoney.txt

[6] Habert, Stuart y Stornetta, Scott, 1991. HOW TO TIME-STAMP A DIGITAL DOCUMENT. [En línea]. Disponible en: https://www.anf.es/pdf/Haber_Stornetta.pdf [**consulta: 05/02/2019**]

> *"… With post-unforgeable logs, via a hierachical system of one-way hash functions, a party can publically commit to transactions as they are completed by publishing signed cumulative hashes of the transaction stream…"* (2)

2.3 Bitcoin.

En 2008 'Satoshi Nakamoto' publicó el *paper* de Bitcoin[7] en la lista de correo de la web www.metzdowd.com. Bitcoin es un sistema de pagos descentralizado basado en criptografía y PoW, que surge como una mezcla de *b-money*, *bit gold*, *Hashcash* y algunas mejoras.

Bitcoin es una propuesta para realizar envíos dinerarios sin la intervención de ninguna entidad bancaria, que soluciona el problema del doble gasto en redes peer-to-peer (P2P). Sin embargo, no se utiliza una moneda estándar asegurada por una reserva de oro ni por ninguna entidad centralizada, se introduce el concepto de criptomoneda. En este caso la criptomoneda comparte el nombre del protocolo, *"bitcoin"*, y su valor lo adquiere por las leyes de la oferta y la demanda. Lo que sí está definido y limitado es su oferta, 21 millones de unidades. Esta limitación es posible porque cada 210.000 bloques minados, la cantidad con la que se recompensa al minero se reduce a la mitad.

$$N = 210000 \cdot \left(50 + \frac{50}{2} + \frac{50}{4} + \cdots\right) = 210000 \cdot 50 \cdot \sum_{n=0}^{\infty} \frac{1}{2^n} = 21.000.000$$

La única manera de obtener bitcoins en sus orígenes era mediante el minado. La acción de minar consiste en resolver la PoW, de manera que se obtiene un hash válido y el bloque se puede incorporar a la cadena. Para que esto ocurra deben sucederse algunos pasos anteriormente.

Al tratarse de una red P2P, el nuevo usuario debe buscar algún par perteneciente a la red para poder unirse. Cuando lo encuentra, solicita al par o pares encontrados una copia de la cadena de bloques y tras recibirla, la descarga y verifica por completo. Desde el primer bloque de la *blockchain*.

El siguiente paso es la creación de una transacción, para lo que se requiere disponer de un *wallet* asociado a una dirección donde guardar los bitcoins. Más adelante se profundizará en los detalles referentes a qué es un *wallet* y cómo se crean las direcciones.

Una transacción de bitcoin se compone de una serie de *inputs* y *outputs*, que podemos asociar con dinero a enviar y dinero enviado. Si la suma de los *outputs* es inferior a la de los *inputs*, el resto (*fee*) se utiliza como incentivo para mantener la red. Como *inputs*

[7] Nakamoto, Satoshi, 31 de octubre de 2008. Bitcoin: A Peer-to-Peer Electronic Cash System https://bitcoin.org/bitcoin.pdf

se incluyen las direcciones origen de los bitcoin a transferir y como *outputs* las direcciones destino.

La transacción se firma con la clave privada de la dirección origen y se incluye la clave pública de la misma para que el resto de usuarios puedan comprobar la legitimidad del usuario para transferir esos bitcoins. Sin embargo, esto no es más que una petición a la red. La transacción no será efectiva hasta que un usuario mine el bloque correspondiente y se incluya a la red.

Un bloque de Bitcoin está compuesto por:

1. Número mágico (4bytes): **oxD9B4BEF9**. Identifica la estructura del tipo de datos contenido en el bloque.
2. Tamaño del bloque (4bytes): Número de bytes máximo contenidos en el bloque sin contar los dos primeros campos.
3. Cabecera (80bytes): Cuenta con la versión (4bytes) del bloque, el hash antecesor (256bits – 32bytes), el hash raíz del árbol de Merkle (32bytes) que incluye todas las transacciones incluidas, un *timestamp* (4bytes), el campo Nbits (4bytes) que indica el valor de dificultad o número de ceros y un *nonce* (4bytes).
4. Cantidad de transacciones (1-9bytes): Contiene el número de transacciones del bloque incluyendo la transacción *coinbase*.
5. Listado de transacciones (bytes en función del tamaño del bloque). **(3)**

Una vez generado el bloque, se procede a pasar la PoW. Para ello se utiliza la función *Hashcash*, la cual recibe como parámetros la cabecera del bloque, un *nonce* (32 bits) y un *extraNonce* por si la capacidad del *nonce* no fuera suficiente para hallar un hash válido.

Todos los bloques contienen una transacción única, la transacción *coinbase*, que no dispone de *inputs*, se utiliza para recompensar a los mineros por superar la PoW y contribuir al proceso de mantenimiento de la red. Se añade automáticamente a todos los bloques por la configuración del protocolo.

Bitcoin supuso, por tanto, la primera red *blockchain*. De hecho, el término *blockchain* tal y como lo conocemos, no existía en ese momento, se acuño con posterioridad.

La red de Bitcoin calcula la capacidad computacional que tiene la propia red y ajusta automáticamente la dificultad de la PoW para que solo se pueda crear un nuevo bloque cada 10 minutos de media. De esta forma, cuantos más usuarios pertenecen a la red, más complicado es conseguir la solución a la PoW para crear un nuevo bloque y, por consiguiente, también se incrementa el coste necesario para llevar a cabo un ataque a la red. **(4)**

2.4 Características de una red *blockchain*.

Una red *blockchain* es un sistema descentralizado compuesto por nodos que se comunican mediante un protocolo estandarizado y están conectados entre sí en una misma red mediante una conexión de tipo P2P. Esos nodos almacenan copias idénticas de la información contenida en la cadena de bloques.

Todas las redes *blockchain* disponen de un protocolo en forma de software informático. El protocolo es el conjunto de reglas que estandarizan la forma en la que deben comunicarse entre sí los nodos. El código es considerado la ley.

Un nodo de la red puede ser desde un ordenador personal hasta una megacomputadora. Todos los nodos de la red disponen del mismo software (protocolo). Los nodos son los encargados de crear los bloques de la cadena y verificar las transacciones que se realizan.

Al tratarse de un sistema descentralizado, es imprescindible el consenso entre los usuarios, para poder sustituir a esa figura de autoridad de las redes centralizadas. En *blockchain*, la autoridad reside en todos sus nodos. Por lo tanto, son los propios nodos quienes deben verificar las transacciones y bloques que se están generando. Es decir, son los nodos los que aplican los algoritmos que conforman el protocolo de consenso.

En los sistemas centralizados, el control queda reservado a una única entidad, que se encarga de estipular los criterios de admisión de la red, los criterios de validez de la información, etc. Por todo lo anteriormente expuesto queda claro que, en el caso de los sistemas descentralizados, como por ejemplo *blockchain*, son los propios integrantes de la red los que implementan el control sobre la red.

Por último, es indispensable un sistema de criptografía que codifique la información que se almacena en el *ledger* y que nos permita gestionar las firmas e identidades digitales.

El resultado de la unión de todos los elementos anteriores es un *ledger* distribuido, cuyo contenido no puede modificarse gracias a los algoritmos matemáticos que protegen y relacionan los bloques y que, gracias al protocolo, se asegura la irreversibilidad del contenido de la *blockchain*. Es decir, todo lo que se publica en una *blockchain* permanecerá intacto durante el resto de los tiempos.

Una vez formada la red, un elemento básico para su mantenimiento es el grupo de mineros. El concepto de minar bloques hace referencia al proceso de creación de los bloques de la cadena tras superar el algoritmo de consenso establecido con el protocolo y la posterior recompensa ofrecida al minero.

2.5 Tipos de redes *blockchain* (y características).

2.5.1 *Blockchains* públicas sin permisos (permissionless, public, shared systems).

Este tipo de redes *blockchain* son: públicas, porque permiten a cualquier persona acceder a la información y consultar las transacciones realizadas; descentralizadas, porque todos los usuarios disponen del *ledger;* y pseudoanónimas, porque las direcciones de los propietarios son públicas y, pese a que no se pueda identificar a los usuarios, sí que pueden rastrearse las transacciones en las que ha estado involucrada la dirección en cuestión.

Por último, que sea una *blockchain* sin permisos implica que cualquier usuario puede crear bloques y procesar transacciones, siendo necesarios *tokens* nativos para incentivar a los usuarios para que mantengan la red (en este caso el concepto de *token* refiere a criptomoneda).

Algunos ejemplos de este tipo de redes son: Bitcoin, Ethereum o Litecoin[8].

2.5.2 *Blockchains* públicas con permisos (permissioned, public, shared systems).

A diferencia de las anteriores, en este tipo de redes *blockchain* la creación de bloques y el procesado de transacciones queda reservado a un grupo de participantes que debe ser conocido, no siendo necesario *tokens* nativos.

Algunos ejemplos de este tipo de redes, también conocidas como federadas, son: BigchainDB[9] o Multichain[10].

2.5.3 *Blockchains* privadas con permisos (permissioned, private, shared system).

Este tipo de redes *blockchain* son: privadas, porque el acceso a la información registrada en el *ledger* está reservado a determinados usuarios; cerradas, porque solo pueden ser usuarios de la red aquellas personas o compañías invitadas, siendo posible crear distintos niveles de privilegio para los usuarios, para proteger el acceso a determinada información o para reservar los derechos a registrar nueva información; distribuidas, que no descentralizadas, porque el *ledger* se distribuye únicamente entre un grupo cerrado de usuarios; y anónimas, porque se puede regular el nivel de anonimato de sus transacciones y usuarios.

En este caso, solo pueden crearse redes *blockchain* con permisos, ya que por definición una red *blockchain* privada sin permisos sería contradictoria.

[8] Sitio web Litecoin: https://litecoin.org/
[9] Sitio web BigchainDB: https://www.bigchaindb.com/
[10] Sitio web Multichain: https://www.multichain.com/

Algunos ejemplos de este tipo de redes son: Ripple[11], Corda[12] o Hyperledger[13].

2.6 Criptografía.

2.6.1 Funciones de Hash.

Las funciones de hash se utilizan para garantizar la integridad y proporcionar autenticidad al mensaje y como base de los sistemas de firma digital. Para ser considerada una función de hash válida, segura y eficiente se deben cumplir las siguientes propiedades:

- **Resistencia a preimagen o unidireccionalidad**: Dado un resultado de la función de hash ($h(M)$) debe ser computacionalmente imposible encontrar el mensaje.
- **Compresión**: $h(M)$ debe tener una longitud fija, sea cual sea la longitud del mensaje a procesar, el hash debe tener siempre la misma longitud.
- **Facilidad de cálculo**: El hash se debe de poder calcular rápidamente con un bajo coste.
- **Difusión**: El cambio de 1 solo bit del contenido del mensaje debe producir un efecto avalancha en el hash resultante, esa modificación debe provocar que al menos el 50% de los bits se vean afectados.
- **Segunda preimagen o colisión simple**: A partir de un mensaje conocido debe ser computacionalmente imposible encontrar otro mensaje cuyos hashes resultantes sean iguales ($h(M)=h(M')$).
- **Colisión fuerte**: Debe ser computacionalmente difícil encontrar un par <M, M'> tal que h(M)=h(M').

El protocolo de Bitcoin utiliza el algoritmo SHA-256 como función de hash principal y RIPEMD-160 en el proceso de creación de direcciones. Por otro lado, Ethereum utiliza el algoritmo Keccak-256[14], una implementación del estándar SHA-3.

2.6.2 Criptografía asimétrica.

La criptografía asimétrica o de clave pública surgió como solución a los problemas que presentaba la criptografía simétrica: Distribución de claves y firmas digitales. Whitfield Diffie y Martin Hellman (1976) fueron los primeros en introducir estos algoritmos.

A diferencia de la criptografía simétrica, en los que con la misma clave se cifra y descifra, en la criptografía asimétrica se tienen dos claves diferentes que forman un par ($key_{private}$, key_{public}). La clave pública se calcula a partir de la clave privada, que se genera

[11] Sitio web Ripple: https://ripple.com/
[12] Sitio web Corda: https://www.corda.net/
[13] Sitio web Hyperledger: https://www.hyperledger.org/
[14] Sitio web Keecak: https://keccak.team/keccak.html

aleatoriamente. Sin embargo, conociendo la clave pública debe ser extremadamente difícil conseguir la otra.

Típicamente el cifrado asimétrico se utiliza para cifrar, normalmente claves simétricas, utilizando la k_{pu} para cifrar y la k_{pr} para descifrar, y para implementar firmas digitales, utilizando k_{pr} para cifrar y k_{pu} para descifrar. La implementación de *blockchain* está relacionada con las firmas digitales, se utiliza la k_{pr} para firmar las transacciones y la k_{pu} para verificarlas.

Existen dos tipos de algoritmos de clava pública: los basados en exponenciación y los basados en curvas elípticas (*ECC – Elliptic Curve Cryptography)*. Los segundos ofrecen niveles de seguridad semejante a los primeros con claves de menor tamaño, permitiéndonos ahorrar memoria y ofreciendo mayor velocidad de procesamiento.

Los algoritmos tipo ECC, utilizan curvas criptográficas de la siguiente familia:

$$y^2 = x^3 + ax + b$$

La mayor parte de los protocolos *blockchain* utilizan los ECC. En concreto, tanto Bitcoin como Ethereum utilizan ECDSA (*Elliptic Curve Digital Secure Algorithm*) para generar las claves pública y privada que se utilizan durante el protocolo. Para ser más exacto, utilizan la curva elíptica ECDSA denominada *secp256k1*, que particulariza los parámetros $a=0$ y $b=7$:

$$y^2 = x^3 + 7$$

Los algoritmos ECC utilizan dos claves que forman un par de la forma: $(a,aP) = (k_{pr}, [k_{pr}*Punto\ Base=k_{pu}])$. Para poder calcular este par de claves se deben tener ciertas nociones de cálculo en curvas elípticas, tanto de suma como de doblado.

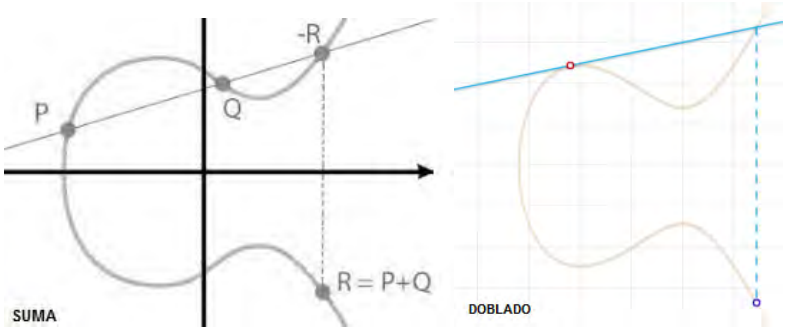


Fig. 4. Suma y doblado de punto en curvas elípticas.
Tomada de: (5)

Cabe destacar, que en las sucesivas operaciones lo que parece a primera vista una común división se trata en realidad de inversos multiplicativos. Un inverso multiplicativo puede ser calculado mediante el Algoritmo Extendido de Euclides, pero en definitiva puede decirse el inverso multiplicativo de 'x' en módulo n es aquel número que cumple:

$$a \cdot x = 1 \; mod \; n$$

Este inverso 'a' solo existirá si 'x' y 'n' son primos relativos entre sí.

La operación 'suma' hace uso de la regla de la cuerda:

$$P = (x_1, y_1), Q = (x_2, y_2) \rightarrow P + Q = (x_3, -y_3) = (\lambda^2 - x_1 - x_2, \lambda^3 + \lambda \cdot (x_1 + x_2) - v)$$
$$v = y_1 - \lambda \cdot x_1 \quad , \quad \lambda = \frac{y_2 - y_1}{x_2 - x_1}$$

La operación 'doblado' se apoya en el uso de la regla de la tangente:

$$2P = (\lambda^2 - 2x, -\lambda^3 + 2\lambda x - v)$$
$$v = \frac{-x^3 + ax + 2b}{2y} \quad , \quad \lambda = \frac{3x^2 + a}{2y}$$

A primera vista las matemáticas que se han expuestos pueden resultar muy poco intuitivas, pero esto no es cierto y únicamente se requiere generar algún par de claves para darse cuenta de la sencillez del mecanismo. Sin embargo, esto no supone uno de los objetivos del trabajo y se considera suficiente tener presente el concepto y método de cifrado.

2.7 *Wallets.*

Para poder trabajar con criptomonedas se requiere un *wallet*, que no es más que una dirección a la que se asocian tus criptomonedas. Este *wallet* está estrechamente relacionado con el par de claves pública y privada.

Por ejemplo, en el caso de Ethereum, la dirección asociada al *wallet* se obtiene de tomar los últimos 20 bytes de la clave pública. Por su parte, la clave privada, que se almacena siempre cifrada y protegida por una contraseña que decide el usuario que crea la dirección, es la que permite gestionar las criptomonedas que contenga el *wallet*. Por consiguiente, es sumamente importante no compartir esta contraseña y por supuesto, tampoco olvidarla.

En la actualidad, existen *wallets* que tienen la capacidad de utilizar diferentes direcciones para dificultar la trazabilidad de un usuario. Es decir, permiten a los usuarios preservar con mayor seguridad su anonimato.

Algunos de los *wallets* más populares en *blockchain* son: Coinbase[15], Mist[16] o Electrum[17]. Cada *wallet* destaca por algún motivo, bien sea por seguridad, por eficiencia online, por estar optimizado para dispositivos móviles, etc.

[15] Sitio web Coinbase: https://www.coinbase.com/

[16] Sitio web Mist, wallet oficial de Ethereum: https://github.com/ethereum/mist/releases

[17] Sitio web Electrum, wallet específico de Bitcoin: https://electrum.org/#home

2.8 Consenso en redes blockchain.

Como ya se ha comentado, el consenso es una herramienta necesaria e imprescindible para que las redes blockchain puedan ser descentralizadas y no se requiera de una entidad central de confianza.

El objetivo de un algoritmo de consenso es asegurar la validez de los bloques que se generan. Es decir, se asume que puede haber nodos malintencionados en la red con dos propósitos, la creación de bloques falsos o el spam.

Los algoritmos de consenso están, por consiguiente, totalmente relacionados con la minería. De hecho, la minería surge a consecuencia de los algoritmos de consenso. Los nodos mineros se encargan de añadir los bloques a la cadena y, una vez que superan la condición del algoritmo de consenso, reciben la recompensa estimada por el protocolo.

Por consiguiente, el termino consenso implica, en este contexto, que los nodos de la red se pongan de acuerdo respecto al estado global de la red.

Los nodos encargados de ejecutar el protocolo de consenso reciben el nombre de '*miners*' ('*validators*' en redes privadas). Un minero o validador que agrega las transacciones en un bloque y lo transmite a la red, recibe el nombre de '*proposer*'.

2.8.1 *Proof-Of-Work.*

Proof-of-Work es el tipo de protocolo de consenso introducido por Bitcoin y utilizado por la mayoría de redes *blockchain*. La PoW que utiliza Bitcoin consiste en un algoritmo *Hashcash*, cuya aplicación se reduce a encontrar un hash SHA256 con una determinada dificultad del tipo *leading zeros* (*i.e.* 000xx…xx → dificultad 3). Ethereum, por su parte, utiliza un algoritmo denominado *EtHash*.

Cuando un *proposer* propone un bloque al resto de nodos de la red, éstos esperan que junto con el bloque aparezca la PoW, para poder validarlo y añadirlo a la *blockchain*. En caso de que la PoW no aparezca o sea errónea, el resto de nodos rechazarán el bloque propuesto.

Simultáneamente pueden descubrirse, a partir de bloques formados por diferentes grupos de transacciones, diferentes soluciones para la PoW. Cuando esto sucede, cada minero añade su bloque a su registro de la *blockchain* y lo comparte a sus vecinos.

Esto deriva en diferentes ramificaciones temporales de la blockchain. Sin embargo, esto no es problema, ya que se impone la cadena más larga, la cadena que cuente con más bloques. Por consiguiente, cuando un nodo *miner* recibe dos posibles ramas de la misma *blockchain*, elegirá la bifurcación de mayor longitud para iniciar la búsqueda de un nuevo bloque válido.

Cuando una de las bifurcaciones se acaba imponiendo, la rama "perdedora" queda invalidada. Esto implica que los bloques pertenecientes a la cadena "perdedora" quedan invalidados, lo que previene a la red de creaciones de bifurcaciones en la cadena por parte

de nodos maliciosos, ya que al quedar invalidado un bloque, se pierde la recompensa asociada por haber minado el bloque.

Por este motivo, los nodos tratan de identificar lo que el resto de nodos estimaran como el estado global que el resto de nodos estiman como el propio estado global de la cadena.

Por ello se dice que *proof-of-work* no es interactivo, y es precisamente esta cualidad, lo que permite un consenso rápido y seguro.

De hecho, los algoritmos de tipo PoW no aplican el algoritmo de consenso, sino que lo incentivan para fomentar la existencia de nodos bondadosos, cuyo único objetivo sea el de recibir la recompensa del minado, que protejan a la red de la posible actividad de nodos maliciosos.

El puzle a resolver puede asemejarse a una competición, ya que todos los nodos de la red compiten por resolver el mismo puzle al mismo tiempo. El tiempo que transcurre desde el inicio del puzle hasta que se descubre su solución, se denomina *blocktime*. El proceso para obtener la solución es el siguiente:

1. Se agrupan transacciones que no pertenezcan a ningún bloque en uno nuevo, previa validación de las transacciones.
2. Se añade un número aleatorio (*nonce*).
3. Se obtiene el hash del conjunto (bloque & *nonce*).
4. Se comprueba los ceros que contiene al inicio el hash (*leading zeros*).
5. Si se obtiene un hash que supere la dificultad actual, el bloque puede proponerse junto con el *nonce*, que supone la muestra de la PoW.
6. En caso contrario, se vuelve al paso 2. **(1)**

Como puede intuirse, la complejidad de resolver el puzle es sustancialmente mayor que la de comprobar su solución. La manera de validar un bloque es tan sencilla como realizar el hash del bloque recibido junto con el *nonce* adjunto y comprobar si supera la dificultad actual.

La capacidad de la red para determinar nuevos hashes se denomina *hash rate*. Esta se incrementa cuando se unen nuevos mineros o se reduce cuando los mineros abandonan la red. La *hash rate* de Bitcoin se sitúa en torno a los 47.000.000 TeraHashes por segundo (TH/s)[18], mientras que la de Bitcoin es aproximadamente de 154.000 TH/s[19].

Obviamente, para llegar a esas tasas de generación de hashes, se requiere mucha capacidad computacional y mucha, muchísima, energía. Es uno de los grandes inconvenientes de los algoritmos de consenso basados en PoW, el consumo de energía. De hecho, en algunos lugares de China existen plantas hidroeléctricas dedicadas

[18] Dato actualizado a 19 de febrero de 2019. La información a tiempo real puede consultarse en https://www.blockchain.com/es/charts/hash-rate

[19] Dato actualizado a 19 de febrero de 2019. La información a tiempo real puede consultarse en https://etherscan.io/chart/hashrate

exclusivamente a proveer de energía a granjas de minería de Bitcoin. Este es un aspecto que las *blockchains* deben mejorar y por ello existen diversos proyectos para encontrar nuevos algoritmos de consenso.

2.8.2 *Proof-Of-Stake.*

Como alternativa a los algoritmos de *proof-of-work*, surgen los basados en *proof-of-stake* (PoS). Este tipo de algoritmos surgen principalmente para intentar conseguir la misma eficacia en el consenso, pero reduciendo el coste de energía para implementar el control sobre la red.

En este tipo de algoritmos, la responsabilidad de crear un nuevo bloque se reparte mediante *round-robin*. Las recompensas, en caso de crear un bloque válido, se mantienen. Simplemente se limita la posibilidad de intentar crear un bloque nuevo. Como en este escenario, la responsabilidad de crear un nuevo bloque va a recaer sobre un único nodo, aunque vaya variando, la forma de evitar la actividad maliciosa es mediante multas. Cuando un nodo viola determinadas reglas de creación de bloques, en vez de recibir la recompensa perderá dinero.

Mientras que los algoritmos de PoW tienen como objetivo premiar la actividad de los mineros, el funcionamiento de los algoritmos de PoS se basa en castigar la intención de dividir la cadena. De hecho, cuando dos nodos no están de acuerdo respecto a cuál debe ser el siguiente bloque, tienen la obligación por decantarse por un único camino, aunque esto supongo una pérdida de *stake* para uno de los nodos.

Este tipo de algoritmos es menos conocido, pero existen algunas propuestas como por ejemplo Casper, una actualización de Ethereum que está actualmente en fase de desarrollo.

De hecho, ya en 1997 Nick Szabo hablaba de sistemas de reputación basados en la trayectoria de los usuarios. Siendo necesario definir casos objetivos en los que se incrementa la reputación negativa. Sin embargo, su implementación, al ser más compleja que la de los algoritmos PoW, ha limitado su expansión.

2.9 ICOs.

Los nuevos protocolos que surgen en redes blockchain utilizan normalmente un método de financiación denominado *Initial Coin Offering* (ICO). Esta herramienta consiste básicamente en ofrecer a los inversores *tokens* a un precio especial.

Pero antes de continuar, hay que detenerse y aclarar el concepto de *token*. Un *token* es una unidad de valor que representa un activo digital o un valor financiero. Sin embargo, el *token* y la criptomoneda no son exactamente el mismo tipo de criptoactivo. De hecho, la criptomoneda es un tipo de *token* que supone el equivalente al dinero fiduciario y sirve para ejecutar transacciones. En sí mismo, un *token* es una cadena alfanumérica que

representa cualquier tipo de activo del mundo real, desde una moneda hasta la propiedad de un bien como pueda ser una vivienda.

Cuando el *token* representa un bien del mundo real recibe el nombre de *digital asset* mientras que, cuando lo que representa es la propiedad de ese bien se denomina *mirror asset*.

Las ICOs pueden ser de venta limitada, ofreciendo un número de *tokens* fijos o un periodo de oferta determinado, o bien de venta ilimitada, en las que la disponibilidad de adquisición de *tokens* no está restringida ni en términos de cantidad ni en términos de tiempo.

Vitalik Buterin, analizó diversas ICOs y sus respectivos problemas para sintetizar una lista de propiedades deseadas.

- **Certeza de valuación**: Al participar en una ICO se debe tener certeza sobre al menos un límite en la valoración.
- **Certeza de participación**: En caso de querer participar en una ICO se debería poder contribuir en la financiación.
- **Limitar la cantidad a recibir**: Para mitigar el riesgo de atención reguladora o la centralización.
- **Sin banco central**: El emisor de los *tokens* debe finalizar la ICO sin una cantidad de estos criptoactivos que le permitan controlar el mercado.
- **Eficiencia**: Para evitar configuraciones económicas subóptimas o pérdidas irrecuperables.

Sin embargo, conseguir que estas 5 condiciones se cumplan simultáneamente, no es una tarea sencilla. Las dos primeras interfieren con las otras tres, por lo que aún se están estudiando formas de conseguir la ICO más justa y eficiente en términos económicos. (4)

2.10 Actualizaciones en *blockchain*.

Cuando se detectan vulnerabilidades o nichos de mejora en un protocolo *blockchain*, como en cualquier otra tecnología, existen diferentes formas de actualizarlo y adaptarlo a las nuevas necesidades o evitar las deficiencias detectadas.

2.10.1 *Hard fork*.

Se trata de una actualización de tipo *hard fork* cuando se produce un cambio en las reglas del protocolo que provoca incompatibilidad con su versión anterior. Esta bifurcación es permanente, por lo que todos los usuarios que quieran utilizar la nueva versión deberán actualizar la versión de su software.

Cuando se produce una actualización *hard fork* los usuarios con la nueva versión generan bloques que no son válidos para la versión anterior. De la misma manera que los bloques generados con la versión anterior no pueden ser validados con la actualización.

Este tipo de actualización puede ser consensuada o no. La diferencia es simple, en el caso de ser consensuada todos los usuarios actualizaran a la nueva versión, mientras que en caso contrario se producirá una división en la red. Esta división provocará la creación de una nueva *blockchain* y una criptomoneda totalmente independiente a la anterior. El ejemplo más conocido de una actualización no consensuada de tipo *hard fork* sucedió con el protocolo Ethereum, cuya red se dividió entre Ethereum (ETH) y Ethereum Classic (ETC), la versión inicial.

Otra característica interesante de este tipo de actualizaciones es que permiten reescribir el pasado, por ejemplo, para revertir ataques, robos o solventar vulnerabilidades en el proceso de la transacción, como ocurrió en el mencionado *hard fork* de Ethereum.

Desde el punto de vista del desarrollador, son las actualizaciones más sencillas de implementar. Las nuevas reglas diseñadas no tienen por qué ser compatibles con ninguna de las reglas anteriores.

2.10.2 *Soft fork.*

Este tipo de actualizaciones es mucho más compleja en lo que a implementación de código se refiere. Sin embargo, tiene una ventaja respecto a la anterior, no es necesario la creación de una nueva *blockchain* paralela.

En este caso, los usuarios que disponen del software post *soft fork*, generan bloques que son válidos para el *software* anterior, por lo que las personas que no actualicen sus equipos a la nueva versión podrán seguir participando en la blockchain. Pero a la inversa no sucede igual, los usuarios que disponen de la versión previa al *soft fork*, generan bloques que no son válidos para el nuevo *software*.

Por lo tanto, este tipo de actualizaciones son menos propensas a generar divisiones en la cadena y requieren únicamente el consentimiento de los nodos mineros o validadores, mientras que el resto de usuarios pueden mantener el software antiguo.

Un ejemplo de *soft fork* sucedió en el protocolo Bitcoin al incorporar un algoritmo de hash opcional de pago (P2SH[20]).

2.10.3 *Split chain.*

Cuando una *blockchain* se actualiza y, debido a esa actualización, surgen dos *blockchains* diferentes que comparten la misma historia hasta el punto en el que se sucede el *fork*, se produce un fenómeno denominado *'Split chain'*.

Debido a que a partir del *fork*, las *blockchains* son totalmente independientes, cada cadena pasa a tener su propia moneda. El conjunto de nodos que mantiene la versión del protocolo anterior a la actualización mantiene la criptomoneda original, mientas que los

[20] Información más detallada en: https://en.bitcoin.it/wiki/Pay_to_script_hash

nodos que se han unido a la nueva versión del protocolo comienzan a utilizar una nueva criptomoneda.

Un ejemplo de esta casuística sucedió tras un *hard fork* en el protocolo Bitcoin que propició la aparición de una nueva *blockchain* independiente, Bitcoin Cash (BCH)[21].

2.11 *Light clients*.

Como se ha comentado en el apartado de Bitcoin, cuando un usuario se une a la red de un protocolo *blockchain*, tiene que descargar y validar cada uno de los bloques de la cadena desde los inicios del protocolo. Esto supone un problema de escalabilidad.

Los *light clients* permiten el acceso a la *blockchain* de manera segura y descentralizada sin la necesidad de descargar la cadena de bloques al completo y permitiendo al usuario interactuar únicamente con la información de la cadena de bloques que considere oportuna. Sin embargo, estos clientes o nodos no son autónomos por sí solos, requieren de *full nodes* para que actúen de intermediarios con la blockchain.

Por consiguiente, los protocolos *blockchain* requieren de *full nodes* para poder mantener la red. Son este tipo de nodos los que están conectados a la red permanentemente y se encargan de validar todas las transacciones y reenviar la información a sus vecinos P2P. De hecho, los nodos mineros son un ejemplo de *full nodes* que se encargan de buscar soluciones al problema matemático del algoritmo de consenso establecido por el protocolo para añadir nuevos bloques a la cadena.

Un *light client* valida únicamente las cabeceras de los bloques, lo que supone un reto importante a la hora de ser diseñado. Típicamente, este proceso se realiza apoyándose en el árbol de Merkle que contienen las cabeceras.

Un árbol de Merkle no es más que la sucesión de hashes de las diferentes transacciones que conforman el bloque. Los nodos del nivel más profundo del árbol corresponden con las transacciones y, el resto de los nodos supone el hash de los dos nodos que están inmediatamente por debajo de él. Por consiguiente, la raíz del árbol contiene un hash que identifica a todas las transacciones del bloque y que permite validar si, en efecto, una transacción pertenece a un determinado bloque.

[21] Sitio web de Bitcoin Cash: https://www.bitcoincash.org/

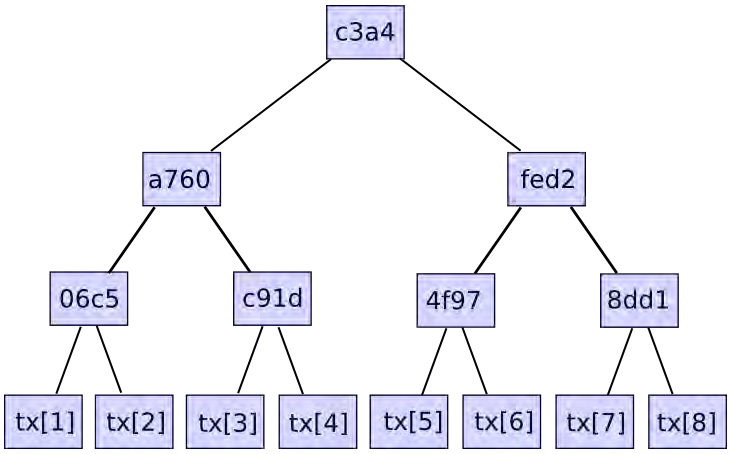


Fig. 5. Merkel *tree*.
Tomada de: (7)

Algunos de ejemplos de usos de *light client* pueden ser los siguientes: Conocer el estado de una cuenta (su balance, su código…); Comprobar si una transacción ha sido aceptada y confirmada; Filtrar las cabeceras recibidas en función los intereses del usuario, como por ejemplo en función de la dirección que haya realizado la transacción.

3 ETHEREUM.

3.1 Introducción.

Casi todos los conceptos anteriormente expuestos, estaban presentes en la arquitectura del protocolo Bitcoin. Es indiscutible que Bitcoin supuso una revolución e introdujo el concepto de *blockchain* en el panorama tecnológico.

Sin embargo, esto no era suficiente para algunos de los desarrolladores del momento. El protocolo de Bitcoin fue concebido como un libro de contabilidad global en el que las transacciones utilizaban los propios *tokens* del protocolo (bitcoins) para facilitar los intercambios económicos sin la necesidad de confiar en un tercero. Su programación es muy eficiente pero muy poco flexible.

Por consiguiente, algunos programadores se enfocaron en buscar la manera de aprovechar todas las ventajas que ofrecía Bitcoin, pero ofreciendo alguna alternativa más flexible.

En diciembre de 2013, un joven ruso de tan solo 19 años, que estaba trabajando e investigando en la comunidad Bitcoin, publicó un *whitepaper* en el que explicaba detalladamente el diseño técnico y racional de un nuevo protocolo que operaba sobre *blockchain*. El joven se llamaba Vitalik Buterin y su propuesta se denominaba Ethereum.

Esta publicación atrajo a otros programadores, que consideraron realmente interesantes las propuestas de Vitalik. De hecho, Vitalik comenzó a trabajar con el Dr. Gavin Wood, quien escribió en 2015 el *yellowpaper*, donde se describe el funcionamiento de la *Ethereum Virtual Machine* (EVM), y con Joseph Lubin, quien fundó simultáneamente ConsenSys[22], una compañía de software orientada a la creación de aplicaciones descentralizadas (DApps). Todos ellos son considerados los fundadores de la plataforma Ethereum, no así sus dueños.

Una cosa muy importante es que, aunque los fundadores y desarrolladores de Ethereum estén claramente identificados, el protocolo *per se* no pertenece a nadie. Pertenece a la comunidad. Pertenece a todos aquellos que forman parte de la red Ethereum.

3.2 Vitalik Buterin.

Antes de continuar profundizando en el protocolo Ethereum, es necesario detenerse a presentar al principal "culpable" de que esta propuesta haya tomado vida.

Vitalik Buterin nació el 31 de enero de 1994 nacido en la ciudad rusa de Kolomna. A la temprana edad de 6 años, sus padres decidieron emigrar a Canadá para buscar mejores oportunidades y ofrecerles un mejor futuro a sus hijos.

[22] Sitio web ConsenSys: https://new.consensys.net/

Desde muy joven Vitalik empezó a diferenciarse por sus capacidades mentales y su interés por la informática, quizás influenciado indirectamente por la ocupación de su padre, Dmitry Buterin, quien se autodefine como un emprendedor en el mundo de la informática.

Entre 2007 y 2010 fue un jugador activo del universo World of Warcraft, un conocido videojuego de rol multijugador en línea. Sin embargo, Blizzard[23] decidió eliminar la característica ofensiva principal del personaje predilecto de Vitalik, Siphon Life. En este momento comenzó a tomar una postura contraria respecto a los sistemas centralizados y, al poco tiempo, dejó de jugar.

Pronto descubrió Bitcoin y centró sus esfuerzos en comprender los entresijos de ese proyecto. De hecho, comenzó a escribir *posts* para el blog "Bitcoin Weekly", lo que le permitió conocer a Mihai Alisie, con quien fundaría "Bitcoin Magazine".

En 2012 comenzó sus estudios universitarios en la University of Waterloo, pero el propio Vitalik llegó a la conclusión de que estos estudios le estaban aportando menos que los cripto-proyectos en los que estaba involucrado. Por ese motivo, tan solo un año más tarde, decidió abandonar la universidad e iniciar una "vuelta al mundo" para conocer distintos cripto-proyectos.

De cada uno de ellos, fue obteniendo diferentes ideas y contactos. Sin embargo, Vitalik consideraba que todos los proyectos que iba conociendo, tenían un rango de miras de poca amplitud. Se centraban en aplicaciones específicas, problemas concretos. Entonces, comenzó a gestar el proyecto que le lanzaría a la fama, Ethereum.

El propio Vitalik Buterin recomienda una autobiografía que incluir en aquellas conferencias a las que va a ser invitado, artículos que hacen referencia a su persona, etc. Como no podía ser de otra manera, este trabajo acepta la recomendación:

> "*Vitalik is the creator of Ethereum. He first discovered blockchain and cryptocurrency technologies through Bitcoin in 2011, and was immediately excited by the technology and its potential. He cofounded Bitcoin Magazine in September 2011, and after two and a half years looking at what the existing blockchain technology and applications had to offer, wrote the Ethereum white paper in November 2013. He now leads Ethereum's research team, working on future versions of the Ethereum protocol.*"
>
> (5)

3.3 Filosofía.

El protocolo de Ethereum surgió con el objetivo de convertirse en una plataforma de desarrollo de *smart contracts* y *Decentralized Applications (DApps)* así como la creación de nuevos protocolos basados en el propio Ethereum.

[23] Blizzard Entertainament es la compañía que desarrollo el juego *World of Warcraft*.

La filosofía con la que surgió Ethereum está basada en 5 principios: Simplicidad, universalidad, modularidad, agilidad y permisividad.

El protocolo debe buscar la simplicidad, es decir, debe diseñarse de tal manera que cualquier programador con conocimientos básicos tenga la capacidad de comprender el funcionamiento de Ethereum.

Que el protocolo busque la universalidad significa que debe permitir realizar cualquier tipo de transacción que pueda ser definida matemáticamente a partir de un lenguaje de programación *Turing-complete*.

La modularidad implica que las características funcionales de Ethereum deben ser programadas en módulos de manera independiente para, en caso de ser necesaria alguna modificación no tener que modificar el protocolo por completo.

El protocolo debe ser ágil en lo que a modificaciones se refiere. Es decir, en caso de que se detecte un bug o una mejora sustancialmente necesaria, el proceso de incorporación debe ser rápido y sencillo.

Por último, Ethereum debe caracterizarse por ser un protocolo permisivo, un protocolo que permita que cualquier aplicación se implemente sobre su plataforma y que las modificaciones busquen únicamente reducir posibles vulnerabilidades o implementar mejoras para la comunidad. (6)

3.4 EIPs

Las *Ethereum Improvement Proposals* (EIPs), o propuestas de mejora de Ethereum, conforman un mecanismo para que los usuarios de la comunidad Ethereum propongan y discutan posibles mejoras del protocolo, así como la implantación y /o modificación de estándares.

Una EIP es un documento que incluye las especificaciones técnicas de la propuesta y una justificación razonada de la proposición. Por otro lado, el autor de la EIP es responsable de documentar aquellas opiniones contrarias a su idea.

En la redacción y mantenimiento de la EIP intervienen todos los usuarios que participen en el consenso de la propuesta, el autor de la EIP, los editores autorizados y los desarrolladores principales de la comunidad.

Se diferencian tres tipos de EIP:

- **EIP de seguimiento estándar (*standar track EIP*)**: Se incluyen en esta categoría todas aquellas EIP que provocan cambios en todas o la mayoría de implementaciones de Ethereum. Se componen de un documento de diseño, la implementación de la propuesta y, en caso de ser necesario, una actualización del *yellowpaper*.

  - ***Core***: Propuestas que conllevan un fork bajo consenso o que puedan ser relevantes en discusiones *coredev*[24].
  - ***Networking***: Aquellas EIP relacionadas con *devp2p*, *Light Ethereum Subprotocol* o *whisper*, que se detallarán más adelante.
  - ***Interface***: Proposiciones que engloban cambios en las especificaciones o estándares de la API/RPC de un cliente, el renombramiento de un método o los contratos ABI.
  - **ERC**: *Ethereum Request for Comment*. Estándares a nivel de aplicación, desde *tokens* hasta formatos de *wallets*.
- **EIP informativa**: Se utilizan para transmitir recomendaciones, que pueden ser ignoradas, o para describir algún problema en el diseño de Ethereum, no incluyendo ninguna propuesta. Si se incluye alguna propuesta, pasaría a ser una *standar track EIP*.
- **Meta EIP**: Son semejantes a las *standar track EIPs*, pero aplicando al ecosistema que rodea la protocolo Ethereum, no al protocolo *per se*.

Para que la comunidad conozca en todo momento el estado y fase de desarrollo/aprobación en que se encuentra una EIP, se definieron una serie de estados por los que la propuesta debe avanzar hasta ser aceptada como válida. Los estados que componen el ciclo de aprobación son los siguientes:

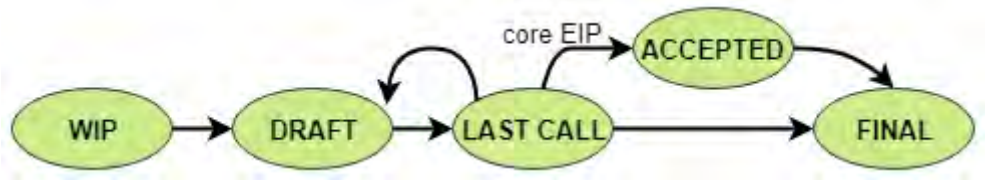


Fig. 6. Flujo de estados de una EIP.
Creada por el autor

- ***WIP***: *Work In Progress* (En progreso). Una vez que la propuesta recibe cierto soporte de la comunidad Ethereum, el autor de la EIP tiene que escribir un documento en el que se especifiquen las características técnicas y, en algunos casos, la implementación. En caso de que los editores validen la propuesta, le asignarán un número de EIP, sino la rechazarán razonando los motivos.
- ***Draft***: (Borrador). En este estado, se realizan todas las modificaciones que el autor considere necesarias para que el documento alcance un grado de calidad y validez tal que, a priori, le permita avanzar de estado. Los editores pueden rechazar

[24] Las discusiones *coredev* o *core development* son congregaciones de equipos de desarrollo de Ethereum en las que se discuten algunas EIPs para definir la dirección en la que se considera más adecuada que evolucione Ethereum.

el avance hacia *last call*, porque se crea que queden correcciones o modificaciones pendientes, o pueden aceptar y asociar el siguiente estado a la EIP.

- ***Last call***: (Última llamada). La EIP se publica[25] provisionalmente para que sea valorada por todos los usuarios que consulten las novedades respecto a estándares del protocolo. Si se encuentran objeciones técnicas considerables, el documento se devuelve al estado *draft* para revisarlo. En caso contrario, continúa avanzando al siguiente estado, que puede ser *accepted* en caso de tratarse de una *Core EIP* o *final* en el resto de caso.
- ***Accepted***: (Aceptada). La propuesta debe ser implementada en al menos 3 clientes Ethereum y adoptada por la comunidad para poder considerarla finalizada.
- ***Final***: La propuesta pasa a considerarse un estándar y únicamente puede modificarse para corregir pequeñas erratas. En el supuesto de considerar una mejora sustancial, debería volver a iniciarse el proceso con una nueva consulta a la comunidad, propuesta, documentación, etc.

Sin embargo, existen algunas casuísticas que no encajan en ninguno de los estados previamente definidos. Por ello se definieron también algunos estados excepcionales:

- ***Deferred***: (Aplazado). EIPs que se consideran potencialmente interesantes o útiles pero para un futuro post-bifurcación. Por ello, se aplaza su revisión y aprobación.
- ***Rejected***: (Rechazado). No se considera útil o viable y el cuerpo de desarrolladores de Ethereum lo rechaza sin implementarlo.
- ***Active***: (Activa). Aquellas EIPs que pueden ser actualizadas sin modificar su número. El mejor ejemplo es la EIP1, que describe la manera de tipificar una EIP, el proceso a seguir, etc.
- ***Superseded***: (Remplazado). Estado que se asocia a aquellas EIPs, que son actualizadas por nuevas EIPs y dejan de ser el estándar de referencia en favor de su actualización.

(7)

3.5 Fases de Ethereum

Como ya se ha comentado, el *whitepaper* de Ethereum fue publicado en diciembre de 2013 por Vitalik Buterin. Rápidamente, en enero de 2014, la empresa suiza *Ethereum Switzerland GmbH* comenzó a financiar al primer equipo de desarrolladores del protocolo

[25] Todas las EIP pueden consultarse en la web: https://eips.ethereum.org/

Ethereum, formado por el propio Vitalik Buterin, Mihai Alisie, Anthony Di Iorio y Charles Hoskinson.

En el mismo 2014, en junio, se funda una organización sin ánimo de lucro con el nombre "*Ethereum Foundation*"[26]. Sin embargo, al tratarse de una organización sin ánimo de lucro, pronto se hicieron latentes las necesidades de financiación para poder sostener la etapa de desarrollo de la primera versión de la red Ethereum.

El equipo de desarrollo consideró que la mejor opción para conseguir financiación era realizar un *crowsdale* o ICO. Entre julio y agosto de 2014, a través de la *Ethereum Foundation* se ofertó la inversión en futura criptomoneda ether a cambio de bitcoin, que por entonces ya disfrutaba de un valor fiduciario en el sistema financiero tradicional. El resultado fue muy sorprendente, la propuesta tuvo una gran recepción en la cripto-comunidad y se obtuvieron fondos por valor de 18 millones de dólares americanos.

En mayo de 2015 se lanzó una *testnet*, a modo de red privada, para que los desarrolladores pudieran poner a prueba el protocolo, en previsión al despliegue de una red pública de Ethereum.

Desde el lanzamiento, de manera pública, la fase de desarrollo de la plataforma Ethereum contemplaba un *roadmap* dividido en cuatro grandes fases, que se proceden a detallar:

3.5.1 Frontier.

El 30 de julio de 2015 se lanzó la etapa Frontier, una versión muy básica de lo que se pretendía que fuera Ethereum, pero que permitía a los usuarios desplegar *smart contracts*, probar aplicaciones descentralizadas o minar ethers.

No se aseveraba la seguridad de la plataforma. Sin embargo, este no era el objetivo principal de esta etapa. En esta etapa se pretendía testar en el mundo real los desarrollos implementados hasta el momento.

El lanzamiento fue, según los objetivos marcados, todo un éxito. No se detectaron *bugs* críticos ni grandes deficiencias en la solución implementada. Se había presentado una plataforma descentralizada que permitía trabajar con *smart contracts* y había respondido satisfactoriamente.

Si bien es cierto, se introdujo una pequeña modificación. La tasa de generación de bloque se consideraba elevada, lo que suponía que la dificultad de la prueba de trabajo (PoW) era reducida. Por ese motivo se decidió implementar una "bomba de dificultad",

[26] La página web oficial de Ethereum Foundation es: https://www.ethereum.org/

que incrementaba la dificultad del algoritmo PoW y la velocidad con la que en el futuro se iría incrementando automáticamente.

Este hecho estaba plenamente orientado a "obligar" al protocolo a buscar un algoritmo de consenso de un tipo diferente, de *Proof of Stake*. Debido a que con el tiempo aumentaría automáticamente la velocidad de incremento de la dificultad de la PoW, llegaría un momento en el que fuera sumamente costoso y fueran más interesantes otras alternativas.

3.5.2 Homestead.

El 14 de marzo de 2016, a partir del bloque 1.150.000 y bajo la EIP2, se introdujo la primera fase estable del protocolo. Con el lanzamiento de la fase Homestead se inició la primera etapa segura, según el propio equipo de desarrollo de *Ethereum*. El objetivo de esta fase era asentar la plataforma Ethereum en su versión estable a la par que se trabajaba en la siguiente actualización.

En esta actualización se incrementó el coste de despliegue de *smart contracts* (gas) y se presentaba *Mist*, una interfaz gráfica que permitía interactuar con la *blockchain* sin la necesidad de tener que recurrir a una ventana de comandos.

3.5.2.1 *The DAO*

Sin entrar en detalles de qué es una *Decentralized Anonymous Organization* (DAO), ya que se dedicará un capítulo más adelante, en esta etapa Slock.it[27] desplegó una DAO con el nombre de "The DAO". El objetivo de esta organización era actuar como un fondo de inversión.

Cualquiera podía exponer su propuesta y solicitar financiación para poder llevarla a cabo. Los inversores votarían entre las diferentes propuestas y, en el supuesto de que se lograra el consenso adecuado, se transferían los fondos a la dirección del solicitante.

Como idea era muy potente, ya que ofrecía a grupos sin apenas poder adquisitivo una posibilidad para desarrollar sus ideas. De esta forma, se estaba incentivando indirectamente el uso de la plataforma *Ethereum*.

La idea captó la atención de tantos usuarios que, en su *crowsdale* particular alcanzó los 150 millones de dólares americanos. Sin embargo, se detectó un *bug* en el código del *smart contract* que controlaba la organización. Existía una función recursiva que permitía a los usuarios retirar el doble de los fondos que hubieran invertido.

[27] Sitio web Slock.it: https://slock.it/

Esta vulnerabilidad fue detectada y notificada al equipo de desarrollo del *smart contract*, sin embargo, el cofundador de Slock.it, Stephen Tual, descartó que esa vulnerabilidad pudiera afectar realmente a The DAO. A los pocos días de que se desestimara la amenaza, un atacante utilizó la falla para retirar entre 50 y 100 millones de dólares americanos de los fondos de la organización.

A modo de inciso, conviene recordar que el código se considera la ley. El "atacante" realmente no había cometido una ilegalidad, simplemente interactuaba con el código y aprovechaba una casuística que los desarrolladores no habían controlado.

Esto supuso un cisma en la comunidad que derivó en dos corrientes de pensamiento. La comunidad se dividió entre usuarios que estaban de acuerdo con esa filosofía y usuarios que, pese a no renegar que el código es la ley, consideraban que se había cometido un robo y no se podía aceptar.

Como resultado de esta división tuvo lugar un *hard fork* que dividió la cadena de bloques en dos estructuras paralelas. La *blockchain* de "Ethereum Classic" mantuvo las transacciones relacionadas con *The DAO*, mientras que la *blockchain* de Ethereum devolvió los fondos a sus propietarios y reescribió la historia sin dejar rastro de la organización fallida. Este evento está descrito en la EIP779, que se hizo efectiva en el bloque 1.920.000.

3.5.2.2 *Tangerine Whistle.*

En esta etapa se detectó otra posible deficiencia. El coste de ejecutar determinados *opcodes*, como SLOAD o BALANCE, era muy inferior al coste computacional asociado. Esto podía ser aprovechado para planificar ataques de denegación de servicio contra la EVM.

La solución aplicada fue incrementar el coste de gas asociado a ejecutar los *opcodes* investigados. La información está recogida en la EIP150 y se hizo efectiva el 18 de octubre de 2016 a partir del bloque 2.463.000.

3.5.2.3 *Spurious Dragon.*

En la misma línea, se detectó otro posible escenario en el que planificar un ataque de denegación de servicio con éxito. El coste en gas de crear una cuenta sin ninguna información, ni código desplegado, etc. era muy reducido, por lo que a muy bajo coste se podía incrementar considerablemente el tamaño de la cadena de bloques.

Se incrementó el coste de creación de una cuenta y se eliminaron de los registros de la *Ethereum Virtual Machine*, todas aquellas cuentas que no disponían de ningún dato. La propuesta fue discutida en la EIP607 y se hizo efectivo el 22 de noviembre de 2016 a partir del bloque 2.675.000.

3.5.3 Metrópolis.

La tercera fase en el desarrollo de la plataforma Ethereum, se denominó Metrópolis. Su objetivo era hacer crecer la plataforma de una manera estable y segura, ofreciendo mayor capacidad de soporte para un mayor número de usuarios y aplicaciones descentralizadas.

La comunidad de desarrolladores de Ethereum, conscientes del número de modificaciones/mejoras a implementar y alcance de las mismas, decidieron subdividir esta fase en dos etapas.

3.5.3.1 Byzantium.

El 16 de octubre de 2017, a partir del bloque 4.370.000 y bajo la EIP609, se introdujo la primera de las etapas de la tercera fase, Byzantium.

Uno de los principales cambios de esta versión afectaba a la bomba de dificultad. Debido a que el desarrollo de la versión de Ethereum cuyo algoritmo de consenso fuera de tipo PoS se estaba retrasando, se ralentizó la acción de la bomba dificultad. Sin embargo, y pese a ser un cambio relevante, no deja de ser una modificación que no mejora en nada el protocolo.

En este sentido si es importante la inclusión de la instrucción "*REVERT*" en la *Ethereum Virtual Machine*, que permite interrumpir la ejecución de cualquier *smart contract* si se detecta alguna excepción y revertir los cambios realizados, de manera que se reduce el gasto computacional en la EVM y se evita consumir todo el gas proporcionado para la ejecución.

Se incluye la precompilación de los *smart contracts* para permitir la inclusión de **zkSNARKs** en Ethereum. zkSNARKs, o *zero-knowledge Succint Non-Interactive Argument of Knowledge*es, es un protocolo que permite incrementar la privacidad en la *blockchain*, ya que está formado por una serie de algoritmos que permiten evitar la ejecución de un *script*, en nuestro caso un *smart contract*.

En resumidas palabras, el protocolo zkSNARKs permite probar la veracidad de una información sin necesidad de revelarla, a partir de pruebas criptográficas. Es decir, permite comprobar la validez de un bloque o que una transacción con información confidencial ha sido ejecutada correctamente, sin necesidad de conocer su contenido.

Por último, con esta actualización se comenzaba a incluir el estado final de la transacción en los *transaction receipts* o conjunto de datos de salida de la transacción, de manera que permitiera a los usuarios conocer el origen de la información contenida en el mismo *transaction receipt* y evitar algunas ejecuciones.

En caso de ser fallida, la información de salida será la originada tras ejecutar la instrucción REVERT y no será necesario ejecutar la transacción, ya que los cambios temporales habrán sido revertidos. Esta medida simplifica la tarea de los clientes ligeros (*light clients*).

3.5.3.2 Constantinople.

La actualización correspondiente a la segunda etapa de la fase Metrópolis no ha entrado en vigor en la actualidad. Sin embargo, se pueden conocer, en la EIP1013, las mejoras que van a ser implementada.

Una de las propuestas más interesante es la inclusión de operaciones a nivel de bit en la *Ethereum Virtual Machine*. Actualmente la máquina virtual admite operadores lógicos y aritméticos, pero no a nivel de bit. Esto va a permitir reducir el coste de operaciones. Las operaciones van a consistir en el movimiento de bits a izquierda o derecha (*bitwise shifting*)

Otra innovación referente a la EVM consta de la creación de algunos *opcodes*, como por ejemplo "EXTCODEHASH", que devuelve el hash del código de un contrato. Actualmente se dispone del *opcode* "EXTCODECOPY" que devuelve el código por completo del *smart contract*. Sin embargo, para comprobar la existencia de un contrato no es necesario obtener su código, con recibir su hash puede ser necesario. Esta medida reducirá la cantidad de gas utilizada en algunos casos.

Por último, también se van a adoptar medidas externas a la máquina virtual. La bomba de dificultad se va a retrasar nuevamente y se va a reducir la recompensa por minado de bloque, de 3 ether a 2.

Se ha comprobado que los plazos estimados para finalizar el desarrollo de la versión de Ethereum regulada con un algoritmo de consenso de tipo PoS, han sido siempre insuficientes. Por ende, se está discutiendo una propuesta para eliminar la bomba de dificultad (EIP1234).

Debido al retraso que está sufriendo esta actualización, que estaba prevista para 2018, es posible que se añadan nuevas modificaciones. La última previsión de entrada en vigor está estimada para el 25 de febrero de 2019.

3.5.4 Serenity.

Se desconocen por completo los detalles que traerá consigo la cuarta fase del *roadmap* de la plataforma Ethereum. Ni siquiera se conoce la fecha estimada del lanzamiento.

Lo único que se conoce es su principal objetivo. La llegada de Serenity supondrá el paso definitivo para modificar por completo el algoritmo de consenso, dejando atrás la PoW y entrando por completo en el "universo" de la PoS.

Si bien es cierto que ya existen implementaciones de PoS en otros protocolos *blockchain*, en Ethereum está aún en fase de pruebas y desarrollo. Hasta que el equipo de desarrollo no considere que ha obtenido una versión fiable y estable del algoritmo, no se propondrá la actualización pertinente.

3.6 Direcciones

En el protocolo Ethereum existen dos tipos de cuentas o direcciones: Las cuentas externas o *externally owned accounts* (EOAs) y las cuentas de contrato o *contract accounts*.

Las EOAs disponen de una clave privada que otorga el control de la cuenta a quien la conozca. Por su parte, las *contracts accounts* están controladas por el propio código. Es decir, un *smart contract* no es más que un determinado código que controla una dirección de contrato.

Las cuentas de Ethereum tienen una longitud de 20 bytes (160 bits) y albergan el estado o resultado de todas las transacciones ejecutadas en relación a la misma. Cuando se procede a crear una nueva cuenta, estos 160 bits corresponden con los 160 bits menos significativos del resultado de aplicar el algoritmo de hash Keecak-256 a una codificación RLP de una estructura de datos compuesta por la *sender_account* y el *nonce* de la misma.

La información de estado de una cuenta puede cambiar en cualquier momento, por lo que solo se puede tomar como cierta en referencia a un determinado estado global. Los cuatro elementos que componen el estado de la cuenta son:

1. ***Nonce***: Asegura que cada transacción se ejecute una única vez. Indica el número de transacciones que ha enviado la cuenta o el número de contratos desplegados por la cuenta asociada a la dirección.
2. ***Ether balance***: Cantidad de Wei que contiene la cuenta.
3. ***Code_hash***: Solo contenido por las *contract accounts*. Hash del código de la EVM que controla la cuenta y se almacena en la *state database*.
4. ***storage_root***: Hash del nodo raíz que identifica al árbol de almacenamiento donde residen los datos asociados al contrato.

(6)

3.7 Estructuras de información.

3.7.1 Bloque.

Como ya se ha explicado anteriormente, un bloque es un conjunto de transacciones construido por un minero con el fin de añadirlo a la cadena de bloques y recibir una recompensa a cambio.

Sin embargo, hasta ahora no se ha profundizado en los bloques del protocolo Ethereum. Para proceder a diseccionar un bloque de la *blockchain* se va a utilizar el bloque número 6892553[28], obtenido mediante la herramienta *Etherscan*[29]:[30]

- *Height*: Corresponde al número de bloque. (***6892553***)
- *TimeStamp*: Marca temporal que identifica el momento de creación del bloque según el formato de hora +UTC. (***29 secs ago (Dec-15-2018 06:05:05 PM +UTC***)
- *Transactions*: Número de transacciones que contiene el bloque. En este caso se diferencian entre las generadas por EOAs y *contract accounts*. (***40 transacciones y 3 transacciones internas de contratos***)
- *Hash*: Número hash que identifica el bloque. (***0x410e255cf396f56ddc71e94b1fca21028441de7c5876b514bc6ace32eecf3622***)
- *Parent hash*: Número hash del bloque predecesor en la cadena. (***0x47ff9b5b7b954f2d1b09f400467ee8e35a0e70742197d59e35b85ce5f811e6c4***)
- *Sha3Uncles*: Número hash asociado al bloque *uncle*. Este tipo de bloques comparten predecesor con el bloque minado, pero no son aceptados por la *blockchain*. Es decir, son bloques válidos, pero no han sido aceptados por la red por cualquier motivo. Los mineros son recompensados al encontrar bloques de este tipo. (***0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347***)
- *Mined by*: Minero encargado de generar el bloque valido y añadirlo a la cadena. (***0x5a0b54d5dc17e0aadc383d2db43b0a0d3e029c4c***)
- *Difficulty*: Complejidad que ofrece la red para conseguir un bloque válido indicada en número de intentos hasta encontrar dicho bloque. (***2,207,446,435,262,961***)
- *Total Difficulty*: Suma de todas los campos *'difficulty'* desde el inicio de los tiempos. (***8,291,731,830,726,925,382,186***)
- *Size*: Tamaño del bloque. (***16891 bytes***)

[28] Bloque número 6892553 puede consultarse en: https://etherscan.io/block/6892553 [consulta 15 de diciembre de 2018]

[29] Etherscan es una herramienta que permite ver el contenido publicado en la blockchain de Ethereum.

[30] ***La información incluida con este formato corresponde al dato real asociado al bloque***

- *Gas used*: Cantidad de gas que ha sido requerido para procesar el bloque. Identifica el gasto computacional que ha sido necesario. Si esta cantidad supera el valor de *'Gas limit'* el bloque aparecería como fallido y no sería añadido. (***7,996,424***)
- *Gas limit*: Cantidad de gas establecida como máxima para ser verificado y/o ejecutado. (***8,000,029***)
- *Nonce*: hash de 64 bits que permite verificar la validez del bloque y que supone el resultado de realizar la *PoW*. (***0xab7918c8071cfda9***)
- *Block reward*: Recompensa que recibe el minero por minar el bloque. En la actualidad está establecida en 3 Ether más el gas utilizado para realizar la actividad de minado. (***3.029948411662739189 Ether (3 + 0.029948411662739189)***)
- *Uncles reward*: Recompensa recibida por minar el bloque *uncle*. (***0***)
- *Extra Data*: Máximo de 32 bytes de información extra que se consideran relevantes para incluir en el bloque.

Si procedemos a analizar el mismo bloque con otra herramienta (*etherchain*)[31] observamos campos adicionales:

- *Lowest Gas Price*: Precio mínimo que se puede asociar al gas de la transacción expresado en *wei*. (***2.2 Gwei***)
- *Root state*: Indica el estado global de la red. Esto es, el resultado de ejecutar todas las transacciones desde el inicio de los tiempos hasta el momento en el que se genera el bloque y las transacciones han sido ejecutadas. (***0x84af6cbd3517d42a70886f14b4e144aaf1b6e061776b1102d8696cb3d5a46a0a***)
- *Bloom Data*: Relativo al *Bloom filter* que se utiliza en Ethereum. (***0x924418c0718b28305e724183000850840 8c8005c24b10484000208e2410480860008ca4609401141192414620134b000a20a1000440541ec20010882db301899004212d142100000d741a1180820074188381f3e60c65629000438321c40580c80314801820844 1080c0b00c00132a80c47069820126838510e101f20c006147c0a2224026224dd02408008d44004430a04020034000831200170e6211408924100000 0104c044289208d8d0850600001040052600 00ea4064164600144240b28070402f08d510242490940800b0a280a4801883243411014010820094 02a70822402c00030010c280028a64053204e3f79000212039280103c830530b048000***)
- *Transactions and Receipts Root Hash*: Hashes asociados con el *Merkle Tree* compuesto por todas las transacciones del bloque generado. (***Receipts: 0xc5a38dc2c97873e659b7d218b85041b4178a79ffe1e2719dca94154f9fef607e ; Transactions: 0xc26332260d314418c0aeeea793a989477568fdc57b21f06240781659e4f954c9***)

[31] Etherchain es otra herramienta que nos permite explorar la información contenida en la blockchain de Ethereum. El bloque que estábamos analizando puede consultarse en: https://www.etherchain.org/block/6892553 [consultado el 15 de diciembre de 2018]

Una vez descritos cada uno de los elementos que conforman la cabecera del bloque, quedan pendientes dos puntualizaciones.

Si acudimos al *yellowpaper* podremos comprobar que algunos campos reciben otro nombre, aunque son muy parecidos, por lo que no merece la pena detenerse en ello, a excepción de uno. Las referencias a los bloques *uncle* se realizan mediante una alternativa neutral en términos de género, *ommer*.

Los bloques *ommer* puede que no se hayan añadido a la cadena de bloques porque mientras se intentaba superar la PoW, algún minero haya añadido alguna de las transacciones contenidas en el *ommer block* en un bloque ya validado y, por consiguiente, no se puede volver a añadir, quedando el bloque invalidado.

La tasa de generaciones de bloques de Ethereum (*Blocks average per hour*) está en torno a los 245 bloques por hora[32]. Por este motivo, se puede intuir que la creación de *ommer blocks* sucede a menudo y es precisamente por eso por lo que Ethereum también recompensa a los mineros cuando consiguen generar un *ommer block* valido, para recompensar la carga computacional consumida por el minero.

Por otro lado, los filtros de Bloom o *Bloom filters* son estructuras de datos probabilísticas eficientes en el espacio. Se utilizan para saber si un elemento pertenece a una colección o no. En Ethereum, se utilizan para almacenar información relativa al resultado de ejecutar una transacción. El funcionamiento se describe a continuación.

Un *Bloom filter* vacío es un array de m bits, todos ellos inicializados con valor cero. Para un correcto funcionamiento debe haber definidas k funciones de hash que asignan un elemento definido a una posición del array. Típicamente, m es proporcional al número de elementos que se van a añadir y k es una constante mucho menor a m.

Para añadir un elemento, se utiliza el propio elemento como input de cada una de las k funciones de hash definidas. El resultado son k posiciones del array, cuyo valor se modifica a uno.

Para comprobar si un elemento pertenece a la colección, se utiliza el elemento como input de las k funciones de hash y se comprueba que las k resultantes posiciones del array tienen valor uno. En caso contrario el elemento no pertenece a la colección.

Es importante destacar que pueden darse falsos positivos y ni se puede borrar un elemento de este tipo de estructuras, ni se pueden relacionar dos elementos dentro de la estructura.

[32] Referencia consultada en https://bitinfocharts.com/ethereum/ el 29 de diciembre de 2018.

Por último, y tras describir la composición de un bloque, es importante describir a alto nivel el proceso que se ha de seguir para construirlo/verificarlo:

1. Determinar/validar *ommers*.
2. Determinar/validar transacciones.
3. Aplicar recompensas.
4. Calcular un estado global valido y un *nonce* de bloque/verificarlo.

3.7.2 Transacciones.

Existen dos tipos de mensajes en la *blockchain* de Ethereum, aquellos que deben registrarse en la cadena de bloques y aquellos que no.

Una transacción (*transaction*) es un mensaje firmado digitalmente que supone alguna acción relacionada con la cadena de bloques. Se diferencian tres tipos de transacciones, aquellas en las que se transfieren unidades monetarias o *tokens* de una cuenta a otra, el despliegue de un *smart contract* y la ejecución de una función dentro de un *smart contract* que modifique el estado global.

Un mensaje, por el contrario, es un conjunto de datos que se envían por código de la EVM de una cuenta a otra, pero que no produce un cambio en el estado de la red y por consiguiente no debe ser incluido en la *blockchain*.

El mismo procedimiento que se ha realizado anteriormente con un bloque, se va a realizar con una transacción perteneciente el mismo bloque. Si bien es cierto que, en esta ocasión no se explicarán los campos que ya hayan sido explicados para no ser repetitivo. En este caso se ha elegido la transacción que se identifica por el siguiente hash: 0xe2df174fe990590cb150a5663f89f3c8ff85cfdeada0d77561fab28d25276e9f[33]:

- *TxHash*: Hash identificativo de la transacción en cuestión. (***0xe2df174fe990590cb150a5663f89f3c8ff85cfdeada0d77561fab28d25276e9f***)
- *TxReceipt Status*: Indica si la transacción se ha ejecutado correctamente (*success*) o no (*failed*). (***Success***)

[33] La información relativa a la transacción puede consultarse en los siguientes enlaces:
Etherscan:
https://etherscan.io/tx/0xe2df174fe990590cb150a5663f89f3c8ff85cfdeada0d77561fab28d25276e9f
Etherchain:
https://www.etherchain.org/tx/e2df174fe990590cb150a5663f89f3c8ff85cfdeada0d77561fab28d25276e9f
[Consultado el 15 de diciembre de 2018]

- *Block Height*: Bloque al que pertenece la transacción. Entre paréntesis pueden aparecer el número de confirmaciones que ha recibido el bloque. (***6892553 (599 Block Confirmations)***)
- *TimeStamp*: (***2 hrs 18 mins ago (Dec-15-2018 06:05:05 PM +UTC)***)
- *From*: Este campo indica la cuenta que supone el origen de la transacción. (***0xf86c2d425521bd1f7c15e193c7d1c676ab38f9cf***)
- *To*: Cuenta destino. (***0x347072fcd71a9e251d18a292d9bbc8bc3149cd8a***)
- *Value*: Cantidad de ether transferida. (0.0002423845 ether)
- *Gas Limit*: (***21000***)
- *Gas used by Trasanction*: (***21000***)
- *Gas Price*: (***2.2 Gwei***)
- *Actual Tx Cost/Fee*: Coste total de procesar la transacción. Su valor se obtiene de multiplicar el *'gas used by transaction'* por el *'gas price'*. (***0.0000462 ether ($0.003876)***)
- *Nonce*: (***3***)
- *Input Data*: En este campo se puede incluir bien el código compilado del *smart contract* a desplegar o bien almacenar alguna información relacionada con llamadas a funciones de un *smart contract*. En este caso, al tratarse de un envío monetario, el contenido es nulo.
- *Conflicting Tx*: Contiene posibles irregularidades relacionadas con la transacción. (***We found 1 transaction with the same nonce from the same account: 0x1cba21d5d336720a0fff81c89f7b789bbdeab2190a1ce35c8f194622eda05534***)

En los diferentes documentos que describen el funcionamiento y las especificaciones de Ethereum (*whitepaper* y *yellowpaper*) se realiza una especificación en lo que a la nomenclatura y contenido del campo *Input Data* se refiere, que conviene destacar.

Si la transacción es una *message call*, el campo debe denominarse "*data*" y contener los datos de entrada. Sin embargo, en caso de tratarse de una creación de contrato el campo se denomina "*init*" y debe contener el código de la EVM que creará el contrato. Esta segunda opción devuelve un fragmento de código denominado *body*, que supone el código que se ejecutará en futuras llamadas al contrato recientemente creado.

3.8 EVM.

En muchas publicaciones se habla de Ethereum como un "ordenador universal" y no es una casualidad. Todos los usuarios pertenecientes a la red Ethereum tienen acceso a toda la información que circula por ella y puede hacer libre uso de ella. Ocurre exactamente igual con la EVM, que no es más que un componente básico en el protocolo. A continuación,

se profundizará explicando el funcionamiento de la *Ethereum Virtual Machine* en el porqué de esta asociación.

En Ethereum, las transacciones se definen como scripts, que deben ser ejecutados para hacer efectivas la transacciones. Por consiguiente, cuando una transacción se añade a la cadena de bloques, se deben ejecutar las instrucciones que incluye, desde el despliegue de un *smart contract* hasta el envío de *ethers*.

Como profundizaremos más adelante, el lenguaje de programación más usado en el protocolo Ethereum es "*Solidity*". Este lenguaje tiene una sintaxis de alto nivel que debe ser traducido a lenguaje máquina para que la computadora pueda interpretarlo.

Sin embargo, dependiendo del sistema operativo y la arquitectura de la computadora encargada de compilar el código asociado a la transacción, el archivo en lenguaje máquina es diferente. Es decir, una vez compilado el mismo código en una máquina con sistema operativo Windows, se obtiene un fichero diferente al generado por una computadora Linux.

Una solución ante este problema es la implementación de una máquina virtual, que permite la portabilidad del código compilado entre máquinas y supone un incremento en la robustez del sistema, ya que pueden realizarse pequeñas comprobaciones durante la ejecución para alertar de posibles errores.

En el caso del protocolo Ethereum, se utiliza la *Ethereum Virtual Machine*, que ofrece varios compiladores, cada uno optimizado para procesar un lenguaje de programación, e hilos de ejecución. El motivo de que existan varios compiladores es debido a que los lenguajes de programación para *smart contracts* son de alto nivel y la EVM solo ejecuta *bytecodes* (lenguaje máquina).

3.8.1 Arquitectura.

La EVM tiene una arquitectura *stack-based*. Para poder profundizar en su funcionamiento, hay que detenerse primero a explicar cómo se gestionan los datos. Se diferencian al menos cuatro tipos de datos: *stack*, *calldata*, memoria (*memory*) y almacenamiento (*storage*).

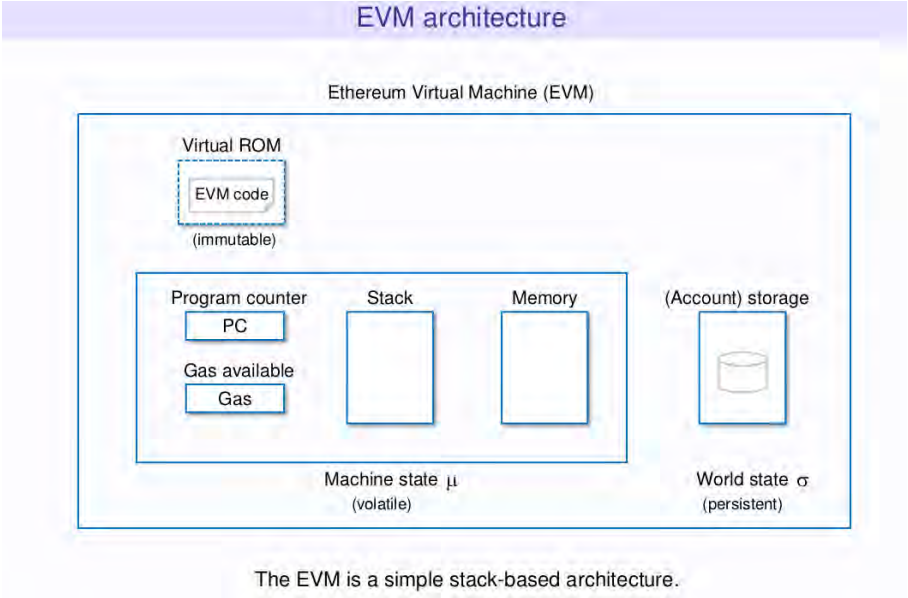


Fig. 7. EVM architecture.
Tomada de: **(8)**

Que la EVM sea una ***stack*** *machine* implica que no se utilizan registros, sino una *stack* virtual. Para facilitar el esquema del algoritmo de hash *Keecak256*, la máquina se implementa con una arquitectura de datos de 256-bit. La memoria de la pila tiene un tamaño máximo de 1024-bits. La mayoría de los *opcodes*[34] acuden a esta memoria para tomar sus parámetros.

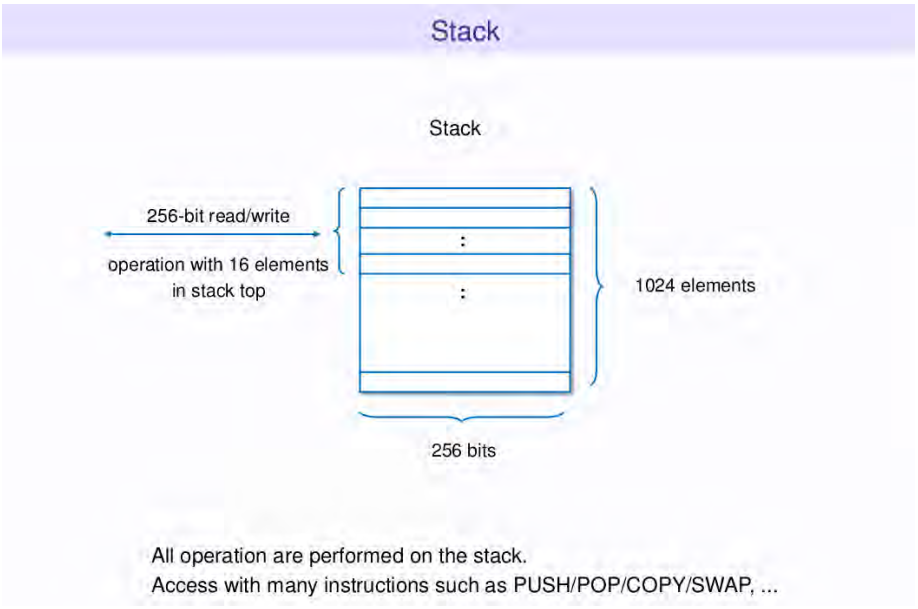


Fig. 8. Stack architecture.
Tomada de: (8)

Para almacenar los parámetros de una transacción se utiliza un espacio de solo lectura que permite ser accedido a nivel de byte y que recibe el nombre de ***calldata***. Esto

[34] En el ANEXO A pueden consultarse los *opcodes* definidos en la EVM.

supone por consiguiente, que cuando se quiere consultar alguna información, debe referenciarse respecto a un byte inicial y una longitud.

La ***memoria (memory)*** es volátil y se utiliza para almacenar información durante la ejecución y para pasar argumentos a funciones. Se puede rellenar a nivel de byte, pero únicamente puede leerse por bloques de 256-bit.

El coste de incrementar la memoria se incrementa linealmente durante los primeros 724 bytes y cuadráticamente a partir de entonces. Además, los primeros 64 bytes se reservan para memoria interna de la propia EVM, para dar soporte a Solidity.

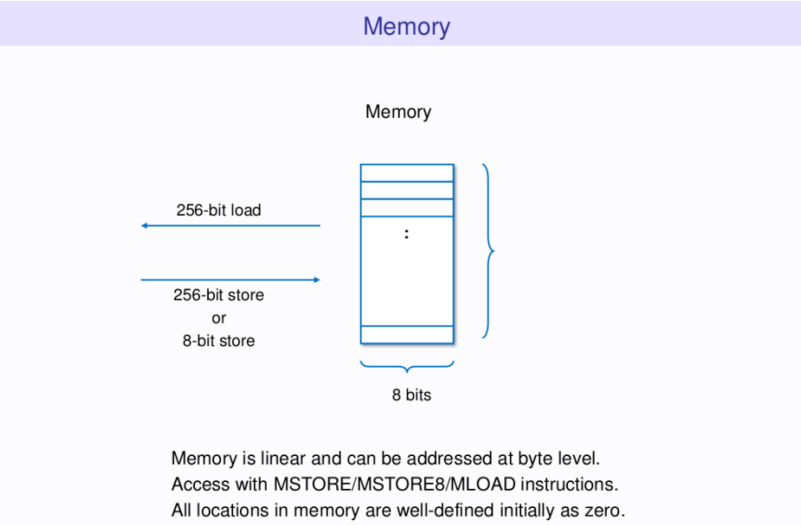


Fig. 9. Memory architecture.
Tomada de (8)

El ***almacenamiento (storage)*** se utiliza para guardar datos de forma permanente, en todos los nodos pertenecientes a la red Ethereum. Se compone de 2^{256} contenedores de 256-bits con formato clave-valor (*key-value*). Tanto la lectura como la escritura se realizan a nivel de contenedor. Los contratos pueden únicamente interactuar con su propio almacenamiento.

Interactuar con este componente puede conllevar unos gastos muy elevados, por lo que conviene tener claro cuando es realmente necesario acudir a él. Tanto este espacio como los anteriores se inicializan con todas las posiciones como cero. Modificar una de esas posiciones para escribir un dato, en este componente, implica un gasto de 20.000 gas.

Por ese motivo, cuando una transacción incluye la necesidad de escribir en el *storage* se deben incluir una tasa equivalente al coste de todas las lecturas/escrituras asociadas a la ejecución de la transacción.

Por otro lado, para incentivar la minimización del uso del almacenamiento, cuando una posición contiene un dato y se modifica para otorgarle el valor inicial, se ofrece una recompensa de 15.000 gas.

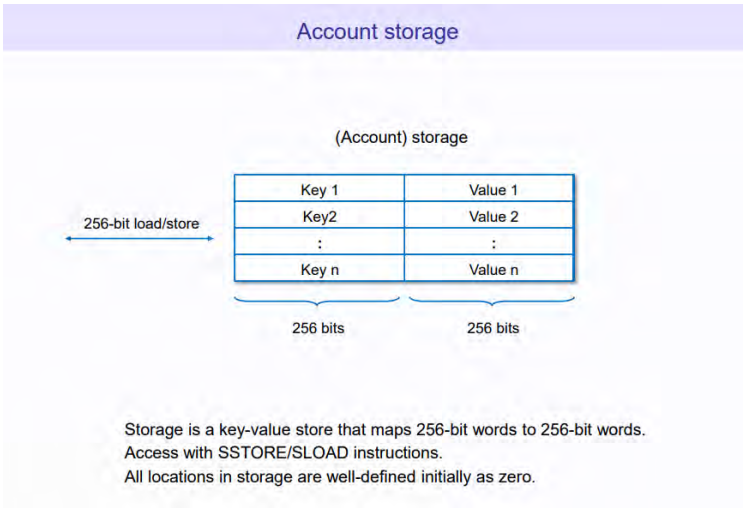


Fig. 10. Storage architecture.
Tomada de: (8)

Respecto a la arquitectura de la EVM, el último componente que requiere mención especial es la ***Virtual ROM***, donde se almacena el código del programa. Para poder interactuar con este componente existen una serie de instrucciones específicas.

3.8.2 Funcionamiento.

Para poder ejecutar una transacción debe realizarse primero una validación intrínseca compuesta por los siguientes puntos:

1) El RLP de la transacción está bien construido.
2) La firma digital de la transacción es válida.
3) El *nonce* de la transacción es igual al contenido en la cuenta del emisario.
4) El *gas_limit* es mayor o igual que el *intrinsic_gas*.
5) La cuenta del emisario puede afrontar el coste de ejecutar la transacción.

Cuando se ejecuta una transacción, la información del resultado se almacena en el *tx_receipt* (*transaction receipt*) junto a los *logs* generados durante la propia ejecución (*logs_set*), el filtro de Bloom asociado a dichos *logs* (*logs_bloom*), el estado post-transacción y el gas intrínseco. Por consiguiente, el *tx_receipt* es una recopilación de todo lo que genera la ejecución de la transacción.

Una vez que se supera la primera fase de validación intrínseca, se incrementa el *nonce* de la cuenta origen y se resta el gas intrínseco, o cantidad de gas requerida para ejecutar la transacción, del balance.

Durante la ejecución se genera un subestado asociado compuesto por una tupla de información: el conjunto de autodestrucción, compuesto por aquellas cuentas que deben ser destruidas tras la finalización, un conjunto de *logs_series* que facilitan la monitorización de la ejecución y el balance de recompensa.

Para que la transacción pueda ser ejecutada, el usuario debe proporcionar una serie de información a modo de argumentos:

- Dirección de la cuenta a la que pertenece el código.
- Dirección de la cuenta que solicita la ejecución.
- Precio por gas establecido para la transacción.
- Dirección de la cuenta que provoca que el código se ejecute (si el agente de ejecución es una transacción este campo llevará la información del *transaction sender*).
- Valor (*wei*) enviado a la cuenta (si se trata de una transacción será el *transaction value*).
- Código a ejecutar en forma de array de bytes.
- Cabecera del bloque al que pertenece.
- Profundidad de la *contract-creation* o *message-call stack*. Esto significa el número de llamadas recursivas. Es decir, si dentro de la transacción hay una llamada a un contrato, la profundidad de los *opcodes* ejecutados en esa llamada pasaría a ser 2. Si esa llamada implicara nuevas llamadas, estás últimas tendrían profundidad 3 y así sucesivamente.
- Permiso para realizar modificaciones en el estado global.

De esta forma, el modelo de ejecución define una función de estado de transacción que calcula el estado final, la cantidad de gas restante, el subestado acumulado y la salida resultante.

Sin embargo, todo esto es una mera introducción para poder explicar *per se* la función de ejecución. Esta *execution function* es de tipo iterativa, estudiando en cada ronda el estado global (*world state*) y local (*machine state*).

El estado local se define mediante una tupla de datos compuesta por el gas disponible, el contador del programa, una serie de ceros de tamaño 2^{256} o contenidos de memoria, el número de palabras activas en la memoria y el contenido de la pila.

Por consiguiente, en cada una de las rondas del proceso iterativo, se extrae o añade un elemente de la pila (*stack*), bajo el principio *left-most*, se resta el coste de ejecutar dicha instrucción del gas disponible y, en la mayoría de los casos, se incrementa el contador del programa.

Bien sea durante o al final de la ejecución, si sucede alguna excepción, se eliminan los elementos de la *stack* asociados a la ejecución, se detiene la ejecución del script y se descartan los cambios. (12)

3.8.2.1 Message Calls.

Un mensaje de llamada puede tener como origen una transacción o bien la propia ejecución del código de un contrato.

La actividad computacional de un mensaje de llamada puede que no se ejecute durante el estado presente debido a que, si una cuenta se está creando y todavía no dispone de código desplegado, aunque reciba una llamada, hasta que no disponga del código no podrá procesar la llamada.

Para poder ejecutar las transacciones del tipo *message calls* se requiere la siguiente tupla de información:

- *sender account.*
- Transacción origen.
- Beneficiario.
- Cuenta cuyo código hay que ejecutar.
- Gas disponible.
- Valor a transferir.
- Precio del gas.
- Un array de bytes de longitud arbitraria.
- *Input data* de la llamada.
- La profundidad de la pila de llamadas de mensajes o creaciones de contrato.
- Permiso para modificar el estado global.

3.8.2.2 Contract Creation.

Para proceder a la creación de un *smart contract* se requiere generar una transacción con valor *nothing* en el parámetro *to* y el código asociado al contrato en el campo *init*. De esta forma, la EVM lo ejecutará, grosso modo, según se ha descrito anteriormente e iniciará la cuenta.

Sin embargo es conveniente entrar en detalles y describir los parámetros de entrada y salida, así como algún error que puede suceder.

Como parámetros de entrada se tienen los siguientes:

- *sender account.*

- *original transactor*, que suele coincidir con la cuenta del emisor, pero puede diferir si en vez de proceder de un transacción, procede de una ejecución de la EVM.
- Gas disponible.
- Precio del gas.
- Valor a transferir si la creación resulta exitosa (*endowment*).
- Código EVM para inicializar el contrato.
- La profundidad de la pila de llamadas de mensajes o creaciones de contrato.
- Permiso para modificar el estado global.

Es muy importante que la cantidad de gas disponible indicada sea suficiente para poder ejecutar el código de inicialización del contrato, en caso contrario la creación se detendrá y se revertirán todos los cambios que se hayan podido realizar.

Si la *contract_creation* es exitosa, debe pagarse una tasa final de creación que es proporcional al tamaño del código de *body*. Como se explicó anteriormente, *body* hace referencia a un fragmento de código resultante de la ejecución del código de inicialización, y es ese el código que se ejecuta cuando se invoca al contrato.

De igual manera, si el contrato se crea, se resta el *endowment* al balance de la cuenta emisora. (12)

La tupla de datos de salida en caso de éxito es la siguiente:

- Nuevo estado global.
- Gas disponible (tras la creación).
- Subestado acumulado.
- Mensaje de error.
- Información del contrato:
  - *nonce* = 1
  - balance = *endowment*
  - *storage* = Vacío
  - *code hash* = KEC[35](string vacío)

3.8.2.3 Excepciones

Durante la ejecución en la EVM de los diferentes *opcodes* que forman un código, pueden darse una serie de excepciones que paralizan automáticamente el proceso:

- *out-of-gas* (oog): Implica que el gas necesario para ejecutar el código es superior al valor inicial de gas disponible. Por consiguiente, en algún

[35] Notación para referenciar a la función de hash Keecak-256

momento no habrá suficiente gas para ejecutar el siguiente *opcode* y se detendrá el proceso.

- *Invalid_jump_destination*: Si el destino referido al ejecutar el *opcode* *JUMP* o *JUMPI* no es válido.
- *Invalid_instruction*: El *opcode* o instrucción no está definido.
- *stack_underflow*: Esta excepción sucede si la pila se queda sin elementos suficientes.
- Se produce otra excepción cuando se modifica el tamaño de la pila y se le otorga un valor superior a 1024bits

Si bien es cierto, estas excepciones no suceden durante la ejecución del *opcode* en cuestión. Son fallos generados al procesar los parámetros de entrada de la siguiente instrucción.

4 *SMART CONTRACTS.*

La tecnología *blockchain* surgió como una solución para descentralizar la autoridad de una red sin el problema del doble gasto. Como ocurre con cualquier tecnología y/o solución revolucionaria, los escépticos la ponían en duda.

Se trata de una tecnología que apareció hace tan solo una década, por lo que aún quedan muchos detalles por perfeccionar y otras tantas aplicaciones por descubrir. Sin embargo, hay que poner en valor algunos de los grandes avances que se han conseguido.

Las características más relevantes de las redes *blockchain* son la descentralización, tanto de la autoridad como de la información así como su inmutabilidad y transparencia. Todos estos conceptos son esenciales para hablar de *smart contracts*.

Como se ha comentado en capítulos anteriores, un *smart contract* no es más que un *script* con unas líneas de código. Por consiguiente, que los *smart contracts* sean descentralizados implica que el código es descentralizado.

El concepto de código descentralizado hace referencia a que ese *script* se almacena de manera idéntica en multitud de ordenadores y se ejecuta de la misma forma en todos ellos, mediante la ya explicada *Ethereum Virtual Machine*. Esto provoca que la red *blockchain* sea una red determinista, en el sentido de que todas las funciones que se ejecuten en ella, deben producir la misma salida para la misma entrada cuando se procesen.

El código se despliega, siempre, como parte de una transacción, por lo que no se puede modificar, es inmutable, y al haberse publicado en la *blockchain*, puede ser consultado por cualquier usuario, siendo totalmente transparente.

Esto puede generar ciertas suspicacias en referencia a la seguridad y validez de los resultados. Sin embargo, al ser parte de una transacción, todos los mineros deberán validar en primer lugar el bloque en el que se incluye la transacción que contiene el *script* y, tras ello, deben ejecutar todas las operaciones contenidas en él para actualizar los estados de cuentas pertinentes. Por consiguiente, todo *smart contract* es ejecutado y validado por todos los nodos mineros de la red, otorgándole validez al mismo.

Sin embargo, antes de continuar entrando en detalles con los *smart contracts*, es necesario detenerse a entender el concepto de su análogo al español y de la evolución histórica del concepto *per se*.

El concepto contrato inteligente no existe como tal en español, por lo que se va a definir la referencia más cercana, contrato. Un contrato se define, según la Real Academia

Española, cómo "pacto o convenio, oral o escrito, entre partes que se obligan sobre materia o cosa determinada, y a cuyo cumplimiento pueden ser compelidas."[36]

Es decir, un contrato es un acuerdo entre dos o más partes, en el que se definen unas condiciones a cumplimentar. Si todas las partes lo aceptan, se comprometen a hacer efectivas las acciones pertinentes para cumplir las condiciones acordadas. Si bien es cierto, es necesario destacar que este contenido es interpretable, lo que supone en muchos casos disputas post-contractuales para clarificar qué parte tiene razón.

El concepto de *smart contract* surgió en 1994 gracias al criptógrafo Nick Szabo, quién en 1997 publicó un post de perfil técnico en el que explicaba el concepto de *smart contract*.

4.1 Nick Szabo.

En todos los sitios, incluido el presente documento, se habla de *blockchain* como una tecnología que revolucionará nuestras vidas, modificando considerablemente el escenario empresarial, al suprimir los intermediarios en muchos campos. De igual manera se menciona que es una tecnología que aún tiene mucho por hacer para ser considerada como madura y consolidarse.

Sin embargo, conviene hacer un parón y echar la vista atrás para darse cuenta de que las ideas que conforman la filosofía de *blockchain* ya se habían publicado, en su mayor parte en 1997. Nick Szabo publicó el 1 de septiembre de 1997 un artículo titulado "*Formalizing and Securing Relationships on Public Network*". A continuación, se van a extraer las ideas más relevantes, entre las que se incluye el concepto de *smart contract*.

En 1997, pronostica que en un corto periodo de tiempo los *smart contracts* transformarían los procedimientos empresariales y generarían nuevos modelos y estructuras de negocio. No específica a qué se refiere con un corto periodo de tiempo, pero a tenor del panorama tecnológico actual, no puede negarse que al menos en el planteamiento era un adelantado a su tiempo.

En esta publicación se define el contrato como una serie de cláusulas que ponen de manifiesto el acuerdo entre dos o más partes y que requiere de un tercero para otorgarle validez.

Se introduce el concepto de *smart contract* como un conjunto de cláusulas contenidas en *hardware* o *software*, de tal forma que se penalice considerablemente a quien

[36] Se puede consultar la definición en el siguiente enlace: https://dle.rae.es/?id=AdXPxYJ

incumpla el contrato. Estos *smart contracts* permitirían eliminar terceras partes, permitiendo una relación directa entre las dos partes interesadas en el contrato.

Si bien es cierto, si contempla unas terceras partes necesarias. Unas terceras partes que le proporcionen información a los *smart contracts* en ocasiones especiales. No acuñó un nombre para estos intermediarios, pero como en más adelante se va a detallar, esto se corresponde en la tecnología *blockchain* con los oráculos.

Sin embargo, advierte que, en esos momentos, ni los ordenadores tienen la capacidad de ejecutar los algoritmos necesarios para proteger y ejecutar los *smart contracts*, ni la red soporta la transmisión de mensajes suficientemente sofisticados en un tiempo útil.

De hecho, Nick Szabo, para clarificar el concepto que está introduciendo explica un posible caso de uso. Un sistema de venta de vehículos financiados. Esto es, un usuario decide comprarse un automóvil y estipula con el vendedor el pago de un determinado importe durante x mensualidades. El sistema asume una llave con conexión a internet para estar en contacto con el *smart contract*.

La llave, metafóricamente, estaría compuesta de dos puertas. Una puerta delantera que daría acceso al control de la llave al comprador y una puerta trasera que otorgaría el control al vendedor.

Mientras los pagos se hagan efectivos en las fechas acordadas, la puerta delantera permanecería abierta y la trasera cerrada, de manera que solo el comprador tuviera el control del automóvil. Sin embargo, si el comprador se retrasara en sus cuotas o suspendiera su pago, la puerta delantera se cerraría y la trasera se abriría, devolviendo el control al vendedor.

Una vez abonado los importes acordados en las fechas correspondientes, la puerta trasera se cerraría de por vida, haciéndose efectivo el traspaso de la propiedad del vehículo.

Toda esta metáfora, tendría un elemento subyacente que controlaría la cerradura de ambas puertas, el *smart contract*. Pese a ser un mero planteamiento teórico, el ejemplo denota que Nick tenía muy claro el potencial de su idea.

Sin embargo, no se detiene en ese punto. Nick Szabo expone la crucial importancia de los algoritmos criptográficos para:

- Aseverar la integridad de la información
- Poder eliminar la necesidad de un tercero que otorgue la confianza al sistema, ya que “hace mucho que se ha reconocido que un intermediario genera más confianza cuando es distribuido”

- Proporcionar mayor trazabilidad de la información y facilitar tareas relacionadas con la auditoría.
- Implementar sistemas de seguridad proactivos (preventivos), a diferencia de los existentes en la época, todos ellos reactivos.

Analizando con más detalle algunos rasgos necesarios e importantes a la hora de diseñar un contrato en el entorno de los *smart contracts*, se destacan las siguientes propiedades:

- Los *smart contracts* deben ser **transparentes para los implicados** en el mismo. Es decir, cada una de las partes debe tener la capacidad tanto de comprobar si el resto están cumpliendo debidamente sus obligaciones, como de poder demostrar que han cumplido su propia parte.
- Las partes no involucradas en el *smart contract* deben poder **verificar** si el contrato ha sido incumplido por alguna de las partes.
- La información relativa a los contratos debe ser **privada** y únicamente conocida por las partes implicadas.

Todas las consideraciones las relaciona, dejando claro que es el pilar básico sobre el que deben sostenerse los *smart contract*, con la criptografía. El entorno de los contratos inteligentes debe contener sistemas de criptografía asimétrica, tanto para realizar firmas digitales, como para proteger la información.

Por último, es necesario resaltar que, menciona la posibilidad de implementar una máquina virtual para ejecutar las mismas acciones en diferentes máquinas, aunque desecha su propia idea, entre otras cosas porque algunos usuarios recibirían la información antes que otros.

Pese a que el mismo descartara la solución en ese momento, planteaba una idea muy interesante. De hecho, se puede decir que cometió un error en su previsión, ya que como se ha explicado en un capítulo anterior, la EVM es de vital importancia en el protocolo Ethereum. (3)

4.2 El concepto de *smart contract* en Ethereum.

Las similitudes entre el concepto de *smart contract* que proponía y preveía Nick Szabo y el que finalmente se ha implantado en la tecnología *blockchain* son mayoría respecto a sus equivocaciones.

Actualmente se entiende por *smart contract* un conjunto de reglas y decisiones que se definen mediante un *script* de código y que son tomadas autónomamente por el propio código, en función de la información que recibe. De igual manera, es posible que el *smart*

contract dependa de algún dato *extra-blockchain*, por lo que es necesario implementar soluciones para los oráculos.

Debido a que estos fragmentos de código se almacenan en la cadena de bloques como parte de una transacción, su contenido está cifrado y únicamente es conocido por las partes directamente implicadas, se ejecutan de manera autónoma, se verifican constantemente al ser ejecutados en la EVM, lo que también implica que son deterministas, y permiten desintermediar la relación entre las partes interesadas.

A diferencia de los *smart contracts* de la archiconocida red Bitcoin, el protocolo Ethereum permite crear entre otros:

- Contratos multifirma: en el entorno de critpoactivos implica que los fondos almacenados en la dirección asociada al *smart contract* solo pueden retirarse en el caso de que la mayoría de los firmantes estén de acuerdo.
- Contratos que requieren dobles depósitos: el *smart contract* obliga a cada una de las partes a entregar el doble de la cantidad requerida para cumplimentar su parte a nivel económico. Si el contrato depende de alguna condición adicional, los fondos permanecerán en la dirección del contrato hasta que se satisfaga la cláusula en cuestión.

  En el supuesto de que cada uno de los implicados cumpla con su parte, los fondos extras depositados se devolverán a las direcciones pertinentes. Sin embargo, si alguna cláusula se incumple, todos los fondos son automáticamente redirigidos a una dirección que no permita extraer los fondos. Es decir, de alguna manera destruye esos fondos.

Es conveniente reseñar las diferencias más relevantes entre los contratos tradicionales y los *smart contracts* de la tecnología blockchain.

- El lenguaje de programación no es natural. Aunque se requieran conocimientos técnicos para poder comprenderlos, esto está directamente relacionado con la siguiente circunstancia,
- Los *smart contracts* no admiten interpretación. Los contratos tradicionales se escriben de tal forma que dan lugar a ambigüedades y requieren de un conocimiento técnico elevado para comprender realmente las posibles interpretaciones. Los *smart contracts* son un conjunto de líneas de código, por lo que, si se entiende el lenguaje, se entiende la operativa del contrato.
- En los contratos tradicionales se requiere de un tercero, como pueda ser un notario, para otorgar validez al mismo. Sin embargo, los *smart contracts* eliminan ese intermediario. Esto implica que la responsabilidad legal en caso de fraude no está muy clara. Es un asunto pendiente de resolver en el futuro.

4.3 Ventajas y desventajas de los *smart contracts*.

Una vez entendidas las diferencias entre el concepto y las implicaciones de los contratos tradicionales y los *smart contracts*, se pueden extraer las ventajas e inconvenientes fundamentales que ofrecen los contratos inteligentes.

⊕ Autonomía: No requieren de un tercero para ejecutarse, lo que implica:

⊕ Reducción de costes y aumento de velocidad: La ejecución de un *smart contract* elimina los trámites burocráticos, por lo que la toma de decisión es cuasi-inmediata y los costes son mínimos.

⊕ Mayor seguridad: Los *smart contracts* son inmutables, descentralizados y están cifrados al residir en la *blockchain*.

⊗ Inmutabilidad de la información: Como puede observarse la inmutabilidad es un arma de doble filo. Es un aspecto positivo porque la información se registra de por vida y es un inconveniente precisamente por lo mismo. Si se publica alguna información confidencial, esta no podrá ser eliminada.

⊗ Un *smart contract* mal programado puede ser susceptible de recibir ataques maliciosos que ejecuten un "robo legal" aprovechando pequeños *bugs* del *script*.

4.4 Casos de uso.

Una vez el presentado y explicado el concepto de *smart contract* es interesante poner el foco sobre los posibles casos de uso que pueden ofrecer estos contratos y algunos ejemplos en cada caso.

- Banca: Mediante *smart contracts* se pueden implementar servicios financieros como puedan ser los préstamos o depósitos de fondos. Un ejemplo es *omiseGO*[37], una DApp que permite transacciones financieras como pagos, depósitos salariales, etc.
- Servicio sanitario: Es posible diseñar un registro de la identidad del paciente que contenga todo su historial clínico actualizado y en el que se limite el acceso mediante un contrato inteligente. Es decir, que sea el usuario quien decida qué fuentes pueden consultar sus datos y cuáles no. Un ejemplo es *Helix*[3],[38] que propone descentralizar la información clínica de los pacientes ofreciéndoles a los mismos la posibilidad de decidir quién puede consultar su historial.
- Sector seguros: Al ser posible eliminar a los intermediarios, es factible implementar un *smart contract* que tipifique de manera clara y concisa las diferentes casuísticas y remuneraciones a ofrecer por la dirección que actúe como aseguradora y el pago

[37] Sitio web de omiseGo: https://omisego.network/

[38] Sitio web de Helix[3]: https://www.helix3.co/

que deba efectuar la dirección asegurada. Un ejemplo es *Etherisc*[39], una compañía que ofrece seguros ante retrasos en los horarios de algunas compañías aéreas o frente a desastres naturales como huracanes.

- Cadenas de suministro: Mediante contratos inteligentes se puede acelerar la cadena de suministro de una entidad. Actualmente, intervienen diversos intermediarios en el proceso. Cada uno de ellos, realiza una serie de actividades burocráticas que ralentizan la cadena de suministro. Un *smart contract* puede eliminar esta burocracia y agilizar sustancialmente el proceso, todo ello unido a un registro completamente trazable de las distintas etapas que hayan compuesto la cadena de suministro. Un ejemplo es el *Commonwealth Bank of Australia*, que realizó una exitosa prueba en relación a una cadena de suministro de ámbito global[40].
- Propiedad intelectual: El acceso a una publicación puede estar regulado por un *smart contract*, de manera que cuando un usuario quiera acceder a ella deba efectuar un micropago destinado al creador de la misma. Un ejemplo es *Vezt*[41], quien hace realidad esto mismo aplicado al mundo de la música.
- IoT: En el ámbito de *Internet of Things* existen infinidad de aplicaciones, desde la implementación de un sistema que controle el *stock* de un comercio hasta la implementación de sistemas de seguridad ante piratería en los *wearables* tan extendidos hoy en día. Un ejemplo es *Vechain*[42], que propone un ecosistema de comunicación para dispositivos IoT.
- *Blockchain*: Creación de nuevos proyectos en forma de *token*. Algunos ejemplos son *omiseGO* y *Vechain*, que en concreto operan gracias al *token* ERC20[43].

4.5 *Tokens*

Durante la introducción al documento, ya se introdujo el concepto de *token per se*. Sin embargo, en el protocolo Ethereum juega un papel clave para contribuir a la denominación de "ordenador universal" y por ello, merece un apartado en el que profundizar.

Ethereum se presenta más que como un sistema con una finalidad única, como pueda ser Bitcoin como un sistema financiero sólido y seguro, como una plataforma. Como

[39] Sitio web de *Etherisc*: https://etherisc.com/
[40] Se puede profundizar con más detalle en la siguiente publicación: https://www.commbank.com.au/guidance/newsroom/commonwealth-bank-completes-new-blockchain-enabled-global-trade--201807.html
[41] Sitio web de *Vezt*: https://www.vezt.co/
[42] Sitio web de *Vechain*: https://www.vechain.org/technology#iot
[43] Se pueden consultar una gran cantidad de soluciones basadas en el *token* ERC20 en la web: https://eidoo.io/erc20-tokens-list/

los cimientos de un nuevo ecosistema, aún por explotar. Las dos claves que otorgan solidez a esa propuesta son la máquina virtual de Ethereum y la manera de llevar a cabo la tokenización.

Los *tokens* nativos son aquellas criptomonedas que pertenecen a las *blockchains* "originales". Los ejemplos más sonados son bitcoin (Bitcoin) y ether (Ethereum). Sin embargo, en Ethereum se plantea la posibilidad de instanciar *tokens* que operen sobre la propia red de protocolo.

Estos *tokens* no son otra cosa que *smart contracts* ejecutándose continuamente sobre la *blockchain* de Ethereum y funcionan como soporte de *blockchains* paralelas. Esto supone por consiguiente, que todas las transacciones asociadas a esa cadena de bloques paralela se van a ejecutar en la EVM, utilizando como *ether* como gas. Es importante destacar, que el valor del nuevo *token* no depende del valor del *ether*.

4.5.1 ERC-20.

Vitalik Buterin propuso la implementación de un estándar para la creación de los *tokens* y la idea fue muy bien recibida por la comunidad. Finalmente, se publicó el 19 de noviembre de 2015 y validó el estándar ERC20.

Como principales ventajas de la utilización de estos *tokens* podemos encontrar que facilita el traspaso entre diferentes *wallets* o Dapps de aquellos *tokens* creados a partir del mismo estándar y que reduce la complejidad de comprensión del ecosistema, es decir, al compartir estos *tokens* las funciones y eventos básicos, se puede predecir de forma genérica el comportamiento de los *tokens*.

El estándar se compone de 9 funciones y 2 eventos en su versión más básica y acepta la adición de métodos como por ejemplo, recargar el gas disponible cuando decaiga por debajo de determinado valor.

A continuación, se van a detallar los métodos y eventos mínimos definidos por el estándar. Si bien, previamente conviene destacar dos aspectos: aquellos métodos etiquetados como opcionales indican que ni los contratos ni las interfaces deben ser implementados esperando que estos valores hayan sido definidos; y que los programadores deben controlar el valor *false* de aquellos métodos que retornen un booleano. **(9)**

- *name*: Opcional. Devuelve el nombre del *token*.

```
function name() public view returns (string)
```

- *symbol*: Opcional. Símbolo del *token*.

```
function symbol() public view returns (string)
```

- *decimals*: Opcional. Número de decimales que utilizará el *token*. Es decir si este valor fuera 2, la cantidad mínima de transferencia de ese *token* sería 0.01, si el valor fuera 3, la cantidad mínima sería 0.001 y así sucesivamente.

  function decimals() public view returns (uint8)

- *totalSupply*: Cantidad de *tokens* que existirán.

  function totalSupply() public view returns (uint256)

- *balanceOf*: Balance de la cuenta.

  function balanceOf(address _owner) public view returns (uint256 balance)

- *transfer*: Transfiere una cantidad de *tokens* de una cuenta a otra. Debe comprobar si la dirección origen tiene *tokens* suficientes para afrontar la transferencia y activar el evento *Transfer*.

  function transfer(address _to, uint256 _value) public returns (bool success)

- *transferFrom*: Se utiliza para proporcionar permisos a un *smart contract* para que transfiera fondos desde una determinada cuenta. Si la cuenta origen no está explícitamente autorizada para ejecutar está función, se debe lanzar una excepción.

  function transferFrom(address _from, address _to, uint256 _value) public returns (bool success)

- *approve*: Un usuario permite a una cuenta (_spender) que retire un número de fondos de su propia cuenta. Esta función actualiza el valor de *allowance*.

  function approve(address _spender, uint256 _value) public returns (bool success)

- *allowance*: Devuelve el valor que _spender puede retirar de una cuenta.

  function allowance(address _owner, address _spender) public view returns (uint256 remaining)

- *Transfer*: Los contratos que creen nuevos *tokens* deben activar este evento con la dirección origen *0x0*.

  event Transfer(address indexed _from, address indexed _to, uint256 _value)

- *Approval*: Este evento se activa únicamente cuando una función *approve* resulta exitosa.

  event Approval(address indexed _owner, address indexed _spender, uint256 _value)

4.5.2 ERC-223.

Sin embargo no es el único estándar definido en el ecosistema de Ethereum para la creación de nuevos *tokens*. El estándar ERC-223 surgió a raíz de encontrar un bug en el estándar ERC-20. Cuando se transfiere una cantidad de *ether* a un contrato no compatible con ERC-20, dicha cantidad quedará bloqueada. El *smart contract* no tiene la capacidad de interpretar el estándar de las transacciones que le invocan y por consiguiente, este valor no se transferiría por completo pero si se restaría del balance de la cuenta origen.

Por otro lado, este estándar unifica las transferencias de *tokens* en una única función (*transfer*), independientemente de si el destino es una EOA o un contrato. Esto implica

que las transferencias a contratos, requieren de la mitad de gas para ejecutarse que en el estándar ERC-20.

Cuando en el estándar ERC-20 se quiere transferir *tokens* a un *smart contract*, se deben ejecutar las funciones *approve* y *transferFrom*, para autorizar al contrato a disponer de los fondos y posteriormente transferirlos. En este nuevo estándar (ERC-223) únicamente es necesario ejecutar *transfer*. **(10)**

El estándar queda definido por las siguientes funciones y eventos[44]:

- *totalSupply*: function totalSupply() public view returns (uint256)
- *name*: function name() constant returns (string _name)
- *symbol*: function symbol() constant returns (bytes32 _symbol)
- *decimals*: function decimals() constant returns (uint8 _decimals)
- *balanceOf*: function balanceOf(address _owner) constant returns (uint256 balance)
- *transfer*: Requerida por motivos de compatibilidad con el estándar ERC20, cuya función no contiene un argumento de tipo *bytes*.

  function transfer(address _to, uint _value) returns (bool)
- *transfer*: Permite transferir *tokens* a cualquier cuenta, tanto EOA como *contract account*. El argumento *_data* permite asociar a la transacción determinada información adicional, pero puede ser nula y no añadir nada.

  function transfer(address _to, uint _value, bytes _data) returns (bool)
- *tokenFallback*: Cuando el envío de *transfer* está dirigido a un contrato, ambas funciones deben invocar a esta función. Si está función no se define en la cuenta destino, debería de paralizar la transferencia de *tokens* y establecer como fallida la transacción.

  function tokenFallback(address _from, uint _value, bytes _data)
- *Transfer*: event Transfer(addres indexed _from, address indexed _to, uint256 indexed _value, bytes _data)

4.5.3 ERC-777

Otro de los estándares que ha tratado de mejorar las prestaciones ofrecidas por el ERC-20 es el ERC-777 basada a su vez en la ERC-820.

Se otorga tanto a las EOAs como a las *contract accounts*, la capacidad de controlar y/o rechazar tanto los envíos como las recepciones de *token*, pudiendo rechazar alguna transacción. Para ello, se utilizan los ganchos *tokensToSend* y *tokensReceived*, que se

[44] Las funciones compartidas por el estándar ERC20 no vuelven a explicarse.

ejecutan cuando la transacción se ha ejecutado pero antes de confirmar el estado final para que la cuenta pueda decidir si acepta o no la transacción.

Nuevamente, se reduce a la mitad la cantidad de gas requerida para transferir *tokens* a la cuenta de un contrato registrando el gancho *tokensReceived*. Otra mejora reseñable es que el titular del *token* puede a autorizar y desautorizar a ciertos operadores para que envíen *tokens* bajo su consentimiento.

Se incluyen eventos que permiten crear (*Minted*) y destruir (*Burned*) *tokens*, está última mediante las funciones *burn,* de una cuenta propia, y *operatorBurn*, de una cuenta de la que dispongamos consentimiento.

Sin embargo, el uso de este *token* no está muy extendido en la actualidad y por ese motivo no se van a detallar las funciones, eventos y ganchos que forman el estándar. Si bien, podrán consultarse en un anexo. (11)

4.5.4 ERC-721.

Hasta ahora, todos los *tokens* analizados estaban orientados a operar con bienes fungibles (criptomonedas). Sin embargo, la ERC-721 describe la información de un estándar para *tokens* no fungibles.

Un *token* no fungible es aquel que no tiene iguales. En el caso de Ethereum, un ether es intercambiable, lo que supone que es un *token* fungible. Pese a ser ethers "diferentes", *per se* simbolizan lo mismo. En contraposición, un *token* no fungible puede ser la propiedad de una casa. Siguiendo con el ejemplo, es posible intercambiar una propiedad por otra, pero su valor puede ser diferente y, en el momento en el que se transfiere el *token* deja de pertenecerte. (12)

El proyecto basado en este estándar más conocido es *CryptoKitties*, un juego basado en la singularidad de los *tokens* ERC-721. Cada uno de los *tokens* es un gato con características especiales y el juego consiste en criar, comprar e intercambiar estos gatos.

La iniciativa de la empresa AxiomZen[45], surgió con la idea de obtener un producto que ofreciera una visión más desenfadada de la cadena de bloques, para que nuevos usuarios comenzaran a interactuar con los *smart contracts* y se familiarizaran con ellos. Para poder criar un nuevo criptogato, es necesario disponer de un criptogato masculino y uno femenino y ejecutar un *smart contract*, cuya tarea consiste en conjugar las características de los padres y crear un nuevo *token* único.

[45] Sitio web de AxiomZen: https://www.axiomzen.co/

Dejando de lado el juego, este estándar ofrece la posibilidad de crear sistemas de registro de la propiedad, donde podría quedar perfectamente identificado a quién pertenece una parcela de tierra o una vivienda, estando disponible en todo momento en la cadena de bloques y pudiendo rastrear en qué momento se producen las transferencias o a quién ha pertenecido el *token*.

4.6 Lenguajes de programación para smart contracts.

La EVM procesa y ejecuta scripts escritos en lenguaje máquina (*bytecodes*). Sin embargo, la tarea de programar en lenguaje máquina es una tarea imposible. Por ese motivo existen lenguajes de programación de alto nivel y compiladores, que convierten el código de alto nivel a código máquina.

Existen diferentes lenguajes de programación diseñados especialmente para diseñar *smart contracts*:

- *LLL*[46]: *Low-level Lisp-like Language*. Fue el primer lenguaje de programación de alto nivel creado para trabajar con los *smart contracts* de *Ethereum*. Es muy semejante al lenguaje ensamblador y actualmente está en desuso.
- *Serpent*: Lenguaje inspirado en *Python*, pero obsoleto.
- *Flint*[47]: Es un lenguaje de programación, que está en fase de desarrollo, orientado a diseñar contratos inteligentes más robustos.
- *Pyramid Scheme*[48]: Una versión del lenguaje de programación *Scheme*, que se inspira también en *Lisp*, pero orientado a los requisitos de la EVM. Actualmente está en fase experimental.
- *Bamboo*[49]: Se inspira en el lenguaje de programación concurrente *Erlang*. Se caracteriza por no disponer de bucles y hacer explícito el resultado de la ejecución. Está en fase de desarrollo.
- *Vyper*: Diseñado recientemente por Vitalik Buterin, quien publicó el 4 de octubre de 2018 una documentación[50] que expone todos sus detalles. Es un lenguaje de programación que pretende ofrecer la posibilidad de crear contratos inteligentes

[46] Se puede consultar documentación relacionada con LLL en el siguiente enlace: https://lll-docs.readthedocs.io/en/latest/lll_introduction.html

[47] Toda la documentación relativa al estado del desarrollo, manuales, etc. puede consultarse en: https://docs.flintlang.org/

[48] Dispone de un repositorio en GitHub donde consultar la información relevante: https://github.com/MichaelBurge/pyramid-scheme

[49] El repositorio de GitHub asociado es: https://github.com/pirapira/bamboo

[50] Buterin, Vitalik, 4 de octubre de 2018. VYPER DOCUMENTATION. [En línea]. Disponible en: https://media.readthedocs.org/pdf/viper/latest/viper.pdf **[consulta: 12 de febrero de 2019]**

seguros y cuya lectura sea sencilla para usuarios con poca experiencia en la programación. Es una propuesta muy interesante a tener en cuenta en el futuro.

- *Solidity:* Es el lenguaje de programación más utilizado para desarrollar *smart contracts*. Está en constante desarrollo, pero en comparación con el resto de lenguajes, se puede decir que está en una fase de madurez avanzada. Por ese motivo se va a profundizar en él a continuación.

4.7 *Solidity.*

Se trata de un lenguaje de programación orientado a objetos (POO) con tipos estáticos y sensibles a mayúsculas y minúsculas, que está inspirado en C++, *Python* y *JavaScript* y ha sido especialmente diseñado para los *smart contracts* de *Ethereum*.

Un fichero *Solidity*, identificado con la extensión **.sol**, se compone de los siguientes constructores de alto nivel:

- **Pragma:** Permite identificar la versión o versiones del compilador compatibles con el código implementado. La versión se específica con un formato [*major.minor*], donde *major* se incrementa con cambios de gran alcance y *minor* con pequeñas modificaciones. Por último, se puede incluir previo a la versión el carácter "^", lo que implicaría que el código puede ser compilado con cualquier versión del compilador perteneciente a esa *major versión* y cualquier *minor versión*.
- ***Comments:*** Se diferencian tres posibles tipos de comentarios. Para una sola línea ("//"), multi-línea ("/* … */") y un tercer tipo acuñado como **Ethereum Natural Specification** (Natspec), que se utiliza para documentar paso a paso el contrato en el propio fichero. En el caso de Natspec, un comentario de una sola línea se especifica con "///" y si se extiende en varias líneas "/** … */".
- ***Import:*** Permite acceder al código de otros ficheros.
- ***Contracts:*** Recoge el comportamiento del *smart contract* en cuestión. Es el lugar donde se definen las funciones, variables, etc.
- ***Library:*** Son semejantes a los contratos. Se diferencian en que, cuando se llama a un contrato, su código se ejecuta en el contexto del contrato llamado, si se llama a una librería, la funcionalidad se ejecuta en el contexto del contrato llamante (mediante el *opcode* DELEGATECALL).
- ***Interface:*** Se utiliza para expresar los inputs y outputs de un *smart contract*. No puede contener ninguna definición ni heredar de ningún otro contrato, aunque si pueda heredar de otras interfaces.

4.7.1 Estructura de un contrato.

En *Solidity*, un contrato es la analogía a una clase en cualquier otro lenguaje de programación orientado a objetos. En los contratos se pueden definir constructores, variables de estado (*state variables*), estructuras de datos, modificadores, eventos, enumeraciones y funciones.

4.7.1.1 Constructor.

En los lenguajes POO se implementan constructores que permiten crear instancias de clases y otorgar un valor inicial a las diferentes variables "globales" de la clase en cuestión. *Solidity* no es la excepción y utiliza los constructores para que se puedan crear instancias de contratos. Cualquiera de los dos ejemplos expuestos a continuación es válido.

```
contract example {
    address owner;

    constructor() public {
        owner = msg.sender;
    }
}
```

```
contract example {
    address owner;

    function example() public {
        owner = msg.sender;
    }
}
```

En este lenguaje está permitido que unos contratos hereden de otros. Esto implica que el contrato "hijo" puede disponer de las variables y funciones que hayan sido definidas como *public* o *internal* en el contrato "padre". La herencia puede ser múltiple. Para denotarlo se debe seguir la siguiente lógica:

```
contract exampleA {
    […]
}
```

```
contract exampleB is exampleA {
    […]
}
```

4.7.1.2 *State variables.*

Las variables de estado contienen aquella información que se almacena de manera permanente en la *blockchain*. Las variables de estado no pertenecen a ninguna función. De hecho, son las variables que recogen el estado actual del contrato.

El tipo de estas variables debe ser definido de manera estática y pueden incluir algún indicador relativo a su accesibilidad. Los indicadores son los siguientes:

- **`internal`**: Se asume por defecto. Indica que la variable solo puede ser utilizada por las funciones que pertenecen o heredan del contrato. Pese a que puedan ser observadas desde el exterior del contrato, no pueden ser modificadas.
- **`public`**: Permite acceder a la variable desde cualquier punto. En caso de elegir este indicador, *Solidity* creará automáticamente un método *getter* que devuelve el valor de la variable en cuestión.
- **`private`**: A diferencia de las variables *internal*, solo pueden ser utilizadas por las funciones que pertenecen al contrato, no así en los contratos que hereden del mismo.
- **`constant`**: Es un indicador adicional, no relacionado con la accesibilidad de la variable si no con su modificabilidad. La variable que sea definida con este indicador, deberá ser inicializada en el mismo momento y no podrá ser modificada.

```
int internal variable; // The same like "int variable"
int public publicVariable;
int private privateVariable;
int constant variable = 3;
```

Respecto al tipo de variables, pueden definirse los siguientes tipos:

- **Boolean:** Utilizada para representar escenarios con resultados binarios. Puede únicamente contener "True" o "False".
- **Integer:** Pueden ser *signed* (negativos o positivos) o *unsigned* (solo positivos y ceros). El espacio reservado se incrementa de 8 en 8 bits y pueden definirse enteros de 8 bits hasta 256. Por defecto, se asume un entero de 256 bits. Se pueden efectuar operaciones a nivel de bit.
- **Struct:** Permite crear clases de tipos que contienen varias variables de varios tipos. El acceso a cada una de las variables se realizaría de igual manera que en el resto de lenguajes POO.
- **Address:** Diseñado específicamente para almacenar direcciones de Ethereum. Por ende, su tamaño es de 20 bytes o 160 bits. Existe una segunda versión de este tipo que incluye los métodos `transfer` y `send`, **address payable**.

- **Byte Arrays de dimensión fija:** Son variables que permiten almacenar información en formato binario. Se pueden almacenar desde 1 a 32 bytes.
- **Enums:** Permite almacenar una serie de valores constantes en una única variable.
- **String:** Permite almacenar cadenas de texto.
- **Mapping:** Permite almacenar información en formato clave/valor. Es para *Solidity* lo que suponen los diccionarios en algunos lenguajes POO. Sin embargo, en este caso no se puede iterar sobre los *mappings*.

```
int internal variable; //The same like "int variable"
bool isOpen;
int counter; //signed
int8 counter_8bits; //signed
uint256 counter_256bits; //unsigned
struct person{
      string name;
      uint8 edad;
      boolean isDead;
}
address payable myAccount;
enum result {win, draft, defeat};
mapping (address => uint) accounts; /*key of type address,
value of type uint*/
```

4.7.1.3 Funciones.

Las funciones son la unidad mínima de ejecución de código en un contrato. Cuando una función es invocada o llamada, su resultado desemboca en la creación de una transacción. *Solidity* permite que sus funciones devuelvan más de un parámetro.

Se pueden incluir modificadores que alteren el comportamiento de las funciones. Sin embargo, esto se explicará en el siguiente epígrafe. Al igual que ocurría con las variables de estado, se pueden incluir indicadores relativos a su visibilidad:

- **`Internal:`** Se asume por defecto. Indica que las funciones no pertenecen a la interfaz del contrato y que únicamente pueden ser usadas en el *scope* relativo al contrato al que pertenecen y a los contratos que hereden del mismo.
- **`External:`** En este caso, la función pertenece a la interfaz del contrato, pero no podría ser accedida desde el interior del mismo, sino desde el exterior.
- **`Public:`** Las funciones públicas deben ser incluidas en la interfaz del contrato y pueden ser llamadas tanto de manera interna como externa.
- **`Private:`** Indica que la función no pertenece a la interfaz del contrato y que únicamente puede ser llamada desde el *smart contract* al que pertenece, no así desde los contratos que hereden de él.

Existe otro tipo de calificadores de la función en relación a su capacidad para modificar las variables de estado del *smart contract*:

- **`view:`** No pueden modificar las variables de estado pero si acceder a ellas para leerlas y devolver su valor.
- **`pure:`** No tienen acceso, ni de lectura ni de escritura, a las variables de estado. Por consiguiente, este tipo de funciones se utilizan exclusivamente para lógica auxiliar, típicamente acciones que van a ser repetidas durante el contrato.
- **`Payable:`** Le otorga a la función la capacidad de aceptar *Ether* del llamante. En caso contrario, al invocar una función e introducir como parámetro de entrada una cantidad de *Ether*, será rechazada dicha cantidad.

Algunos ejemplos de funciones pueden ser `trasnfer` o `send`, ambas utilizadas para transferir *Ether* pero con una diferencia sustancial. En caso de fallo, la función `send` retorna un booleano con valor *false*, mientras que la función `transfer` lanza una excepción y revoca todos los cambios aplicados hasta el momento. El input de ambas funciones es un literal que indica el valor a transferir en wei.

Típicamente las funciones suelen tener parámetros de entrada y salida. Sin embargo, en todos los *smart contracts* existe o debería existir una función sin argumentos de entrada ni de salida. Esta función se denomina función de retirada o *fallback function*.

El motivo de su necesidad no es otro que, cuando se llama a una función que no existe o se le introducen argumentos erróneos, automáticamente se invoca a esta función. La función debe contener el calificador `external` y, en caso de ser necesario tratar con *Ethers*, `payable`.

Debido a que de manera bienintencionada no puede invocarse la *fallback function*, no se puede proporcionar gas para su ejecución. Por este motivo, la EVM proporciona una cantidad fija de 2300 gas para soportar su posible ejecución. Esto implica que la función de retirada debe ser implementada y probada para asegurar que no supere dicha cantidad de gas y genere una excepción del tipo *out-of-gas*.

```
function() payable{
      logme("fallback invoked");
}; //Example of simple fallback function
```

```
function doubleStock(uint amount) public returns (uint){
      return amount*2;
}
```

```
function payDebt(address receiver, uint amount)public payable
returns (bool){
      return receiver.send(amount);
}
```

```
function payLoan(address payable receiver, uint amount) public
payable returns (uint balance) {
        uint balanceBeforePay = address(this).balance;
```

```
        receiver.transfer(amount);
        assert(address(this).balance == balanceBeforePay - amount);
        return address(this).balance;
}/*Se incluye un mecanismo de control que será explicado más
adelante, assert.*/
```

```
function getWinnerTeam() public view returns (string winner){
            winner =  teams[getWinner()].name;
{
```

Anteriormente se ha definido la manera de instanciar un contrato, mediante el constructor. *Solidity* predefine una función que permite también destruir el contrato. En caso de que el contrato no contemple esta casuística, el *smart contract* permanecerá de por vida en la *blockchain*, aunque deje de ser útil, no sería posible eliminarlo. Un ejemplo podría ser el siguiente:

```
function destroy() public {
        if(msg.sender==owner){ //Si quién lo invoca es el dueño
            selfdestruct(owner); //Destruye el smart contract
        }
}
```

Es importante declarar la función con el calificador `public` para que esta función pueda ser invocada desde el exterior del contrato.

4.7.1.4 Modificadores.

Como se ha comentado en la subsección anterior, el comportamiento de una función se puede alterar mediante la inclusión de modificadores, un concepto único de *Solidty*. Conviene destacar que los modificadores pueden únicamente asociarse a funciones.

Los modificadores permiten abstraer determinadas funcionalidades y aplicarlas en diferentes contextos para, típicamente, restringir los casos de usos. En el siguiente ejemplo se van a exponer dos códigos cuya funcionalidad es exactamente idéntica:

```
contract example1 {
   address owner;

   function example1() public{
       owner = msg.sender;
   }

   function payMoney(address receiver, uint amount) public payable
returns(bool){
        if(msg.sender == owner){
            return receiver.send(amount);
        }
   }
}
```

```
contract example2{

    address owner;

    function example2() public {
        owner = msg.sender
    }

    modifier isOwner {
        if(msg.sender == owner) {
            _;//The logic of the functionModified will replace _;
        }
    }

    function payMoney(address receiver, uint amount) public payable
isOwner returns(bool){
        return receiver.send(amount);
    }
}
```

En el contrato `ejemplo1` se ha definido la función `payMoney`, que permite efectuar una transferencia de *ether* únicamente a la cuenta que desplegó el contrato. El `ejemplo2` es funcionalmente igual. Sin embargo, la restricción que comprueba si la dirección que está invocando al contrato es la misma que la dirección que lo desplegó, se ha definido en un modificador.

Por consiguiente, todas las funciones que quisieran implementar la misma comprobación, podrían incluir el modificador `isOwner` sin necesidad de volver a escribir el mismo código. Este permite implementar códigos más eficientes y más sencillos de mantener y modificar.

4.7.1.5 Eventos.

Es un mecanismo para reportar el resultado de una ejecución y almacenarlo en la *blockchain*, de manera que cualquier interesado puede conectarse con el evento. Estos eventos pueden registrar cualquier acción que haya sucedido durante la ejecución del contrato y se considere relevante reseñar.

```
Contract example {
        event acceptEther(address from, uint amount);
        //Se acepta cualquier ingreso
        function () public payable {
                emit acceptEther(msg.sender,msg.value);
        }
}
```

4.7.2 Variables y funciones globales.

Se dispone de una cantidad elevada tanto de variables como de funciones globales predefinidas. A continuación se incluyen las más relevantes:

Literales *ether*	
Variable	**Equivalente**
`1 wei`	1
`1 szabo`	1e12
`1 finney`	1e15
`1 ether`	1e18

Tabla 1. Literales ether.
Tomada de: **(17)**

Literales temporales	
Variable	**Equivalente**
`1`	1 segundo
`1 minutes`	60 segundos
`1 hours`	60 minutos
`1 day`	24 horas
`1 week`	7 días

Tabla 2. Literales temporales.
Tomada de: **(17)**

Propiedades relacionadas con el bloque y la transacción	
Variable o función (type returns)	**Descripción**
`blockhash(uint blockNumber) returns (bytes32)`	*Hash* del bloque al que se hace referencia. Se tiene únicamente acceso a los últimos 256 bloques.
`block.coinbase (address payable)`	Indica la dirección del minero.
`block.difficulty (uint)`	Dificultad relativa al bloque actual.
`block.gaslimit (uint)`	Límite de gas para ejecutar todas las transacciones del bloque.
`block.number (uint)`	Número del bloque. Este valor se incrementa en uno cada vez que mina un nuevo bloque.
`block.timestamp (uint)`	Marca temporal que indica cuando fue creado el bloque. El formato utilizado son los segundos desde transcurridos desde medianoche UTC del 1 de enero de 1970.
`now (uint)`	
`gasleft() returns (uint256)`	Gas disponible.

`msg.data (bytes)`	Información y parámetros de la función que ha creado la transacción.
`msg.sender (address payable)`	Dirección de la cuenta que ha invocado al contrato (en el despliegue) o a la función. Puede ser diferente que `tx.origin`. ***
`msg.sig (bytes4)`	Identificador de la función.
`msg.value (uint)`	Cantidad de wei enviada en la transacción.
`tx.gasprice (uint)`	Precio del gas que el llamante está dispuesto a asumir.
`tx.origin (address payable)`	Dirección de la cuenta que originó la transacción. Este valor siempre retorna el mismo resultado durante toda la ejecución, a diferencia de `msg.sender.` ***

Tabla 3. Propiedades relacionadas con el bloque y la transacción.
Tomada de: **(17)**

*** `tx.origin` y `msg.sender` pueden ser diferentes en un escenario como el siguiente: Si la dirección "A" invoca al contrato "B" y este a su vez invoca al contrato "C" tendremos dos transacciones:

A→B: `tx.origin` = A, **`msg.sender`** = A.
B→C: `tx.origin` = A, **`msg.sender`** = B.

Funciones matemáticas	
Función (type returns)	**Descripción**
`keccak256(bytes memory) returns (bytes32)`	Ejecuta la función de hash del algoritmo Keecak-256 sobre el input.
`sha256(bytes memory) returns (bytes32)`	Idéntico pero aplicando el algoritmo SHA-256.
`ripemd160(bytes memory) returns (bytes20)`	En este caso aplicando el algoritmo RIPEMD-160

Tabla 4. Funciones matemáticas.
Tomada de: **(17)**

Propiedades relacionadas con una cuenta	
Variable o función (type returns)	**Descripción**
`<address>.balance (uint256)`	Cantidad en wei disponible.
`<address payable>.transfer( uint256 amount)`	Transfiere una cantidad en wei. Genera una excepción en caso de fallo y requiere 2300 gas para ejecutarse.

`<address payable>.send(uint256 amount) returns (bool)`	Igual que la anterior pero, en vez de generar una excepción, retorna un booleano con valor *false*.
`<address>.call(bytes memory) returns (bool, bytes memory)`	Método que permite interactuar a dos contratos. Ejecuta el código del contrato almacenado en `memory` en el contexto del contrato invocado.
`<address>.delegatecall(bytes memory) returns (bool, bytes memory)`	Igual que la anterior pero, el código se ejecuta en el contexto del contrato que invoca. Si el contrato invocado contiene código malicioso, la ejecución se realizará en nombre del contrato invocador.
`<address>.staticcall(bytes memory) returns (bool, bytes memory)`	En este caso el código ejecutado no puede modificar el estado global. En caso de intentarlo se generaría una excepción.

Tabla 5. Propiedades relacionadas con una cuenta.
Tomada de: (17)

Funciones relacionadas con el *smart contract*	
Función	**Descripción**
`this`	Hace referencia al contrato actual.
`selfdestruct(address payable recipient)`	Destruye el contrato actual y envía los fondos disponibles a la dirección `recipient`.

Tabla 6. Funciones relacionadas con el *smart contract*.
Tomada de: (17)

4.7.3 *Exception Handling.*

Una recomendación que se incluye en todos los manuales de buenas prácticas a la hora de implementar programas informáticos, está relacionada con el control de errores y excepciones.

Algunos de estos errores pueden ser las excepciones *out-of-gas*, ejecutar una división por una variable y que en un contexto esa variable valga cero, una excepción *array-out-of-index*, etc. *Solidity* ofrece ciertos mecanismos de control para evitar estas circunstancias y detener la ejecución del contrato:

- **`require`**: Recibe como *input* una expresión booleana. En caso de reportar un valor *true* la ejecución continúa, si por el contrario el valor resultante es *false*,

se lanza una excepción y se detiene la ejecución. Típicamente se suele utilizar esta expresión para comprobar los *inputs* de una función.

- **`assert`**: Con una funcionalidad idéntica a la anterior pero, en este caso, utilizada para comprobar expresiones cuyo resultado se espera que sea *true*.
- **`revert:`** Se utiliza para lanzar excepciones sin necesidad de comprobar ninguna expresión. Es posible incluir un mensaje descriptivo que indique le motivo de la excepción.

```
function payLoan(address payable receiver, uint amount) public payable
returns (uint balance) {
        require (address(this).balance > 0); /*Si la cuenta que invoca
al contrato no dispone de ether la ejecución se detiene.*/
        uint balanceBeforePay = address(this).balance;
        if (receiver.send(amount) == false){
                revert(); //Se interrumpe la ejecución.
        }
receiver.transfer(amount);
        assert(address(this).balance == balanceBeforePay - amount);
/*Si tras la ejecución el balance final no es correcto, la ejecución
se detiene*/
        return address(this).balance;
}
```

4.7.4 ABI.

Una de las problemáticas a la hora de compilar un programa informático reside en su utilidad en otras arquitecturas. La manera en la que se realizan las llamadas al sistema difiere en cada arquitectura. El *kernel* de dos sistemas operativos no tiene por qué comunicarse con el exterior de igual manera, de hecho lo normal es que no lo haga.

Por este motivo, surge la necesidad de buscar una solución que permita al programador despreocuparse de este aspecto. La solución se denomina *Application Binary Interface* (ABI).

Una interfaz binaria de aplicación define la forma en la que deben realizarse las llamadas al sistema y el formato binario en el que se le deben proporcionar los parámetros de entrada. Es decir, supone un puente entre el sistema operativo y los programas de los usuarios.

En *Ethereum* se habla de "ABI del contrato". Esto no es otra cosa que un JSON que incluye toda la información relativa a las funciones y los eventos que compone el

contrato. A continuación se muestra un ejemplo de contrato y su correspondiente ABI, extraído directamente de la propia documentación de *Solidity*[51]:

```
contract Test {
  struct S { uint a; uint[] b; T[] c; }
  struct T { uint x; uint y; }
  function f (S memory s, T memory t, uint a) public;
}
```

```
Contract ABI JSON
[{"name": "f", "type": "function", "inputs": [
{"name": "s", "type": "tuple", "components": [
{"name": "a", "type": "uint256"},
{"name": "b", "type": "uint256[]"},
{"name": "c", "type": "tuple[]","components": [
{"name": "x", "type": "uint256"},
{"name": "y", "type": "uint256"}
]}]},
{"name": "t", "type": "tuple", "components": [
{"name": "x","type": "uint256"},
{"name": "y","type": "uint256"}
]},
{"name": "a","type": "uint256"}
],
"outputs": []}]
```

4.7.5 *Smart contracts* mal programados.

Es muy importante centrar los esfuerzos en diseñar los contratos de la manera más robusta posible y efectuar todas las pruebas necesarias para cerciorarse de que el contrato es seguro y no ofrece ninguna puerta trasera o ningún *bug* que un atacante malicioso pueda aprovechar.

Esto parece algo muy evidente, teniendo en cuenta que estamos tratando con criptoactivos que tienen un valor fiduciario asociado, ¿quién no va a revisar hasta la última coma de su contrato para asegurarse de que no pone en peligro sus fondos?

Sin embargo, el ser humano no es perfecto. Por muy bueno que fuera el programador que implementara un *smart contract*, siempre puede haber un despiste o puede no haberse planteado una forma de atacar el contrato, de manera que puedan quedar puertas traseras por las que atacar el código.

[51] El código se ha extraído concretamente de la siguiente ruta: https://solidity.readthedocs.io/en/v0.5.3/abi-spec.html [**consulta: 17/02/2019**]

Como se comentó en el capítulo referente a las distintas fases de *Ethereum*, una idea que se posicionaba como unos de los referentes en el ecosistema *blockchain* finalmente desembocó en un *hard fork* muy controvertido. Esta idea era "The DAO".

Peter Vessenes publicó el 9 de enero de 2016 un post en el que indicaba que el *smart contract* que controlaba esa idea, tenía una posible vulnerabilidad que permitía a los usuarios retirar el doble de los fondos que hubieran invertido.

El código que se expone de muestra es el siguiente:

```
function getBalance(address user) constant returns(uint) {
  return userBalances[user];
}

function addToBalance() {
  userBalances[msg.sender] += msg.amount;
}

function withdrawBalance() {
  amountToWithdraw = userBalances[msg.sender];
  if (!(msg.sender.call.value(amountToWithdraw)())) { throw; }
  userBalances[msg.sender] = 0;
}
```

Se puede comprobar como en la tercera función se realiza una llamada a una función de `msg.sender` sin argumentos de entradas, una función por defecto. Se plantea y se expone una posible función que permitiría efectuar un ataque:

```
function () {
  vulnerableContract v;
  uint times;
  if (times == 0 && attackModeIsOn) {
    times = 1;
    v.withdraw();

   } else { times = 0; }
}
```

Si la función por defecto tuviera este formato y se ejecutara una llamada a la función `withdrawBalance()`, ocurriría la siguiente secuencia: El contrato vulnerable ejecutaría la retirada de fondos y llamaría a la función maliciosa, que ejecutaría una nueva retirada de fondos y finalmente asociaría el balance final a cero.

El balance al que se está haciendo referencia sería el relativo a las inversiones del usuario en la organización "*The DAO*".

Este error es el que provocó el desfalco de los fondos de "*The DAO*". Una posible solución, para el código anterior, pasaría por reorganizar las sentencias a ejecutar, asignando el valor cero al balance del usuario antes de ejecutar la transferencia de fondos:

```
function withdrawBalance() {
  amountToWithdraw = userBalances[msg.sender];
  userBalances[msg.sender] = 0;
  if (amountToWithdraw > 0) {
    if (!(msg.sender.send(amountToWithdraw))) { throw; }
  }
}
```

(13)

Este *bug* se conoce como ataque de reentrada, ya que como su propio nombre indica, se puede aprovechar alguna vulnerabilidad del código para ejecutar en más de una vez la función `withdraw()` cuando solo debiera ejecutarse en una ocasión.

Sin embargo, y pese al drama y la controversia que generó el caso "*The DAO*", este no ha sido el único contrato que ha dado pie a efectuar ataques de este tipo. En las fechas en las que se escribió este documento se debería haber hecho efectivo el *hard fork* que actualizaba la *blockchain* de *Ethereum* de la fase *Byzantium* a la fase *Constantinople*.

Pero esto no fue posible debido a que se localizó una vulnerabilidad de este tipo al aplicar la EIP1283. En resumen, esta propuesta iba a reducir el gas necesario para ejecutar el *opcode* SSTORE. Si todos los usuarios fueran bienintencionados, esto permitiría reducir el coste de ejecutar transacciones. Sin embargo, hay que tener presentes a los usuarios maliciosos.

Para explicar la problemática se va a proponer un ejemplo. Supongamos una entidad bancaria que se está replanteando sus procesos internos para ofrecer un mejor servicio. En concreto observan que los usuarios deben esperar durante 10 segundos para poder acceder a su caja y deciden implementar las medidas efectivas para que puedan acceder a ella en tan solo 1 segundo.

Supongamos por otro lado que un experimentando ladrón conoció esta noticia y se entrenó para acceder a una de esas cajas, retirar el dinero y salir, pasando totalmente desapercibido en tan solo 3 segundos.

En el primer escenario, al tardar tanto tiempo en abrirse la caja, el ladrón no podría sacar provecho de su habilidad, sin embargo, en la segunda casuística dispondría de un precioso tiempo para una vez entrado a su caja y retirado sus fondos, acceder a la caja de otro cliente y hacerse con sus fondos sin activar ninguna alarma.

Este hecho provocó que la actualización a la fase *Constantinople*, se dividiera en dos *hard fork*. El que lleva el mismo nombre de la nueva fase e implantará entre otras la EIP1283 y el *hard fork* denominado *Petersburg*, cuyo único objetivo es deshabilitar la misma EIP1283.

4.7.6 Otras consideraciones.

Una vez definidos los aspectos más destacables del lenguaje de programación *Solidity*, es conveniente destacar que hay determinadas funcionalidades que no se han presentado en este documento pero que si son ofrecidas por *Solidity*.

Este lenguaje de programación ofrece la posibilidad de trabajar con estructuras de control `if…else` y con bucles **`while`**, **`for`** y `do…while`. Para alterar el flujo habitual de los bucles se pueden aplicar las declaraciones **`continue`**, para detener el ciclo actual y pasar al siguiente, y **`break`**, para escapar del bucle.

Existen diferentes alternativas para programar *smart contracts* y virtualizar una infraestructura en la que probar los desarrollos. Por ejemplo, se puede utilizar un editor de texto con posibilidades de *debuggear* como **Remix**[52], para implementar el contrato, se pueden utilizar herramientas como **Ganache**[53] para generar una red de *testing* basada en Ethereum y se puede hacer uso de algunas opciones como **Truffle**[54] para desplegar los contratos y ejecutar comandos sobre la *blockchain*.

4.8 Oráculos.

Un *smart contract* puede depender de condiciones/circunstancias externas a la *blockchain*, siempre y cuando al ser comprobadas ofrezcan un único resultado. Es decir, en el supuesto caso en que un *smart contract* consulte una referencia extra-*blockchain* y la respuesta fuera diferente para cada consulta realizada se obtendría un escenario muy particular.

Como se ha comentado en repetidas ocasiones, el código del *smart contract* es ejecutado por todos los nodos mineros para validarlo haciendo efectivo el algoritmo de consenso de la red. Sin embargo, si en cada ejecución se recibe una respuesta diferente, ¿qué ocurre con el consenso?

---

[52] El editor de texto está disponible en: https://remix.ethereum.org/#optimize=false&version=soljson-v0.5.1+commit.c8a2cb62.js

[53] La herramienta puede descargarse en el siguiente link: https://truffleframework.com/ganache

[54] Se puede consultar la solución en: https://truffleframework.com/truffle

Obviamente esto es un supuesto puramente contradictorio con la filosofía *blockchain*. Si se realiza una consulta externa a la red, la respuesta ha de ser determinista, de forma que se ejecute cuando se ejecute la transacción, su resultado pueda ser validado.

Una información válida puede ser, por ejemplo, la proporcionada por un sensor IoT en relación a la velocidad del viento en una región para así poder determinar si ha sucedido una catástrofe natural o no.

Una compañía de seguros que opere en una red *blockchain* podría aprovechar el anterior supuesto, completamente factible, para considerar un escenario de catástrofe natural y abonar a los usuarios que hayan contratado sus servicios una cantidad previamente acordada. Cabe destacar que, al tratarse de un *smart contract*, el pago se haría efectivo en el mismo momento en que el sensor superara una determinada cifra.

Otro ejemplo puede aplicarse a una casa de apuestas *blockchain-made*. La casa de apuestas despliega un *smart contract* con varias cuotas asociadas a un partido y un usuario decide aceptar esas condiciones. Para conocer el resultado del partido y poder transferir la cantidad acordada al ganador de la apuesta se debe acudir a una fuente de datos externa.

Para este caso puede ser cualquier web que ofrezca resultados deportivos. Sin embargo, si se acude a una única fuente, se estaría centralizando en una red *blockchain*. Esto vuelve a suponer una contradicción con la filosofía *blockchain*, entre otras cosas por qué la información podría modificarse entre la consulta y el reporte del resultado.

Por consiguiente, surge la necesidad de autenticar los datos tratados por el oráculo. Esta autenticación puede realizarse mediante pruebas de autenticidad o mediante entornos de ejecución de confianza (*Trusted Execution Environments* – TEEs).

4.8.1 Pruebas de autenticidad.

Las pruebas de autenticidad tienen base puramente criptográfica y pretenden asegurar la integridad de los datos tratados.

Oraclize es una solución orientada precisamente a actuar como oráculo en los *smart contracts*, ofreciendo una prueba de autenticidad basada en TLSNotary. Estas pruebas ofrecen evidencias del tráfico HTTPS ocurrido entre cliente y servidor. Sin embargo, el protocolo HTTPS es seguro per se, por lo que no permite firmas criptográficas y se recurre a las firmas de TLSNotary.

En este proceso intervienen tres componentes:

1. **Auditor**: En este caso el smart contract.
2. **Auditado**: Oraclize.

3. **Servidor**: El oráculo, la fuente de información.

El funcionamiento más en detalle del proceso seguido puede observarse en la siguiente imagen, que describe paso a paso las acciones que tienen lugar hasta que el auditor recibe la información solicitada. En verde se referencian los pasos que distinguen una conexión TLS normal con una de tipo TLSNotary:

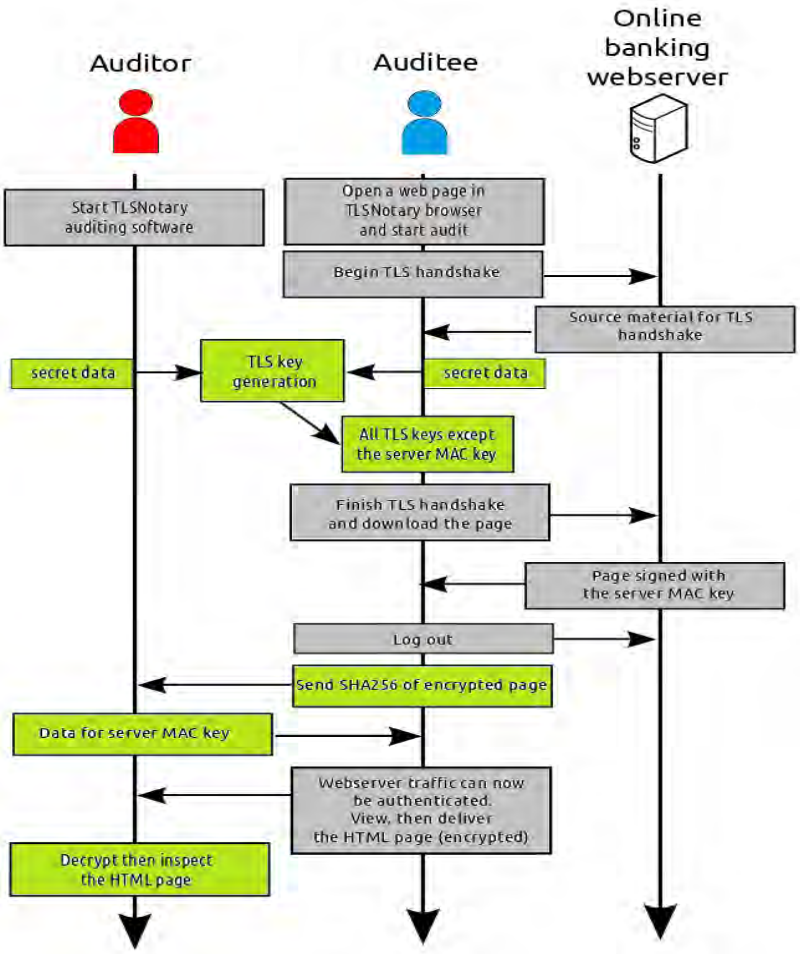


Fig. 11. Funcionamiento TLSNotary.
Tomada de (14)

Desde el punto de vista de programación, será necesario la implementación de la función *__callback* en el smart contract para que el oráculo se comunique con él y pueda transferirle los datos en una nueva transacción. El esquema sería el siguiente:

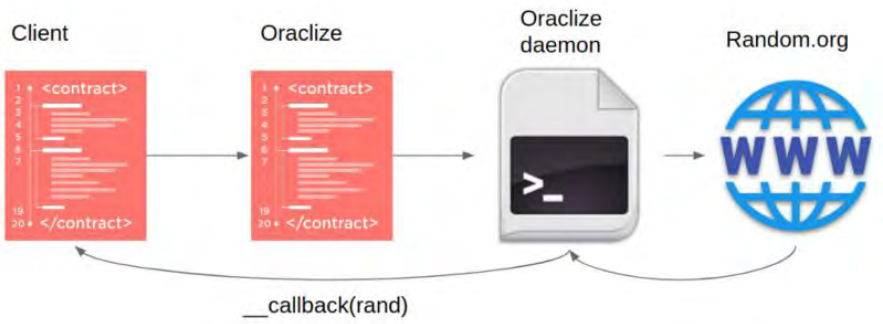


Fig. 12. Consulta Oraclize.
Tomada de (15)

4.8.2 Entornos de ejecución de confianza.

Un *trusted execution environment* (TEE) permite trabajar con información sensible de manera segura, permitiendo ejecutar códigos y/o programas manteniendo seguros algunos datos confidenciales.

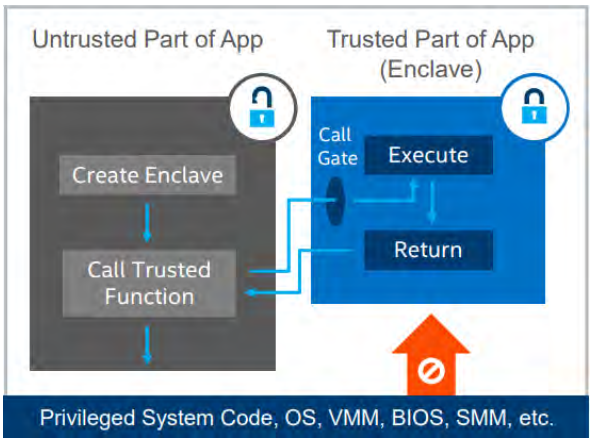
Un buen entorno de ejecución de confianza debe mantener a salvo los datos incluso cuando el sistema operativo o la BIOS hayan sido infectadas con algún software malicioso.

Town Crier es la solución para oráculos basados en entornos de ejecución de confianza más reconocible. En concreto el TEE que utilizan es el Intel SGX (*Software Guard eXtensions*). A continuación se procede a dar una visión más detallada del funcionamiento del entorno de ejecución de confianza de Intel.

La información confidencial se almacenada en zonas de memoria aisladas que se denominan enclaves. En estos enclaves, los programas pueden trabajar sin poner en peligro la información con la que tratan. Para ello, se reserva una parte de la memoria RAM y se cifra.

La aplicación de Intel SGX trabaja con una región confiable y una no confiable. En la región no confiable se crean los enclaves y funciones de confianza. Las funciones de confianza son las que controlan la interacción con la parte confiable. Por decirlo de algún modo actúan como una consulta a una base de datos cifrada. De ninguna otra forma se pueden obtener datos de la parte confiable. Una vez recibida la información, la ejecución puede continuar.

Si uno de los enclaves se ve modificado y ejecuta una llamada a una función de confianza, el software controlador del TEE detectará el cambio y rechazará la petición de información. Esto se realiza para evitar ataques. Si se quiere realizar alguna modificación en algún enclave, se debería eliminar la versión a modificar y construir de cero el nuevo enclave.



Consulta de datos en TEE. Tomada de **(16)**

Mediante el TEE anteriormente descrito, Town Crier ofrece una garantía de autenticidad de los datos, respuestas procesadas de sitios web para reducir la necesidad de almacenamiento en *blockchain* y confidencialidad en sus consultas.

4.9 DApps.

Las aplicaciones descentralizadas o *decentralized apps* (DApps), son como su propio nombre indica, aplicaciones descentralizadas. Esto significa que son una aplicación como otra cualquiera pero desplegada en una red *peer-to-peer*, como pueda ser en este caso una *blockchain*, y su *backend* está formado por uno o varios *smart contracts* que controlan su comportamiento.

Actualmente, son pocas las aplicaciones totalmente descentralizadas que se ejecutan sobre la red *Ethereum*. La mayoría disponen de alguna característica centralizada, bien sea el almacenamiento de datos, el protocolo de comunicación o el *software* que implementa el *frontend* de la DApp.

El objetivo es que en un corto-medio plazo, cuando la tecnología *blockchain* haya madurado, se implementen únicamente aplicaciones 100% descentralizadas.

Existen diferentes soluciones que posibilitan descentralizar cada uno de las características citadas. En relación al almacenamiento, tanto *IPFS*[55] como *Swarm*[56] permiten almacenar de manera descentralizada por ejemplo el fichero que supone el *frontend* de la aplicación.

[55] La documentación de la solución está disponible en: https://ipfs.io/ **[consulta 17/02/2019]**
[56] Se puede profundizar en el link: https://swarm-guide.readthedocs.io/en/latest/introduction.html **[consulta 17/02/2019]**

Respecto al protocolo de comunicación, el más utilizado es *Whisper*[57], un protocolo *peer-to-peer*, que permite a una DApp comunicarse con otra.

Una aplicación descentralizada debe ser *open-source*. Es decir, el código debe estar disponible para que cualquier usuario pueda comprobar el funcionamiento de la aplicación y descartar interactuar posibles códigos maliciosos. Esto es un arma de doble filo, ya que al estar disponible para todo el mundo en todo momento, los *hackers* pueden analizar detenidamente el código para buscar vulnerabilidades.

Si una aplicación descentralizada se despliega sobre una *blockchain*, estará disponible siempre y cuando al menos un nodo de la red haga uso de ella. Esto implica que ningún usuario tiene la capacidad de eliminar la aplicación.

Toda la información que genere la DApp se almacena en la *blockchain*, por lo que es inmutable. Para que esto pueda darse, se requiere un *pool* de mineros que validen las transacciones, las almacenen en bloques y los añadan a la cadena. Por este motivo, es necesario incentivar esta acción, bien sea con *tokens* propios de la DApp o con los *tokens* de la plataforma sobre la que se haya desplegada la aplicación.

De esta última característica surge una distinción de DApps, las que disponen de una *blockchain* propia y por consiguiente un *token* propio, y las que se despliegan sobre una *blockchain* y, típicamente, utilizan los *tokens* de la misma. **(17)**

Aragon[58] es una DApp desplegada sobre la *blockchain* de *Ethereum* que se presenta como una solución para crear y gestionar organizaciones descentralizadas (DAOs). Pese a no disponer de una *blockchain* propia, sí que dispone de un *token* propio (ANT) basado en el estándar ERC20, que otorga el derecho a participar en las votaciones en las que se deciden el futuro de la aplicación.

4.10 DAOs.

Una DAO (*Decentralized Autonomous Organization*) u organización autónoma descentralizada, es la forma más compleja en la que pueda presentarse un *smart contract*. El código y las reglas que se definan en él, controlan el comportamiento de la organización y recogen los estatutos de la misma.

En primer lugar, es necesario explicar qué se entiendo por organización descentralizada o DO (*Decentralized Organization*). Una DO es una organización que imita la estructura jerárquica y funcional de una organización tradicional. La diferencia reside en

[57] Puede consultarse información detallada en el siguiente enlace: https://github.com/ethereum/wiki/wiki/Whisper **[consulta: 17/02/2019]**
[58] Sitio web Aragon Project: https://aragon.org/

que esta estructura pasa de estar controlada por humanos a estar definida en un programa informático que determina la manera en la que deben interactuar los humanos implicados.

Para ser considerada autónoma (DAO), la organización debe disponer de un valor interno y la capacidad de gestionarlo e incentivar a los usuarios que realicen determinadas actividades, como por ejemplo minar transacciones relacionadas con la organización.

En este tipo de organizaciones, las decisiones se suelen consensuar entre todos los componentes de la misma. Desde la actualización del código porque se haya encontrado una posible vulnerabilidad hasta la posible retirada de fondos de una de las partes. Por consiguiente, la gobernanza se distribuye entre todos los miembros de la DAO.

Si bien es cierto, aunque la organización se denomine autónoma, no es del todo cierto. En el supuesto en el que se detecte una vulnerabilidad y sea necesario modificar el código para subsanar la incidencia, el *smart contract* que determina las reglas que rigen la organización no tiene capacidad para implementar estos cambios.

Por ende, en algunos casos requiere de acción humana, aunque sea la propia DAO quien lleve a cabo la contratación de estos servicios.

En la actualidad no existen DAOs de gran calado, quizás debido a la repercusión que tuvo la primera gran DAO, *The DAO*. En el futuro, cuando la tecnología madure y se hayan asentado los conocimientos entre la comunidad informática, posiblemente surgirán nuevas propuestas que se llevarán a cabo en forma de DAOs.

5 EL FUTURO DE LA PLATAFORMA *ETHEREUM*.

Como se ha comentado y repetido en varias ocasiones, *Ethereum* está evolucionando constantemente porque busca ser una plataforma en la que se desplieguen soluciones basadas en *smart contracts*. Es por ello que la comunidad de *Ethereum* busca las debilidades de su protocolo e intenta subsanarlas mediante algún cambio.

5.1 Casper.

Uno de los grandes problemas de las redes *blockchain* cuyo algoritmo de consenso está basado en PoW, es el consumo energético. Más aún cuando tienes una bomba de dificultad que incrementa, cada vez más deprisa, la dificultad para lograr superar esa prueba de trabajo.

La solución que ha tomado la comunidad de *Ethereum*, ha sido el desarrollo de un algoritmo de consenso basado en PoS que ha recibido el nombre de "Casper". En la introducción ya se explicó el funcionamiento de los algoritmos de consenso basados en PoS.

Sin embargo, se van a detallar algunos de los detalles del funcionamiento que tendrá el protocolo y de qué manera se penalizará a los mineros maliciosos.

En primer lugar, todo aquel que quiera tener acceso al minado de bloques, deberá entregar una cantidad de *ether* como si de una fianza se tratara. Una vez pasen a formar parte del grupo de mineros, serán elegidos de manera aleatoria para proponer un nuevo bloque de la cadena. El resto de mineros, decidirán si el bloque es válido o no realizando una apuesta sobre el bloque que consideren válido.

Una vez decidido el bloque a incluir en la cadena, se recompensará en proporción a la apuesta realizada a los mineros que hayan apoyado al bloque vencedor. De igual manera, se penalizará reduciendo o incluso eliminando su *stake* a los mineros que se detecte que están actuando de manera malintencionada.

Todo esto orquestado por un *smart contract*. Casper no será otra cosa que un contrato inteligente con un conjunto de reglas definidas.

La inclusión de Casper se realizará en dos fases. La primera, será una versión hibrida entre PoS y PoW, *Casper the Friendly Finality Gadget* (Casper FFG), impulsada por el propio Vitalik Buterin para facilitar el traspaso de PoW a PoS.

La segunda versión será del algoritmo de consenso será de tipo PoS, *Casper the Friendly GHOST: Correct-by-Construction* (Casper CBC), y está liderada por el desarrollador Vlad Zamfir. (19)

5.2 Sharding.

Sin embargo, para poder instaurar la versión puramente PoS del algoritmo de consenso, se requiere incrementar la escalabilidad de la plataforma. De hecho, aunque no se produjera este cambio en relación al tipo de algoritmo de consenso aplicado, la escalabilidad sería otro de los puntos de mejora detectados por la comunidad.

Como se ha detallado anteriormente, existen tres tipos de nodos en una red *blockchain*, aquellos que descargan parcialmente la información de la cadena de bloques (*light clients*), aquellos que descargan y almacenan la totalidad de los bloques (*full nodes*) y aquellos que además de descargar y almacenar todos los bloques, participan en la creación y anexión de nuevos bloques a la cadena (*miners*).

Realmente, en mayor o menor medida, cada uno de los nodos procesa una a una todas las transacciones. Todos realizan el mismo procesamiento sobre la misma transacción. Por ello, se puede considerar que, pese a ser miles, cientos de miles o millones de computadoras, hacen el trabajo como si de una sola máquina se tratara.

Bien, para esto se ha invertido tiempo en el desarrollo de la propuesta Sharding. En esta idea se pretende dividir la información de la cadena de bloques en pequeñas subcadenas y que cada subcadena sea monitorizada y minada por diferentes grupos de mineros.

De esta forma, se puede paralelizar tanto el procesamiento de nuevos bloques, como el almacenamiento de los mismos. Si bien es cierto, que para ello se "dividiría" la red en varias subredes.

Cada una de estas subredes tendría una cadena de bloques con la misma estructura, las mismas reglas, los mismos protocolos pero, diferentes usuarios, diferentes mineros, diferentes bloques y diferentes estados globales.

El estado global de la red, podría conseguirse a partir de un árbol de Merkle formado por cada uno de los subestados globales. (20)

A raíz de lo tratado en este documento, quedarían pendientes por resolver algunos asuntos como la manera de desplegar una DApp. Al tratarse de subredes y un usuario solo disponer de acceso a su subred, ¿de qué manera se podría desplegar una DApp con la que todas las subredes pudieran interactuar? ¿Sería posible establecer comunicaciones entre subredes?

Estas y otras tantas cuestiones y posibles casos de uso que surjan cuando se despliegue la actualización se irán resolviendo con el tiempo. Mientras tanto, el equipo de

desarrolladores de *Ethereum*, seguirá centrando sus esfuerzos en conseguir una versión estable del protocolo basado en PoS y que incorpore funcionalidades básicas de Sharding.

5.3 Plasma.

Otra solución orientada a incrementar la escalabilidad del protocolo *Ethereum* es Plasma. Esta idea puede resultar semejante a Sharding, pero en realidad su funcionamiento no tiene nada que ver.

La propuesta que se plantea está basada en cadenas secundarias. Una vez desplegado el *smart contract*, por ejemplo de una DApp, se crearía una cadena "hija" enraizada en la cadena principal.

La actividad relacionada con la DApp se iría registrando en la cadena secundaria, mientras que le *smart contract* va creando puntos de control y reportando la información pertinente de estos puntos de control a la cadena principal.

De esta forma, se podrían agrupar las transacciones relacionadas con ciertas DApps, disminuir el tráfico asociado a la cadena principal e incrementar el *throughput* de la red completa, ya que cada cadena secundaria se procesaría en paralelo.

6 CONCLUSIÓN.

Durante la realización del trabajo, el autor ha ido reafirmando lo que señalaba en las motivaciones que le han llevado a escribir el presente documento. Es complicado encontrar publicaciones o manuales que solventen y clarifiquen las dudas que surgen al profundizar en los conceptos de la tecnología.

Como se explicó al inicio del trabajo, es muy común encontrar la misma información en la mayoría de blogs, libros, documentos de página web, *podcasts*, etc. y es ciertamente complicado encontrar documentos que congreguen y detallen los conceptos necesarios para entender y comprender el funcionamiento de la tecnología.

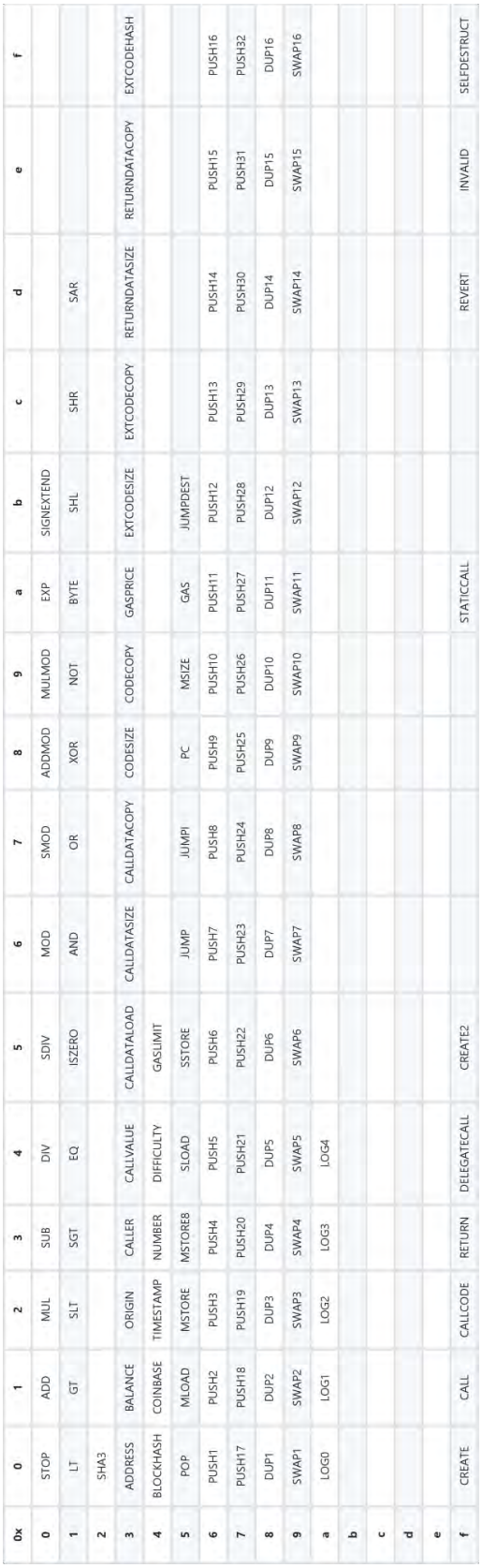
Considero que este trabajo puede suponer una buena herramienta para que un usuario con una pequeña base de conocimientos matemáticos y de programación pueda comprender perfectamente los conceptos fundamentales que determinan el funcionamiento de las redes *blockchain*, de la plataforma *Ethereum* y los *smart contracts*.

Si bien es cierto, para ser adquirir un nivel avanzado de conocimiento de la tecnología se requeriría ahondar en algunos de los conceptos que se han mencionado durante el documento, y en los que típicamente se añadía una nota a pie de página con la página web o el documento al que acudir.

Por consiguiente, y al igual que se ha comentado las ideas de futuro del equipo de desarrollo del protocolo *Ethereum*, el siguiente paso tanto del lector como del autor, pasaría por ahondar en el estudio de alguno de los lenguajes de programación y probar a plantear una solución basada en *smart contracts*.

Para esto último se podría hacer uso de los recursos ofrecidos en la sección de *Solidity*, utilizando Ganache para virtualizar la red, Remix para crear el *smart contract* y Truffle para desplegarlo e interactuar con él.

ANEXO A: *Opcodes* disponibles en EVM.



0x	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	STOP	ADD	MUL	SUB	DIV	SDIV	MOD	SMOD	ADDMOD	MULMOD	EXP	SIGNEXTEND				
1	LT	GT	SLT	SGT	EQ	ISZERO	AND	OR	XOR	NOT	BYTE	SHL	SHR	SAR		
2	SHA3															
3	ADDRESS	BALANCE	ORIGIN	CALLER	CALLVALUE	CALLDATALOAD	CALLDATASIZE	CALLDATACOPY	CODESIZE	CODECOPY	GASPRICE	EXTCODESIZE	EXTCODECOPY	RETURNDATASIZE	RETURNDATACOPY	EXTCODEHASH
4	BLOCKHASH	COINBASE	TIMESTAMP	NUMBER	DIFFICULTY	GASLIMIT										
5	POP	MLOAD	MSTORE	MSTORE8	SLOAD	SSTORE	JUMP	JUMPI	PC	MSIZE	GAS	JUMPDEST				
6	PUSH1	PUSH2	PUSH3	PUSH4	PUSH5	PUSH6	PUSH7	PUSH8	PUSH9	PUSH10	PUSH11	PUSH12	PUSH13	PUSH14	PUSH15	PUSH16
7	PUSH17	PUSH18	PUSH19	PUSH20	PUSH21	PUSH22	PUSH23	PUSH24	PUSH25	PUSH26	PUSH27	PUSH28	PUSH29	PUSH30	PUSH31	PUSH32
8	DUP1	DUP2	DUP3	DUP4	DUP5	DUP6	DUP7	DUP8	DUP9	DUP10	DUP11	DUP12	DUP13	DUP14	DUP15	DUP16
9	SWAP1	SWAP2	SWAP3	SWAP4	SWAP5	SWAP6	SWAP7	SWAP8	SWAP9	SWAP10	SWAP11	SWAP12	SWAP13	SWAP14	SWAP15	SWAP16
a	LOG0	LOG1	LOG2	LOG3	LOG4											
b																
c																
d																
e																
f	CREATE	CALL	CALLCODE	RETURN	DELEGATECALL	CREATE2					STATICCALL			REVERT	INVALID	SELFDESTRUCT

Fig. 13. ***Opcodes*** **disponibles en la EVM**
Tomada de: (25)

ANEXO B: Escala de unidades de la criptomoneda de Ethereum.

Unit	Wei Value	Wei
wei	1 wei	1
Kwei (babbage)	1e3 wei	1,000
Mwei (lovelace)	1e6 wei	1,000,000
Gwei (shannon)	1e9 wei	1,000,000,000
microether (szabo)	1e12 wei	1,000,000,000,000
milliether (finney)	1e15 wei	1,000,000,000,000,000
ether	1e18 wei	1,000,000,000,000,000,000

Fig. 14. Escala de medida – ether

Tomada de: **(26)**

ANEXO C: Project summary.

Just a decade ago, a new technology was born, named blockchain. As well as internet revolutionized our lifestyle, it is expected that blockchain will do it as well.

The internet that we know is called "the internet of information" because its potential is based on the processing of data, while blockchain sets the path to "the internet of value".

This new technology allows the users to deal with digital assets, representing a value in real world, in a secure way. Moreover, blockchain will allow us to disintermediate the relationships between parts.

Currently, the regulatory framework is very scattered. It depends on the government and whether one legislation or another applies. For example, there are countries like Morocco, Nepal or Pakistan that oppose to the adoption of the technology, prohibiting all the activity related to it.

Others, like China, set obstacles to obstruct the adoption of the technology. In the specific case of China, the government does not allow the financial entities to give support to transactions related to cryptocurrencies.

On the other hand, there are countries like Italy, which have propitious regulatory frameworks. In this case, the Italian government is approving a proposal to give legal validity to all blockchain activities. Lastly, in Spain the activities related to cryptocurrency are not protected by any guarantee. Nevertheless, taxes apply in several cases, as the IRPF with capital gains or losses.

In any case, the regulatory framework is the least important aspect of the document. The most important is the technology and how it works.

Blockchain is very similar to a Distributed Ledger. It's made up by a chain of blocks cryptographically related to each other. A block can only be added to the chain when it satisfies a criteria included in the consensus algorithm, which is defined by the protocol.

It's very important to differentiate between the protocol and the cryptocurrency. The protocol is the code that defines the behavior rules of the blockchain, while the cryptocurrency is the value's item that are used by the transactions.

There are different blockchains with different purposes. The two most relevant projects are Bitcoin and Ethereum. On one hand, Bitcoin was the first and showscase an alternative to the traditional finance system, where payments and transactions could be

made in a secure way. On the other hand, Ethereum improves the idea increasing the intelligence of the code and making it simple to program.

Although it may sound very innovative, it is the congregation of many theoretical ideas that had been proposed by cryptographers and programmers but they couldn't put them into practice.

Some of this people and their ideas are: David Chaum, who talks about digital anonymous money and the double-spend problem; the Cypherpunks, who talks about the necessity of a protocol which allows anonymity in communications, what led into the PoW to prevent the spam; Adam Back, who propose a way to make the PoW, Hashcash; and the most relevant, Nick Szabo, who started talking about smart contracts or hashes chains.

Nobody knows its real identity but, in 2008, 'Satoshi Nakamoto' published Bitcoin's whitepaper. A proposal to send money without the interference of any banking entity that solves the problem of double-spend in peer-to-peer networks. The amount of the cryptocurrency was fixed from the beginning in 21 millions of units. The cryptocurrency of the Bitcoin protocol has the same name of it, bitcoin.

But the real magic was below Bitcoin. The real magic was the technology that supports the operations, allow to decentralize the network and all of it in a secure way. The technology was blockchain.

Blockchain networks are peer-to-peer networks where, when a new user wants to become part of the network, it has to connect with any of the nodes, download all the information of the blockchain and validate each of the blocks.

All of the information stored in the blockchain is encrypted and added by transactions that are aggregated in blocks. The blocks are cryptographically related to each other and because of that the data becomes immutable, secure and it will allows to offer transparency when an audit is required.

To add blocks, miners are needed. Miners are the ones who get the transactions, create the blocks and try to satisfy the consensus algorithm. Moreover, they have to validate all the transactions they receive. That implies that every transaction is validated at least one time for each miner of the network.

There are three types of blockchain networks: permissionless, public and shared systems, where any user can participate and where every node is the same; permissioned, public and shared systems, where only a group of specific users can be miners but everyone can be part of the network; and permissioned, private and share system, also known as

private networks, because you need permissions to be member and because there are no miners if not validators.

The most important thing to secure the blockchains is cryptography. Specifically, hash functions and asymmetric cryptography. Every transaction is signed by its founder.

To interact with a blockchain a wallet is needed. A wallet is a sequence of bits that denote the address, where the cryptocurrencies are stored. It does not represent any location of the real world, everything is in the network.

To create that wallets or accounts and to sign the transactions asymmetric cryptography is needed. Bitcoin uses an algorithm based on elliptic curves, Elliptic Curve Digital Secure Algorithm (ECDSA), in the same way as Ethereum.

In order to add a new block to the chain, it is needed to satisfy the consensus algorithm of the protocol. The most knowns are Proof of Stake and Proof of Work. The first, and the least used, requires a stack to the miners and manage it as the reputation. If the miners do their job well, they will received a reward but, if a malicious activity is detected, the miner will lose its reputation, the stake.

The other one, Proof of Work is the most implemented. The protocol requires the demonstration of having completed a challenge as the proof of having worked. Bitcoin, as well as Ethereum, base their PoW in hashes. The goal is to reach a hash with specified difficulty. When the proof is passed, the block is proposed to the neighbors, then to their neighbors and so on. If everything is correct, the block will be added to the blockchain, in any other case, the block will be discarded.

Blockchain is a flexible technology for two reasons. The first, it offers a possibility to obtain financing through Initial Coins Offering (ICOs). This tool consists in offering tokens of the new protocol in presale. A token in this case is referred to as cryptocurrency. In this way, it allows to get financing to implement the solution and deploy it.

The second, because when a bug it's detected in the code of the protocol or the network suffers a cyber-attack, it's possible to fix it. There are two ways of to execute the update of the protocol, through a soft fork, when the old rules are compatible with the news, and through a hard fork, when the new rules are not compatible with the old. The latter may imply the creation of a new cryptocurrency, as occurred in the hard fork of The DAO, which divided the Ethereum blockchain between Ethereum and Ethereum Classic.

One interesting point that is not fully implemented are light clients. The idea is to allow a node to interact with the blockchain without the obligation of download the entire blockchain.

Bitcoin introduced all of this concepts, but it was not enough for many developers of the protocol itself. In December of 2013, Vitalik Buterin, a Russian young boy of 19 years old, published the whitepaper of Ethereum.

The goal of the new protocol was, and is still today, to be a platform to design programming code base on *smart contracts* in a simple and a secure way. The protocol should allow the users to propose new improvements and, if approved, the network should incorporate it.

This proposals can be published and discussed through the EIPs or "Ethereum Improvement Proposals". There are three types of EIPs: standard track EIPs, which include EIPs related to the Core, Networking, Interface and ERC (Ethereum Request for Comments); Informational EIPs; and Meta EIPs.

For an EIP to be approved, it must first agree with the community, be discussed, be validated by the editors and, finally, be drafted under the standard defined in the EIP1.

Since the whitepaper of Ethereum was published, the protocol has evolved in different phases and it's predicted to cross at least two more.

When the protocol was deployed the 30th of July 2015, the Frontier stage began. It was supposed to be a very basic network, but it allowed the developers to interact with smart contracts, mine new blocks and more. In this stage was introduced the difficulty bomb, that increase, more and more quickly, the difficulty of the PoW.

Homestead release was introduced the 14 of March of 2016. It was considered to be the first stable stage of the protocol. An interface that allowed less advanced user profile to interact with the blockchain was launched, Mist.

In this phase, The DAO was born, an organization that allowed to invest in new projects that were considered interesting and that aimed to be a benchmark in the future. However, the code that controlled it had a bug that a user took to steal money, between 50 and 100 millions of dollars.

Because of this, a hard fork was implemented and the Ethereum blockchain was split into two, the one who kept the same rules (Ethereum Classic) and the one who rewrote the past and sent the stolen money to their real owners (Ethereum).

In this phase, two other updates known as Tangerine Whistle and Spurious Dragon were made to increase the security of the protocol, by increasing the cost to execute some actions on the blockchain.

The third stage was Metropolis, but there were so many changes needed, so the developers of Ethereum decided to divide it into two new phases, Byzantium and Constantinople.

Byzantium arrived the 16 of October of 2017, introducing the precompile of the smart contracts to allow the inclusion of zkSNARks to improve the privacity of the network and make available new opcodes to optimize the execution of the smart contracts.

Constantinople is still pending to be updated. It is planned for the next 25th of February. It will include bitwise operations, new opcodes to facilitate the implementation of light clients and reduce the reward that the miners receive when get to mine a block.

In both substages, the difficulty bomb was delayed. The idea was to replace the consensus algorithm based on PoW with a new one based on PoS (Casper). The reason why the difficulty was delayed, was that the programming of the version of an Ethereum network based on PoS had been more complicated than expected.

The last release will be Serenity. There is only one known improvement, it will be the stage when the protocol will be update to PoS. The rest of the concepts are currently unknown.

All the opcodes that have been mentioned are executed in the Ethereum Virtual Machine (EVM). Every block, every transaction, every smart contract is executed in the Ethereum Virtual Machine, that is made of a stack, the storage, the memory and a space named calldata, where the parameters of functions are stored.

The EVM needs information from the smart contracts and the accounts. In Ethereum there are two kinds of accounts, externally owned accounts, which are controlled by humans, and contract accounts, which are controlled by smart contracts. The most important property is the balance that contains the amount of ether available. Moreover, the nonce of the account indicate how many transactions have been signed by this account.

A special kind of fuel is needed to run the Ethereum Virtual Machine. Its name is gas and its value is related to Ether. Every opcode, every function has a cost associated and if the code sent to the EVM requires more gas to execute than the available gas the user has sent, it will throw an out-of-gas exception.

The smart contracts can be programmed with Solidity, a programming language specifically designed to make smart contracts. It includes a lot of basic data types, functions, constructors, events, modifiers of functions and variables and predefined functions.

And this, the smart contracts, is the thing that make Ethereum so powerful. A smart contract is autonomous, allows to reduce costs, increases the speed of execution and increases the security because information is encrypted and immutable. But the developers have to be very carefully while programming, because a small bug can derivate in such case in a disaster as the occurred with The DAO.

The use cases are so many as you can think. It can be applied in health sector to storage the medical history of the patients, in insurance sector to design insurances based on smart contracts without intermediates, in supply chains to increase the traceability of

the products, to manage the information of IoT devices, to implement financial services or to implement new blockchains.

The latter is implemented in Ethereum by tokens that can be defined as an EIP. At the end, it is a solution to implement a blockchain on the blockchain of Ethereum, using the Ethereum Virtual Machine and paying the gas in ether. An example of this casuistry is the token ERC20 that can be taken as a template to create new blockchains on the Ethereum blockchain.

Another token ERC could be the ERC721, which generates tokens, each different from the other. Truly, it consists in a game when you can buy and sell cryptokitties and raising new cryptokitties.

To implement all the use cases an extra-blockchain interaction is needed. This is made by oracles, which allow the smart contract to receive information from a source like, for example, if a bet is to be implemented, a web that contains the result of a game is needed to check it and decide who the winner is.

Smart contracts can be presented in different ways. From the simplest to the most complex, the smart contracts *per se*, Decentralized Applications (DApps) and Decentralized Autonomous Organizations (DAOs).

DApps can be a puzzle of various smart contracts that interact between them under certain conditions and emulated a common application of the real world, but with the components deployed in blockchain or any decentralized protocol or network.

The last are DAOs, that try to copy the organization of a regular company but managed completely with smart contracts.

Ethereum protocol is constantly evolving. The next steps fixed by the Ethereum community is to update to PoS (Casper) and to find a solution for the scalability problem (Sharding or Plasma).

LISTA DE BIBLIOGRAFÍA.

1. **EEUU, Librería del Congreso de los.** Library of Congress. *Regulation of Cryptocurrency Around the World.* [En línea] 01 de 11 de 2018. [Citado el: 11 de 02 de 2019.] https://www.loc.gov/law/help/cryptocurrency/world-survey.php.

2. **Diedrich, Henning.** *ethereum: blockchains, digital assets, smart contracts, decentralized autonomous organizations.* Primera. s.l. : Wildfire Publishing, 2016. pág. 360. 9781523930470.

3. **Szabo, Nick.** first monday. *Peer-reviewed journal on the internet.* [En línea] 01 de 09 de 1997. [Citado el: 05 de 02 de 2019.] https://firstmonday.org/ojs/index.php/fm/article/view/548/469-publisher=First.

4. **Preukschat, Alexander, y otros.** *Blockchain: la revolución industrial de internet.* Sexta. Barcelona : Gestion 2000, 2018. pág. 288. 978-84-9875-447-6.

5. **Molero, Iñigo.** Libro Blockchain. *ECDSA.* [En línea] 31 de 05 de 2017. [Citado el: 05 de 10 de 2018.] https://libroblockchain.com/ecdsa/.

6. **Buterin, Vitalik.** Vitalik Buterin's website. [En línea] 9 de 06 de 2017. [Citado el: 22 de 09 de 2018.] https://vitalik.ca/general/2017/06/09/sales.html.

7. —. Ethereum Blog. *Light Clients and Proof of Stake.* [En línea] 09 de 01 de 2015. [Citado el: 15 de 11 de 2018.] https://blog.ethereum.org/2015/01/10/light-clients-proof-stake/.

8. —. About me. *Vitalik Buterin.* [En línea] 15 de 02 de 2017. [Citado el: 10 de 02 de 2019.] https://about.me/vitalik_buterin.

9. **Buterin, Vitalik y Ray, James.** ethereum/wiki. [En línea] 22 de 08 de 2018. [Citado el: 12 de 11 de 2018.] https://github.com/ethereum/wiki/wiki/White-Paper#introduction-to-bitcoin-and-existing-concepts.

10. **Becze, Martin y Hudson, Jameson.** Propuestas de mejora de Ethereum. *EIP 1: Propósito y directrices de EIP.* [En línea] 01 de 02 de 2017. [Citado el: 01 de 26 de 2019.] https://eips.ethereum.org/EIPS/eip-1.

11. **Tani, Takenobu.** github. *takenobu-hs.* [En línea] Marzo de 2018. [Citado el: 13 de Enero de 2018.] https://takenobu-hs.github.io/downloads/ethereum_evm_illustrated.pdf.

12. **Wood, Gavin.** ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER. [En línea] 12 de 10 de 2018. [Citado el: 13 de 10 de 2018.] https://ethereum.github.io/yellowpaper/paper.pdf.

13. **Vogelsteller, Fabian y Buterin, Vitalik.** EIP 20: ERC-20 Token Standard. [En línea] 15 de 10 de 2018. [Citado el: 17 de 01 de 2019.] https://eips.ethereum.org/EIPS/eip-20.

14. **Dexaran.** Github/Dexaran/ERC223-token-standard. [En línea] 14 de 05 de 2018. [Citado el: 18 de 01 de 2019.] https://github.com/Dexaran/ERC223-token-standard.

15. **Dafflon, Jacques, Baylina, Jordi y Shababi, Thomas.** EIP 777: A New Advanced Token Standard . [En línea] 20 de 11 de 2017. [Citado el: 18 de 01 de 2019.] https://eips.ethereum.org/EIPS/eip-777.

16. **Entriken, William, y otros.** EIP 721: ERC-721 Non-Fungible Token Standard. [En línea] 24 de 01 de 2018. [Citado el: 19 de 01 de 2019.] https://eips.ethereum.org/EIPS/eip-721.

17. **Ethereum Foundation.** Solidity. *Units and Globally Available Variables.* [En línea] 22 de 01 de 2019. [Citado el: 18 de 02 de 2019.] https://solidity.readthedocs.io/en/v0.5.3/units-and-global-variables.html.

18. **Vessenes, Peter.** More Ethereum Attacks: Race-To-Empty is the Real Deal. [En línea] 09 de 01 de 2016. [Citado el: 17 de 02 de 2019.] https://vessenes.com/more-ethereum-attacks-race-to-empty-is-the-real-deal/.

19. **Gibson, Adam.** TLSNotary. *A new kind of auditing - cryptographic proof of online accounts.* [En línea] 08 de 11 de 2014. [Citado el: 03 de 02 de 2019.] https://tlsnotary.org/.

20. **Elgarte, Federico.** Oráculos en Solidity: requests por fuera de la Blockchain con Oraclize. [En línea] 10 de 03 de 2018. [Citado el: 02 de 02 de 2019.] https://medium.com/@shuffledex/or%C3%A1culos-en-solidity-requests-por-fuera-de-la-blockchain-con-oraclize-abcd9863fe80.

21. **Elolaimi, Khalid.** Intel® Software Guard Extensions (Intel® SGX) Webinar. [En línea] 20 de 04 de 2017. [Citado el: 10 de 02 de 2019.] https://software.intel.com/sites/default/files/managed/81/61/intel-sgx-webinar.pdf.

22. **Hussey, Matt.** Decrypt. *Dapps. The Tech - Intermediate.* [En línea] 21 de 01 de 2019. [Citado el: 18 de 02 de 2019.] https://decryptmedia.com/resources/dapps.

23. **Rosic, Ameer.** Blockgeeks. *What is Ethereum Casper Protocol? Crash Course.* [En línea] 21 de 11 de 2017. [Citado el: 20 de 02 de 2019.] https://blockgeeks.com/guides/ethereum-casper/#What_is_Ethereum_Casper_Protocol_Crash_Course.

24. —. Blockgeeks. *What are Ethereum Nodes And Sharding?* [En línea] 05 de 03 de 2018. [Citado el: 02 de 20 de 2019.] https://blockgeeks.com/guides/what-are-ethereum-nodes-and-sharding/.

25. **Hollander, Luit.** Medium. *The Ethereum Virtual Machine - How does it work?* [En línea] 29 de 01 de 2019. [Citado el: 20 de 02 de 2019.] https://medium.com/mycrypto/the-ethereum-virtual-machine-how-does-it-work-9abac2b7c9e.

26. **Ethereum Foundation.** Ethereum Homestead. *Ether.* [En línea] 31 de 01 de 2016. [Citado el: 20 de 02 de 2019.] http://ethdocs.org/en/latest/ether.html.

27. **Lunaticoin**. L08: Cris Carrascosa y la Fiscalidad Cripto. [*podcast* En línea] Lunaticoin. Disponible en: https://anchor.fm/lunaticoin/episodes/L08-Cris-Carrascosa-y-la-Fiscalidad-Cripto-e32bf9 [consulta 31 Jan. 2019].