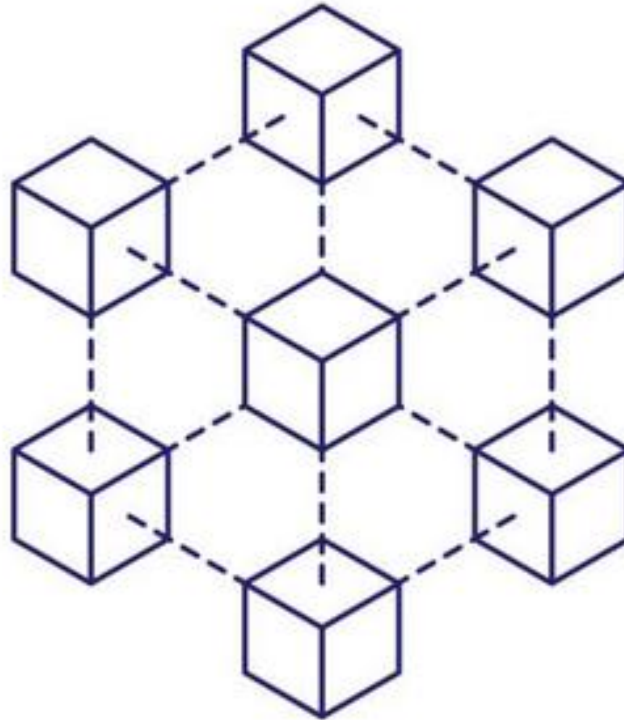
BLOCKCHAIN



JESÚS RUBIO MARTÍN

IES GONZALO NAZARENO

ADMINISTRACIÓN DE SISTEMAS EN RED

CURSO 2020-2022

Contenido

1. Introducción 3
2. ¿Quién inventó las cadenas de bloques? 3
3. ¿Qué es un bloque? 5
4. El Hash 7
5. Libro Mayor 16
6. Redes P2P 18
7. El Minado 21
8. Tolerancia a Fallos Bizantinos 24
9. Protocolos de Consenso 26
10. Creación de una Blockchain en Python 29
11. El Rango del Nonce 34
12. Ataque del 51% 36
13. Claves Públicas y Privadas 38
14. Smart Contracts 39
15. Smart Contract en Solidity 40
16. Conclusión 44

1. Introducción

El documento que está leyendo es el proyecto final para la finalización del grado superior de administración de sistemas en red. En él he tratado de explicar en que consiste una tecnología en constante auge como es el blockchain.

A parte de la correspondiente teoría que debemos conocer para entender el blockchain, también explico programas en Python creando nuestra primera cadena de bloques, así cómo un contrato inteligente programado en Solidity.

Hecha esta breve introducción, creo que ya podemos dar comienzo al objetivo del documento.

2. ¿Quién inventó las cadenas de bloques?

El concepto de blockchain surge en 1991 tras la redacción de un documento denominado "*How to Time-Stamp a Digital Document*" redactado por Stuart Haber y W. Scott Stornetta.

"*La perspectiva de un mundo en el que todos los documentos de texto, audio, imagen y vídeo estén en digital en plataformas fácilmente modificables plantea la cuestión de cómo certificar cuándo se creó o modificó por última vez un documento. El problema es cómo podemos poner un sello de tiempo a los datos, no al soporte. En este informe proponemos procedimientos computacionalmente prácticos para el sellado de tiempo digital de tales documentos, de modo que sea inviable que un usuario retrase o adelante la fecha de su documento, incluso con la confabulación de un servicio de sellado de tiempo. Nuestros procedimientos mantienen la total privacidad de los documentos y no requieren ningún registro por parte del servicio de sellado de tiempo.*"

Como vemos en la introducción de este informe Haber y Stornetta nos van a proponer soluciones para que un archivo que recorre internet pueda tener un sello timestamp inmodificable por el autor del documento. También nos hablan de la importancia que podría tener esto en campos como la creación de patentes evitando conflictos por cual pudo generarse antes.

Nos informan que el método de sellado de documentos digitales necesita de dos propiedades básicas:

- En primer lugar, hay que encontrar una manera de marcar el tiempo, sin depender de las características del soporte en el que aparecen los datos, de modo que sea imposible cambiar incluso un bit del documento sin que el cambio sea evidente

- En segundo lugar, debería ser imposible estampar un documento con una fecha y hora diferentes a las reales.

La propuesta que nos hacen ambos ingenieros en dicho documento es la siguiente:

En el caso de que queramos realizar un sellado de tiempo a nuestro documento, deberemos pasarlo por una función la cual nos devolverá un hash. Esta función debe ser muy sensible ya que con sólo cambiar un bit del documento el hash deberá ser totalmente distinto. Una vez tenemos nuestro documento con la información correspondiente y su hash lo enviaremos a un servicio de sellado de tiempo. Este servicio firmará con su firma digital nuestro documento y nos lo devolverá. De esta manera los documentos serán sellados temporalmente y no podrán ser modificados ya que obtendríamos un hash totalmente diferente.

Sin embargo, no es hasta 2008 tras el lanzamiento del primer informe redactado bajo el pseudónimo de Satoshi Nakamoto cuando la tecnología del blockchain empieza obtener relevancia.

Este informe aporta ideas para transaccionar dinero sin necesitar terceras entidades. Aunque la idea de la firma digital de Haber y Stornetta es una parte de la solución, Nakamoto cree que perdemos todos los beneficios al necesitar a una tercera entidad en la que confiar nuestros datos y aumentando los gastos debido a la compra de los servicios de esta.

Es por ello que Nakamoto propone la idea de introducir una red peer-to-peer. Esta red tendrá que disponer de las siguientes características:

- Las nuevas transacciones se difunden a todos los nodos.
- Cada nodo recopila las nuevas transacciones en un bloque.
- Cada nodo trabaja para encontrar una prueba de trabajo difícil para su bloque.
- Cuando un nodo encuentra una prueba de trabajo, difunde el bloque a todos los nodos.
- Los nodos aceptan el bloque sólo si todas las transacciones que contiene son válidas.
- Los nodos expresan su aceptación del bloque trabajando en la creación del siguiente bloque en la cadena, utilizando el hash del bloque aceptado como hash anterior.

Como vemos Nakamoto nombra el concepto de prueba de trabajo introducido en 2004 por el informático Hal Finney. La prueba de trabajo consiste en lo siguiente:

Una vez hemos obtenido el hash de nuestro bloque mediante una función de hash, como podría ser sha-256, introduciremos una nueva variable para que ese bloque sea añadido a nuestra cadena de bloques. El hash de nuestro bloque deberá empezar por una cifra de ceros y mediante la prueba de trabajo deberemos adivinar el número, nonce, que junto con la información del bloque y el hash previo del bloque anterior conforman el hash del bloque que queremos añadir a nuestra cadena.

Una vez se descubra el nonce, los otros nodos deberán comprobar que efectivamente es el número correcto para formar el hash del bloque. Una vez los nodos estén mayoritariamente de acuerdo el bloque será añadido a la cadena.

3. ¿Qué es un bloque?

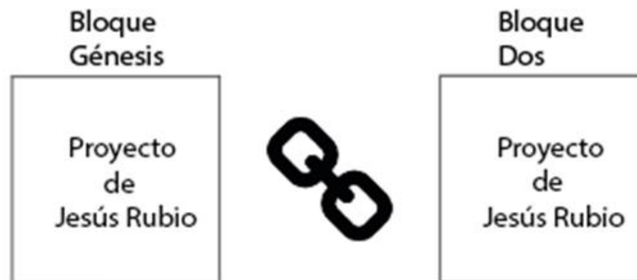
Un bloque es un registro el cual contiene datos en su interior. A dicho bloque, le vamos a añadir el hash, término que explicaremos más en profundidad, del bloque anterior. De esta manera todos los bloques estarán enlazados entre sí. Por lo tanto, contamos con la información que contiene el bloque y el hash del bloque anterior, a partir de estas variables podremos generar el hash, el identificador, de nuestro bloque.

Vamos a explicar este concepto para que quede mucho más claro a través de representación gráfica.

Lo primero que debe tener una cadena de bloque es el bloque génesis o el bloque generador. Este será el bloque a partir del cual se generarán los siguientes, este bloque siempre será el primer bloque. La principal característica de este bloque es su hash previo, ya que al ser el primero estará formado por todo ceros. A modo de ejemplo nuestro bloque generador constará de un texto con información, su hash previo y su hash propio.

Datos: "Proyecto de Jesús Rubio
Hash previo: 0000000000000000
0000000000000000
0000000000000000
0000000000000000
Hash: 0ba0c3736bd5874e66c933
62766c560e179073421e76
257c4e1eca2187e6b83c

A partir de nuestro bloque génesis llegarán los siguientes. Siempre guardando el hash del bloque previo, es decir, están enlazados criptográficamente. Si ahora añadiéramos un segundo bloque a nuestra cadena de ejemplo, este bloque debería conservar íntegramente el hash del bloque inminentemente anterior, en este caso el bloque génesis.

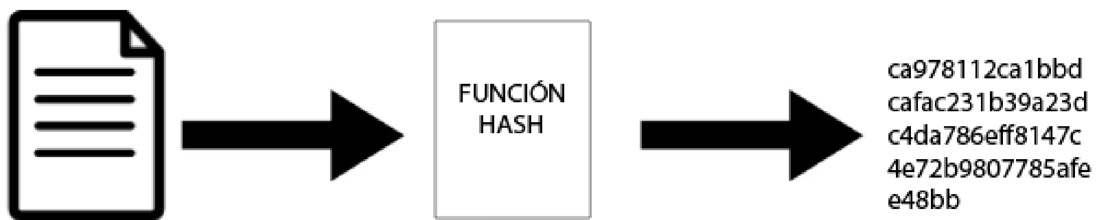


Datos: "Proyecto de Jesús Rubio
Hash previo: 0000000000000000
0000000000000000
0000000000000000
0000000000000000
Hash: 0ba0c3736bd5874e66c933
62766c560e179073421e76
257c4e1eca2187e6b83c

Datos: "Proyecto de Jesús Rubio
Hash previo: 0ba0c3736bd5874e66
c93362766c560e1790
3421e76257c4e1eca2
187e6b83c
Hash: ca978112ca1bbdcafac231b3
9a23dc4da786eff8147c4e72
b9807785afee48bb

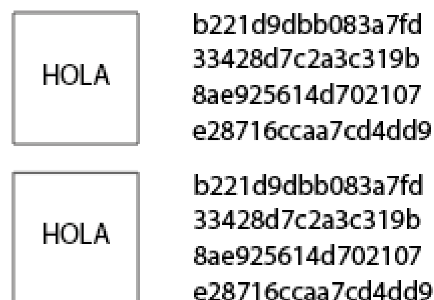
4. El Hash

Un hash es un número hexadecimal de 64 dígitos con el cuál podemos identificar cualquier objeto digital (documento, video, foto, audio, etc). Este número es la salida que proporciona una función hash al nosotros introducirle como parámetro de entrada el objeto que queremos codificar, y eso es lo que queremos hacer codificar, no queremos comprimir o empaquetar nuestro documento en 64 dígitos, queremos obtener un identificador para nuestro documento.

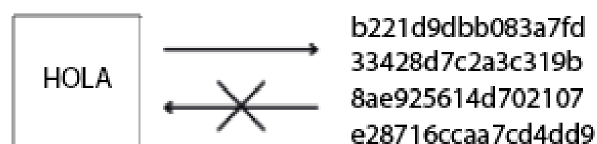


Propiedades deseables en una función hash.

- Permite ejecutarse sobre contenido digital de cualquier tamaño y formato. Es conveniente que podamos obtener un hash sin importar el tipo de objeto que queramos hashear.
- Determinista, es decir, para el mismo mensaje o conjunto de datos de entrada siempre se obtiene el mismo resultado.



- Unidireccional, reconstruir el mensaje original a partir del resultado de la función hash debe ser extremadamente costoso, sino imposible.



- Efecto avalancha, la mínima variación en el mensaje original ha de producir un hash totalmente distinto.

HOLA

b221d9dbb083a7fd
33428d7c2a3c319b
8ae925614d702107
e28716ccaa7cd4dd9

HOLA.

3b952738efe124f6b
34181cb9e60843274
1ab05df195462dc98
b26f50f9ef217

- Cómputo elevado, la calculación del hash debe ser casi instantánea sin importar el peso del documento de entrada.
- Soporte de colisiones, puede ser que ficheros con diferente contenido puedan tener el mismo hash. Aunque el número de hashes que disponemos es gigantesco el número de ficheros distintos es aún más grande por lo que puede darse el caso de que ficheros tengan el mismo hash.

Una de las funciones de hash más usadas actualmente es la conocida SHA-256 diseñadas por la Agencia de Seguridad Nacional (NSA) en 2001. Como ya hemos mencionado anteriormente el objetivo de esta función es resumir cualquier objeto digital en 256 bits de longitud expresados en un número hexadecimal de 64 caracteres denominado hash.

El procedimiento que realiza esta función hash cuando recibe un objeto de entrada es el siguiente:

Lo primero que va a realizar es pasar el mensaje de entrada a hexadecimal. Para ello mirará el valor de cada carácter en la tabla ASCII y será ese número el que pasará a hexadecimal.

Carácter =>	ASCII =>	Hexadecimal
J	74	4A
e	101	65
s	115	73
ú	163	A3
s	115	73

Ahora debemos unir los valores en hexadecimal de nuestro mensaje:

Jesús => Hexadecimal => 4A6573A373

Gonzalo Nazareno => Hexadecimal => 476F6E7A616C6F204E617A6172656E6A

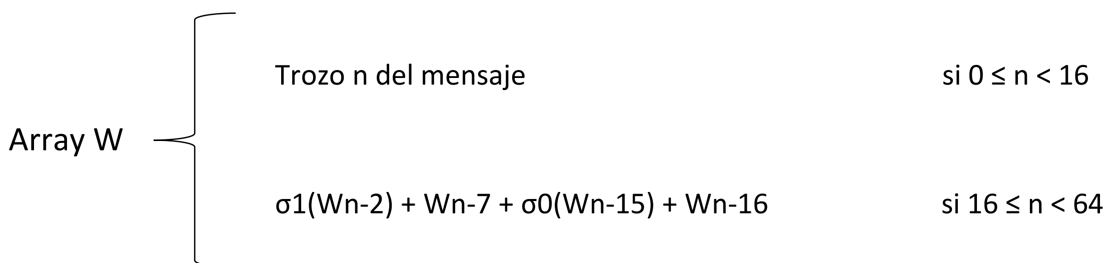
Nótese que los espacios en blanco también ocupan un valor en la tabla ASCII, en concreto el valor 32.

El valor en hexadecimal de nuestro mensaje será guardado en una variable que será usada posteriormente por otros procedimientos.

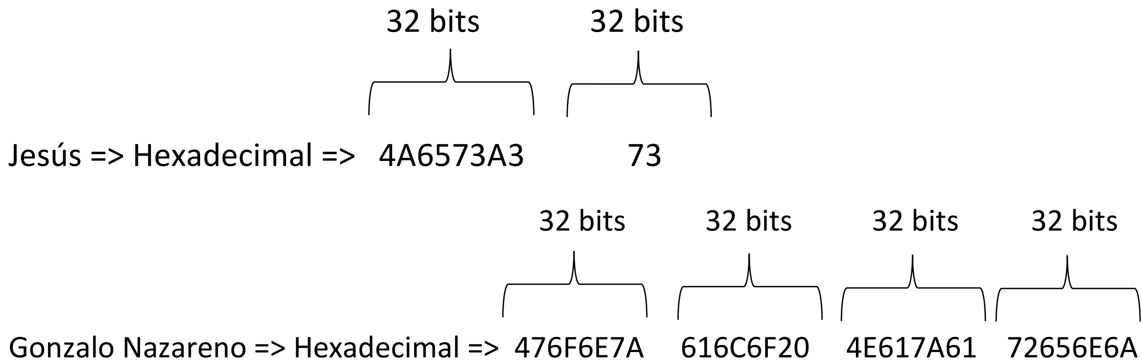
A continuación, calculamos la longitud de nuestro mensaje en bits multiplicando por 8 bits, el valor de cada carácter ASCII. También almacenaremos este valor en otra variable.

Mensaje =>	Bits x Nº de caracteres =>	Longitud en Hexadecimal
Jesús	8 x 5 = 40	28
Gonzalo Nazareno	8 x 16 = 128	80

Una vez tenemos el mensaje y la longitud de este el algoritmo pasará a construir la siguiente variable, la cual se trata de un array, denominada W, de 64 elementos que contiene palabras hexadecimales de 32 bits. Esta variable será inicializada por todo ceros y construida por intervalos, siendo el primer intervalo los 16 primeros registros y el segundo intervalo los 48 restantes, será construida a partir de la siguiente fórmula que explicaremos a continuación.



En el primer intervalo se reservarán los 16 primeros registros para almacenar nuestro mensaje de entrada en formato hexadecimal en separaciones de 32 bits:



Como es lógico no todos los bloques ocuparán los 32 bits exactos, es por ello que el algoritmo los dejará vacíos y se encargará de rellenarlos en procesos posteriores. Por lo tanto, ya tenemos nuestro primer intervalo con un tamaño de 512 bits dividido en 16 bloques de 32 bits.

Ahora tomará un bit que represente el número 1 y lo desplazará al bit más a la izquierda de un byte, por último, lo convertirá en hexadecimal:

1 => binario => 00000001 => pasamos el 1 al bit más a la izquierda => 10000000 => hexadecimal => 80

Añadimos el valor que acabamos de calcular a la derecha de nuestro mensaje:

Jesús => 4A6573A3 + 73 + 80

Gonzalo Nazareno => 476F6E7A + 616C6F20 + 4E617A61 + 72656E6A + 80

A continuación, se añaden 0 hasta llegar a los 448 bits y se cierra con dos bloques de 32 bits con la longitud de nuestro mensaje. En caso de que la longitud no ocupe los 32 bits rellenaremos con 0 por la izquierda.

Jesús – 4A6573A3 + 73 + 80 + 000 + 00000000 + 00000000 + 00000000 + 00000000 + 00000000 + 00000000 + 00000000 + 00000000 + 00000000 + 00000000 + 00000000 + 00000000 + 00000000 + 00000028

Gonzalo Nazareno – 476F6E7A + 616C6F20 + 4E617A61 + 72656E6A + 80 + 000000 + 00000000+ 00000000 + 00000000 + 00000000 + 00000000 + 00000000 + 00000000 + 00000000 + 00000000 + 00000000 + 00000080

El algoritmo ahora inicializará dos variables, la constante K formada por 64 palabras hexadecimales y las 8 variables h que igualmente se compone de 8 palabras hexadecimales.

La constante K queda formada por 64 palabras hexadecimales compuestas por los primeros 32 bits en hexadecimal del resultado de la parte fraccionaria, decimales, de la raíz cúbica de cada uno de los primeros 64 números primos.

2 — $\sqrt[3]{2}$= 1.2599210498948732 => Parte fraccionaria => 0.2599210498948732 => Hexadecimal => 0.428A2F98D728A242D2B4 => 32 bits => 428A2F98

3 — $\sqrt[3]{3}$= 1.4422495703074083=> Parte fraccionaria => 0.4422495703074083 => Hexadecimal => 0.7137449123EF5FDF4D8D => 32 bits => 71374491

Esta operación será realizada con los 62 siguientes números primos.

De igual forma calculará las 8 palabras h pero esta vez serán calculadas a través de los 32 primeros bits de la parte fraccionaria en hexadecimal de la raíz cuadrada de los 8 primeros números primos. Estas 8 palabras serán correlacionadas con las variables A, B, C, D, E, F, G y H respectivamente.

A = 6A09E667

B = BB67AE85

C = 3C6EF372

D = A54F F53A

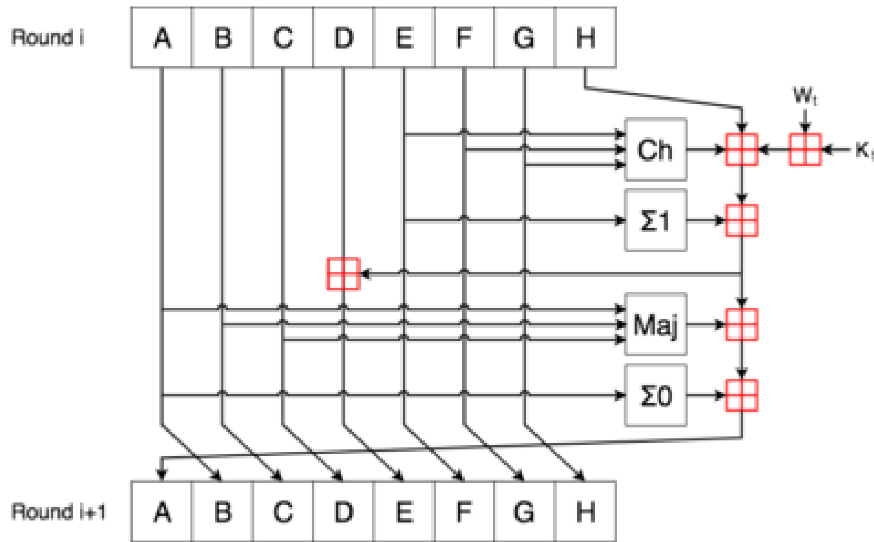
E = 510E527F

F = 9B05688C

G = 1F83D9AB

H = 5BE0CD19

Una vez el algoritmo tiene todos los ingredientes podrá empezar a encriptar nuestro mensaje.



Lo primero que debemos de tener en cuenta es que las variables A, B, C, D, E, F, G y H siempre tendrán el mismo valor en la primera ronda, será a partir de la siguiente cuando empezarán a tomar valores totalmente diferente siguiendo las operaciones que vamos a explicar a continuación.

Variable A

- El valor de A en cada ronda será el que tenga B justo en la siguiente.
- El valor de A en cada ronda será pasado a la función M para obtener el valor de A en la siguiente ronda.
- Por último, también se pasará su valor a la función $\sum 0$ para obtener junto con la función M el valor de A en la siguiente ronda.

Variable B

- El valor de B en cada ronda será el que tenga C justo en la siguiente.
- El valor de B en cada ronda será pasado a la función M para obtener el valor de A en la siguiente ronda.

Variable C

- El valor de C en cada ronda será el que tenga D justo en la siguiente.
- El valor de C en cada ronda será pasado a la función M para obtener el valor de A en la siguiente ronda.

Variable D

- Será uno más de los elementos junto con Wi, Ki, la variable H y E, las funciones ∑1 y Ch. Con la suma en módulo 32 de estos elementos calcularemos el valor de E en la siguiente ronda.

Variable E

- El valor de E en cada ronda será el que tenga F justo en la siguiente.
- El valor de E en cada ronda será pasado a la función ∑ 1 y Ch para obtener el valor de A en la siguiente ronda.

Variable F

- El valor de F en cada ronda será el que tenga G justo en la siguiente.
- El valor de F en cada ronda será pasado a la función Ch para obtener el valor de A en la siguiente ronda

Variable G

- El valor de G en cada ronda será el que tenga H justo en la siguiente.
- El valor de G en cada ronda será pasado a la función Ch para obtener el valor de A en la siguiente ronda.

Variable H

- Será uno más de los elementos junto con Wi, Ki, la variable D y E, las funciones ∑1 y Ch. Con la suma en módulo 32 de estos elementos calcularemos el valor de E en la siguiente ronda.

Función Ch

Es una función que realizará operaciones lógicas tomando como parámetros de entrada las variables E, F y G. Su fórmula es la siguiente:

$$Ch(E,F,G) = (E \land F) \oplus (\neg E \land G)$$

Función Maj

Es una función que realizará operaciones lógicas tomando como parámetros de entrada las variables A, B y C. Su fórmula es la siguiente:

$$Maj(A,B,C) = (A \land B) \oplus (A \land C) \oplus (B \land C)$$

Función ∑0

Es una función que realizará operaciones lógicas y operaciones binarias rotativas tomando como parámetros de entrada la variable A_i. Su fórmula es la siguiente:

$$\sum 0(x) = ROT\ R^{2}(x) \oplus ROT\ R^{13}(x) \oplus ROT\ R^{22}(x)$$

Función ∑1

Es una función que realizará operaciones lógicas y operaciones binarias rotativas tomando como parámetros de entrada la variable Ei. Su fórmula es la siguiente:

$$\sum 1(x) = ROT\ R^{6}(x) \oplus ROT\ R^{11}(x) \oplus ROT\ R^{25}(x)$$

Este proceso se llevaría a cabo durante 64 rondas. Una vez hubiéramos terminado estas rondas sumaríamos los valores que tengan cada una de las variables A,B,C,D,E,F,G,H con el valor que hayan tenido en la primera ronda, esta suma será en módulo 32. Por último, obtendríamos nuestro hash concatenando el valor de las variables A,B,C,D,E,F,G,H.

Por ejemplo, en el caso de calcular el hash para las palabras Gonzalo Nazareno las sumas serían las siguientes:

Suma A

A_0 => 01101010000010011110011001100111

\+

A_{63} => 01101011011111011101100001010100

11010101100001111011111010111011 => Hexadecimal => D587BEBB

Suma B

B_0 => 10111011011001111010111010000101

\+

B_{63} => 11101010100000111101000100001011

10100101111010110111111110010000 => Hexadecimal => A5EB7F90

Suma C

C_0 => 00111100011011101111001101110010

\+

C_{63} => 11010010010000101000011010010011

00001110101100010111101000000101 => Hexadecimal => 0EB17A05

Suma D

D_0 => 10100101010011111111010100111010

\+

D_{63} => 10101010011011010100110111110011

01001111101111010100001100101101 => Hexadecimal => 4FBD432D

Suma E

E_0 => 01010001000011100101001001111111

\+

E_{63} => 11101101100011101000110011001000

00111110100111001101111101000111 => Hexadecimal => 3E9CDF47

Suma F

F_0 => 10011011000001010110100010001100

\+

F_{63} => 11000000001010110000001011010011

01011011001100000110101101011111 => Hexadecimal => 5B306B5F

Suma G

G_0 => 00011111100000111101100110101011

\+

G_{63} => 00100111000111110000011000010111

01000110101000101101111111000010 => Hexadecimal => 46A2DFC2

Suma H

H_0 => 01011011111000001100110100011001

\+

H_{63} => 01100110011011111001101011000101

11000010010100000110011111011110 => Hexadecimal => C25067DE

Una vez tenemos el resultado de las ocho sumas tan sólo nos quedara concatenar los valores para obtener el hash de Gonzalo Nazareno.

Gonzalo Nazareno => A+B+C+D+E+F+G+H => D587BEBBA5EB7F900EB17A054FBD432D
3E9CDF475B306B5F46A2DFC2C25067DE

En caso de que el lector tenga interés por esta tecnología he diseñado una aplicación web la cual calcula automáticamente el hash de la palabra que se introduzca. Para poder acceder a ella puede dirigirse al siguiente enlace: https://jesusrubio3.github.io/hash/

5. Libro Mayor

El libro mayor, también conocido como ledger, es aquel que registra todas las operaciones o transacciones, sin excepción, que se han realizado en la blockchain desde su bloque génesis hasta la actualidad. Es por ello, que el ledger es uno de los elementos fundamentales para la transparencia, seguridad y privacidad en la que se respalda las cadenas de bloques.

Este libro mayor ha sido utilizado por la humanidad desde la aparición de la escritura. Se tiene constancia de que en el Antiguo Oriente Próximo ya eran usados para registrar datos como la producción, el comercio y las deudas. Estos libros han sufrido bastantes evoluciones, una de las más importantes surge en el siglo XIX mediante el cual estos libros pasan a estar centralizados debido al avance de las grandes burocracias. Por lo cual ya resolvimos uno de los conflictos, ya no iban a existir problemas de inconsistencia de datos debido a que solo iba haber un “único libro mayor” basado en la confianza depositada por los ciudadanos en las instituciones centralizadas.

Un siglo más tarde, debido al avance de la tecnología informática, los ledger sufren otra evolución. Aquellos libros escritos en papel iban a pasar a formato digital evitando así perder los datos por causas como incendios, inundaciones o terremotos. Pero hoy día seguimos teniendo el problema que estos libros mayores, convertidos en bases de datos o grandes ficheros, siguen estando centralizados. Digitalizar la información nos permite un análisis, cálculo y velocidad de cómputo incomparable con las versiones en papel, pero estas bases de datos son tan fiables como la organización y las personas por las que son mantenidos.

Es aquí otro de los problemas que se pueden resolver gracias al blockchain. La tecnología de las cadenas de bloques está basada en la filosofía de los libros mayores, pero resolviendo el problema de la dependencia de una autoridad central en la que tengamos que depositar nuestra confianza y sea esta la única responsable de mantener y validar el libro mayor.

Aunque blockchain y ledger son en cierto modo parecidos no podemos caer en el error de creer que son la misma tecnología. El libro mayor es una base de datos en lo que almacenamos toda la información de la red. Esta base de datos no va a estar centralizada en un único servidor, sino que cada nodo de nuestra red va a guardar una copia de la información que almacena dicha base de datos consiguiendo así que sea muy difícil que nuestra red sea atacada con éxito, pues el atacante debería tener la posición de más del 50% de los nodos que forman nuestra red.

Por la otra parte tenemos blockchain que es un tipo de libro mayor segurizado mediante la criptografía y el enlace de los bloques al poseer información del bloque inminentemente anterior. Por lo tanto, conseguimos que si alteramos un solo bits de unos de nuestros bloques de un nodo los demás nodos comprobarán que dicho bloque de dicho nodo tiene una información errónea pues no concuerda con la información que posee la mayoría y automáticamente la base de datos de ese nodo será actualizada.

Los ledgers que podemos encontrarnos pueden ser de dos tipos:

- Ledgers centralizados, los cuales son controlados y validados por una sola entidad. Un ejemplo podría ser el diseño de los bancos o los estados gubernamentales.
- Ledgers distribuidos, pueden ser duplicados y almacenados por cualquier nodo. Su seguridad está asegurada debido a que para introducir un dato debe existir un consenso entre la mayoría de los nodos. Evitamos que una sola entidad posea el control absoluto pues su fuerza posee en la red, la seguridad de la red es directamente proporcional al número de dispositivos que posee la base de datos. Una blockchain con muchos bloques es una cadena más segura.

Por lo tanto, en el caso de que un ciberdelincuente quisiera alterar la información de la base de datos de nuestra cadena, es decir, los datos que contiene uno de nuestros bloques, debería asegurarse de modificar el hash del bloque que va a corromper pues al permutar el contenido cambiará el hash.

Supongamos que lo consigue. El ciberdelincuente ha logrado crackear nuestro sistema y modificar la información de un bloque y su hash, pero le surge un nuevo problema, al modificar el hash de dicho bloque la cadena se corrompe ya que el bloque posterior no le cuadra el valor del hash previo, el bloque que ha sido alterado. Vemos aquí la potencia de la seguridad que guarda la tecnología de bloques, no solo tendría que modificar el hash de ese bloque sino el valor de todos los hashes posteriores a ese bloque.

Aunque el lector se puede dar cuenta que en el ejemplo anterior no estamos exponiendo la ventaja fundamental de la que hemos hablado en este apartado. Una cadena de bloques puede ser más difícil de alterar que una base de datos convencional o un fichero Excel, pero seguimos teniendo el mismo problema si no es más respaldada por otros nodos. Es por ello que la gran mayoría de blockchains usan el sistema peer-to-peer.

6. Redes P2P

Las redes P2P (Igual a Igual, del inglés Peer-to-Peer) empezaron a funcionar a partir de 1999 mediante una aplicación en la que 50 millones de usuarios intercambiaban canciones protegidas por derechos de autor sin el permiso de los propietarios de esos derechos. La idea básica de una red P2P es que muchos nodos se conecten entre sí y reserven sus recursos para formar un sistema de distribución de contenido. Estos nodos no tienen que ser grandes servidores con numerosos recursos, sino que normalmente la red está formada por los ordenadores domésticos de los clientes que forman parte de dicha red. Es por ello que cada nodo recibe el nombre de igual o par, ya que puede actuar tanto como cliente de otro igual y obtener información de este, como servidor para otro cliente ofreciéndole dicha información.

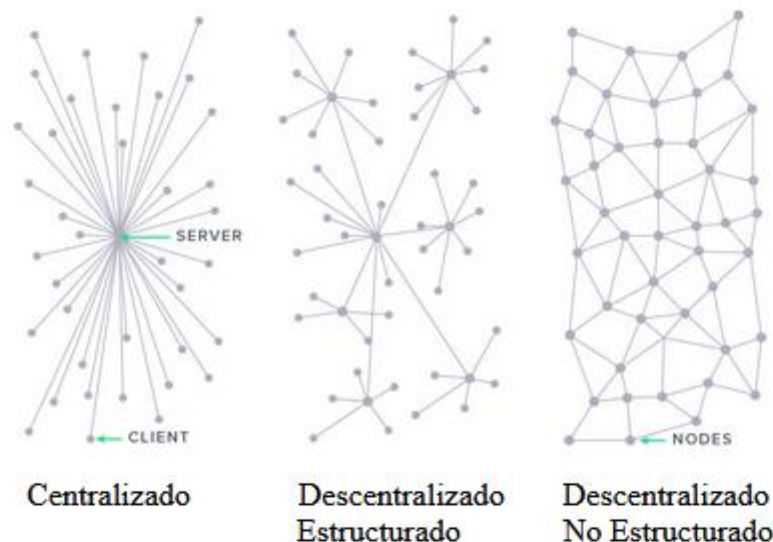
Las grandes ventajas que nos ofrecen usar este tipo de red para nuestras cadenas de bloques son el aumento exponencial de la seguridad de la información de nuestra cadena y la descentralización, al no tener que depender de ningún servidor en específico al que vayan dirigidas nuestras peticiones.

Dentro de las redes P2P nos podemos encontrar con distintos tipos:

- Redes descentralizadas y estructuradas: también las podemos encontrar con el nombre de P2P híbridas. En este tipo de redes no existe un único servidor centralizado que recibe las peticiones de los clientes. El mayor punto débil del sistema centralizado es la falta de alta disponibilidad debido a que si falla el nodo principal nadie podrá seguir recibiendo información. Es por ello que en las redes híbridas existen una serie de

nodos que tienen la capacidad de recibir las peticiones y dar la información evitando así tener un punto crítico en nuestro sistema. Estos nodos "servidores" pueden ser instalados en los ordenadores de los clientes. El número de nodos es directamente proporcional a la disponibilidad de la red.

- Redes descentralizadas y no estructuradas: en este sistema no contamos con los nodos "servidores" como en la red estructurada. Aquí todos los nodos son tratados como iguales por lo que cada nodo que forma parte de la red tiene las mismas funcionalidades que el resto. Esto nos otorgará la capacidad de disponer del mismo número de clientes como de servidores, permitiendo conexiones de cualquier a cualquier equipo y aumentado la velocidad de respuesta del sistema.



Antes de describir los tipos de redes P2P hemos comentado que los beneficios que nos otorgan esta estructura son dos: descentralización y seguridad. Vamos a explicar entonces las ventajas que nos aportan estas características.

La descentralización nos ofrece fundamentalmente los siguientes servicios:

- **Tolerancia a los fallos**: los sistemas descentralizados tienen menos probabilidades de fallar accidentalmente, estadísticamente es más probable que se caiga un único servidor que el fallo de cientos de nodos simultáneamente. Por lo tanto, la filosofía de este beneficio es simple a mayor número de nodos mayor tolerancia a fallos.

- **Resistencia a los ataques**: los sistemas descentralizados son más costosos de atacar y destruir o manipular porque carecen de un punto único que pueda ser atacado.
- **Resistencia a alianzas maliciosas**: es mucho más difícil que los participantes en los sistemas descentralizados se pongan de acuerdo para actuar de forma que les beneficie en contra de los otros participantes, mientras que en los sistemas centralizados es más probables que una persona o un grupo pequeño de personas, como la dirección de una empresa, se puedan corromper.

Respecto a la seguridad hemos comentado que gracias a las redes P2P nuestra cadena va a ser exponencialmente más segura, pero ¿por qué?

La seguridad de nuestra cadena no va a ser más segura por el hecho en sí de usar una arquitectura P2P, ya que por el contrario una de sus desventajas es la pérdida del anonimato de los usuarios que participan en ella a menos que la red esté diseñada para ello.

El potencial de la alianza de P2P con el blockchain es el poder distribuir nuestro libro mayor pudiendo así tener una "base de datos" distribuida descentralizada. Es por esto que las blockchain son un tipo de libro mayor inmutable, una vez que la información ha sido grabada es imposible revertirla.

Recuperando el ciberdelincuente del que hablamos en la sección anterior, supongamos que debido a que la información que quiere alterar es muy valiosa ha conseguido alterar los datos dentro del bloque y modificar los hashes de todos los bloques que seguían a este. Al introducir el sistema de nuestro libro mayor distribuido deberá crackear el 51% de los nodos en un tiempo inferior a los pocos minutos puedan pasar hasta que la información de los bloques que ha alterado en su nodo sea reconstruida. Esto es debido a que los nodos están constantemente revisando su información y comparándola entre sí, por lo tanto, al alterar la información los otros nodos comprobarán que no cuadra con la que tiene la mayoría y por consenso su información será restaurada.

Es por todo esto que la arquitectura Peer-to-peer es la gran aliada de la tecnología blockchain.

7. El Minado

La mayoría de personas relacionadas con el sector informático habrán podido leer estos últimos años los numerosos artículos a cerca de la falta de componentes, mayoritariamente tarjetas gráficas, debido a la “minería” de criptomonedas. Es por ello que en este apartado se va a intentar explicar que es y en que consiste el minar un bloque de la cadena.

Recapitulando lo que vimos en secciones anteriores, vimos que un bloque estaba formado por su número de bloque, los datos que incorporaba, el hash del bloque previo y el suyo propio que sería calculado a partir de los 3 parámetros anteriores.

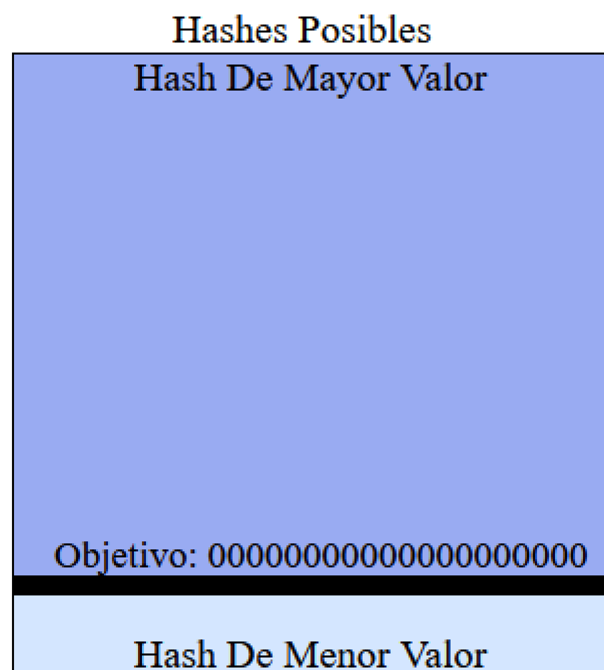
Bloque: #3
Transacciones: Usuario n otorga 200 coins a usuario y Usuario z otorga 50 coins a usuario q Usuario u otorga 1200 coins a usuario y Usuario n otorga 10 coins a usuario k
Hash Previo: B221D9DBB083A7F33428D7 C2A3C3198AE925614D70210 E28716CCAA7CD4DDB79
Hash Actual: D4C73897C0E37D7DF2DA06 473028222BA146A32CB68C5 54F8107DC8EF4508EB6

Pero como podemos apreciar el cálculo del hash es prácticamente inmediato, es por ello que debemos añadir un nuevo elemento denominado nonce. El nonce, cuyo nombre es proveniente de ‘number that can be only used once’ (número que solo puede usarse una vez), es simplemente un número. Un número aleatorio y de características únicas que tiene como finalidad ser usado en sistemas criptográficos. El descubrimiento de este número aleatorio es el causante de la minearía, debido a que como vimos una de las características de los algoritmos de hash es que sean irreversibles, por lo tanto, la única forma que hay de acertar el nonce es probando al azar mediante fuerza bruta. Por lo tanto, a mayor potencia de la instancia mayor será la posibilidad de descubrir el Golden nonce, así es como se denomina al nonce correcto, ya que podré hacer un mayor número de intentos en un menor tiempo.

Ahora, la pregunta que nos surge es ¿Cuál es el nonce válido? Es por esto que las cadenas de bloques marcan un objetivo para facilitar el descubrimiento de este número.

Como ya hemos comentado el hash es un identificador hexadecimal de 64 caracteres, pero este identificador no es más que un número el cual podemos pasar a cualquier base, es decir, este número tiene su equivalente en decimal. Son las cadenas de bloques las que imponen el objetivo que tiene que cumplir el nonce para que sea válido. Por lo tanto, para que el nonce sea válido el hash generado debe ser un número inferior al que marca el objetivo.

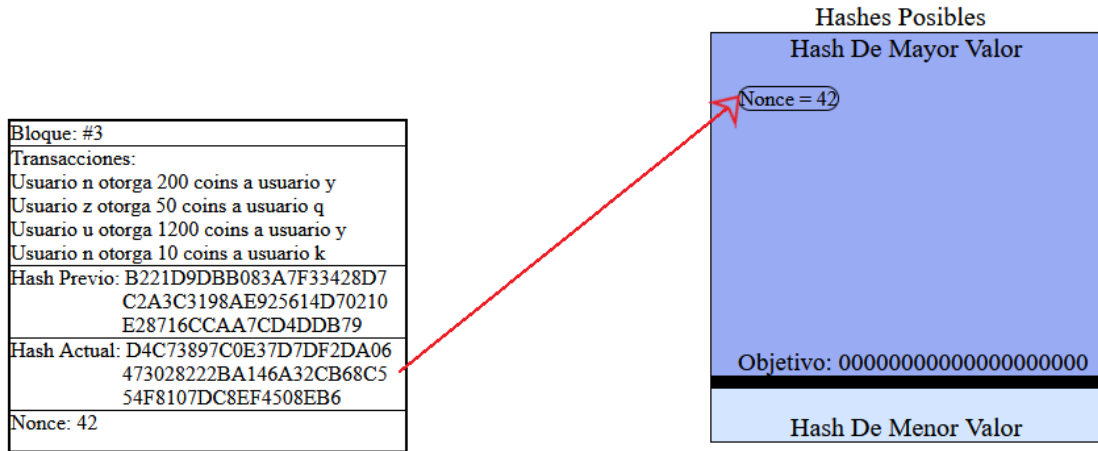
Imaginémonos que creamos una cadena de bloque e imponemos que para que un nonce sea válido el hash deberá empezar por 20 ceros a la izquierda. A mayor número de 0 más difícil será minar el bloque ya que las posibilidades de encontrar un nonce que de un hash válido se reducen.



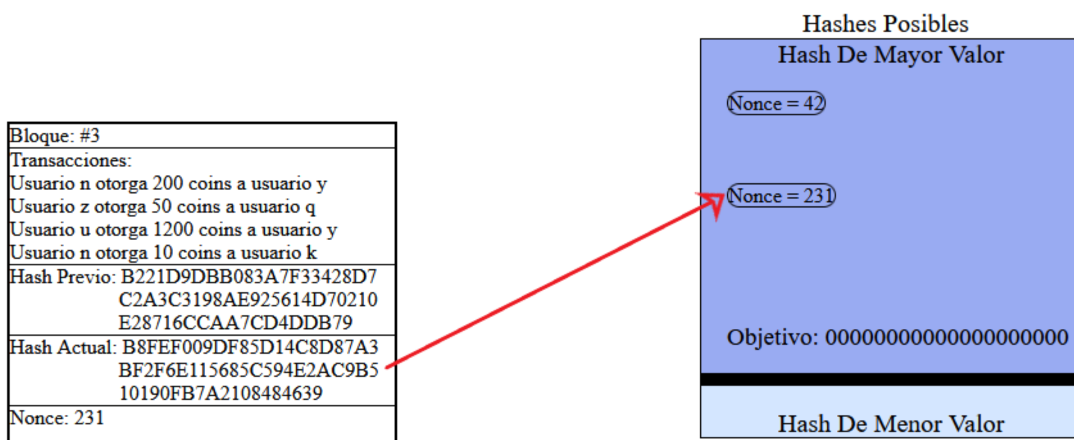
Ahora la función de los mineros es encontrar un nonce valido que cumpla con un hash que esté dentro del objetivo.

Por ejemplo, hemos creado un programa el cual va probando nonce hasta encontrar un hash que entre en nuestro objetivo. Si nos damos cuenta sólo podremos modificar el valor del nonce ya que el número del bloque es fijo, la información que hay dentro del bloque no se puede alterar ya que es la función de las cadenas de bloque y por último el hash previo tampoco puede ser modificado ya que es lo que enlaza a los bloques.

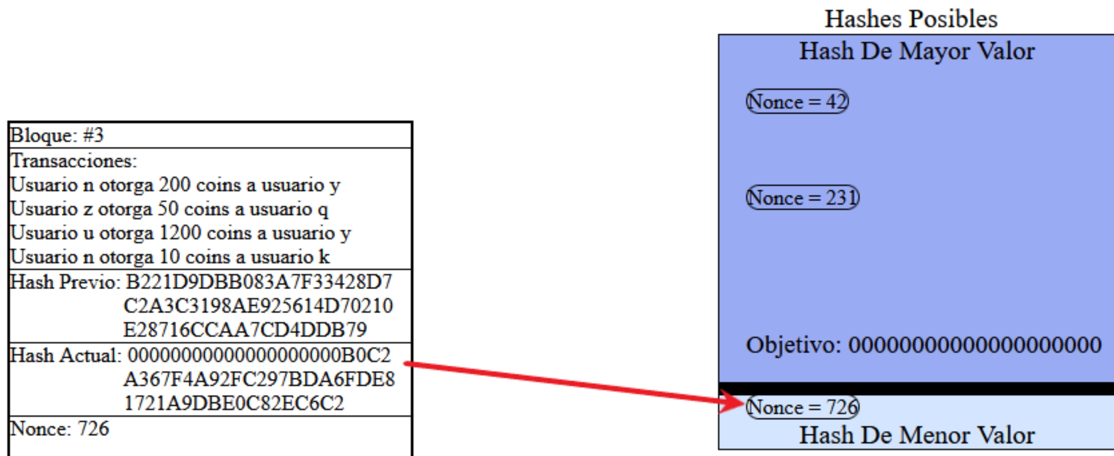
Intentamos con un número aleatorio y vemos que el hash que nos da es superior al objetivo, por lo tanto, ya sabemos que no es válido.



Volvemos a probar con otro número, como vemos, aunque el valor del nonce es superior obtenemos un hash más pequeño ya que no son proporcionales, aunque el valor del nonce sea muy grande no implica que el valor del hash también lo sea.



Probamos con otro nonce y vemos que este hash si encaja en nuestro objetivo ya que al introducir dicho nonce la combinación del número del bloque, el nonce, los datos y el hash previo produce un hash que empieza por veinte ceros, por lo tanto, hemos obtenido un hash que está dentro del objetivo. Ahora sí el bloque será añadido a la cadena.



8. Tolerancia a Fallos Bizantinos

En la mayoría de sistemas informáticos distribuidos, los nodos que forman la red deben de acordar periódicamente el estado de la cadena de bloque, esta acción es realizada mediante el consenso de la mayoría de los nodos. Sin embargo, lograr un consenso en las redes distribuidas no es una tarea fácil de realizar.

Debido a la cuestión de cómo pueden los nodos acordar una decisión mayoritaria, aunque algunos nodos fallen o existan nodos que actúen de manera malintencionada surge el denominado problema de los generales bizantinos.

El problema de los generales bizantinos fue concebido en 1982 por los matemáticos Leslie Lamport, Robert Shostak y Marshall Pease. El problema que nos plantea es la ilustración de un grupo de generales los cuales deben ponerse mayoritariamente de acuerdo en el próximo movimiento que van a realizar.

El dilema consiste en lo siguiente, cierto número de generales, cada cual, con su respectivo ejército, han sitiado una ciudad y deben ponerse mayoritariamente de acuerdo en la decisión que van a tomar. Deben decidir si atacar o elegir la retirada, en ambos casos si logran llegar un consenso saldrán con éxito.

Hay dos reglas fundamentales:

- Cada general tiene que decidir si desea atacar o retirarse.
- Después de que se toma la decisión, no se puede cambiar.

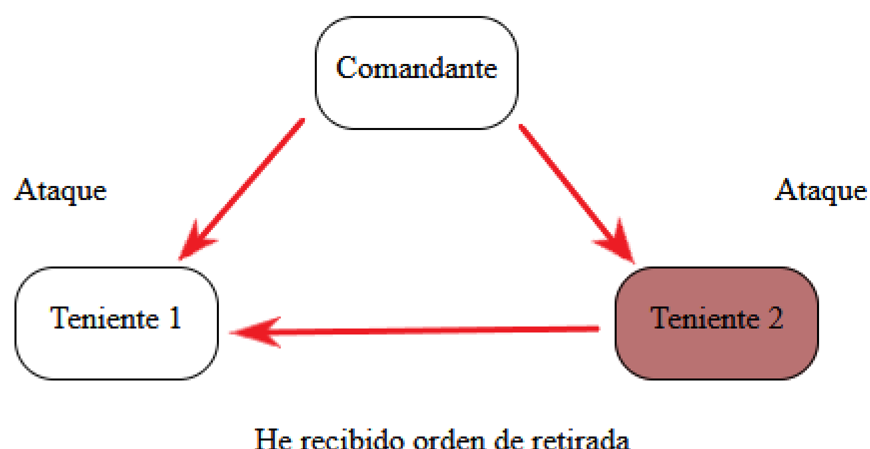
Sin embargo, el problema se complica al saber que pueden existir algunos generales traidores que van a tratar de impedir que los generales leales se pongan de acuerdo.

Es por ello que los generales deben realizar un algoritmo que garantice lo siguiente:

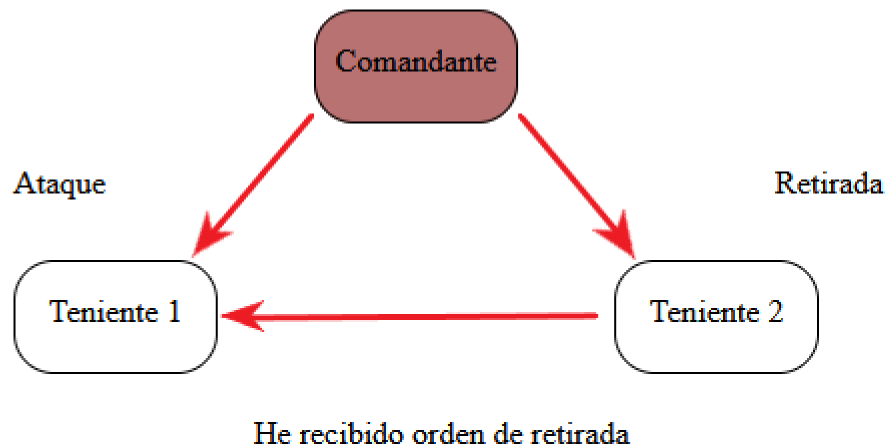
- Todos los generales leales deciden el mismo plan de acción. El algoritmo debe garantizar esta condición, independientemente de lo que hagan los traidores.
- Un pequeño grupo de traidores no puede hacer que los generales leales adopten un plan incorrecto. Esta condición es difícil de interpretar ya que, si los generales leales estuvieran divididos casi por igual entre las dos posibilidades, ninguna de las dos decisiones podría calificarse de mala.

Para que funcione el algoritmo el número de traidores deber ser estrictamente menor a 1/3 de los participantes. Esto es fácil de demostrar mediante los siguientes ejemplos mediante los cuales vemos que es imposible llegar a un consenso si entre los 3 tenientes existe un traidor. Los mensajes se transmitirán vía oral, por lo tanto, el contenido está completamente bajo el control del emisor, por lo que un emisor traidor puede transmitir cualquier mensaje posible. Un mensaje de este tipo corresponde al tipo de mensaje que los ordenadores normalmente se envían entre sí.

En la imagen 1 el comandante es leal y envía una orden de "ataque", pero el teniente 2 es un traidor e informa al teniente 1 de que ha recibido una orden de “retirada". El teniente 1 debería seguir la orden del comandante para que el plan funcionara, pero no tendría un consenso ya que habría recibido un voto de ataque y un voto de retirada.



En cambio, si el comandante es el traidor sería imposible llegar a la estrategia correcta a menos que alguno de los tenientes traicionara al comandante.



Si aplicamos el dilema al contexto de blockchains, cada general representa un nodo de red y los nodos deben alcanzar un consenso sobre el estado actual del sistema. Dicho de otra manera, la mayoría de los participantes dentro de una red distribuida tienen que estar de acuerdo y ejecutar la misma acción para evitar un fallo en la cadena.

Por lo tanto, la tolerancia a fallos bizantinos es la capacidad que tiene un sistema para seguir funcionando independientemente de que algunos nodos fallen o actúen a propósito para corromper el sistema. Este dilema no solo surge en blockchain, también lo podemos ver en el funcionamiento de los motores de un avión.

9. Protocolos de Consenso

Justo para evitar los fallos bizantinos, de los que hemos hablado en la sección anterior, surgen los protocolos de consenso los cuales tienen como objetivo mantener en un correcto estado a la cadena de bloques. Las funciones principales de estos algoritmos de consenso son dos: garantizar que el siguiente bloque de una cadena de bloques es la única versión correcta, y evitar que numerosos ciberdelincuentes descarrilen el sistema y bifurquen la cadena con éxito añadiendo bloques corrompidos.

Aunque el algoritmo de consenso más conocido hoy en día es la prueba de trabajo, vamos a ver que existen varios algoritmos más que pueden pasar a ser los principales en un futuro:

Prueba de trabajo

La filosofía en la que se basa la prueba de trabajo es en la de realizar un duro trabajo para poder resolver o minar un bloque, pero con un método sencillo y fácil de validar.

Como ya hemos comentado este algoritmo entrará en acción en el caso de que se quiera añadir un nuevo bloque a la cadena evitando que el bloque a añadir haya sido falsificado o que la cadena tenga distintas versiones de bloques válidos.

Para evitar el primero de los casos, un bloque malintencionado, lo que realiza el algoritmo es realizar una serie de pruebas para ver que el contenido del bloque no ha sido modificado. En el caso de que el bloque haya sido modificado el nodo responsable se quedará sin la recompensa y será expulsado de la red.

La otra opción que puede ocurrir es que más de un minero hayan conseguido obtener un nonce válido, pero al haber sido minado en un intervalo de tiempo muy corto no se ha podido avisar a todos los nodos de la red, por lo que ahora tenemos dos versiones de la cadena. Para resolver esto, el algoritmo deja ambos bloques en la cadena y espera al siguiente bloque, dependiendo del bloque que tenga el nuevo esa será la cadena válida y la otra pasará a ser descartada.

Por último, cada vez que un nodo distribuya un nuevo bloque lo único que deberán hacer los nodos de la red será verificar que si introducimos el nonce el hash que obtenemos está dentro del objetivo. Una vez que la mayoría de la red este de acuerdo en que el bloque es válido este será añadido a la cadena.

Prueba de participación

La alternativa más común a la prueba de trabajo es la prueba de participación.

En este tipo de algoritmo de consenso, en lugar de invertir en costosos equipos informáticos en una carrera por minar bloques, un "validador" invierte en las monedas del sistema. Esto es debido a que los validadores no necesitan usar cantidades significativas de poder computacional porque se seleccionan al azar y no compiten.

Una vez que el sistema ha elegido a un validador lo único que deberá de hacer es comprobar que el bloque que se va a añadir es correcto. En el caso de que de permiso a un bloque malicioso el validador será expulsado de la red.

Prueba de actividad

La prueba de actividad es un enfoque híbrido que combina la prueba de trabajo y la prueba de participación.

En la prueba de actividad, la minería se inicia de forma tradicional con la prueba de trabajo, en la que los mineros compiten para encontrar el Golden nonce. Dependiendo de la implementación, los bloques minados no contienen ninguna transacción (son más bien plantillas), por lo que el bloque ganador sólo contendrá una cabecera y la dirección de recompensa del minero.

En este punto, el sistema cambia a la prueba de participación. Basándose en la información de la cabecera, se elige un grupo aleatorio de validadores para firmar el nuevo bloque. Dependiendo de los parámetros que elija el sistema un validador, más probabilidades tendrá de ser elegido. El modelo se convierte en un bloque completo en cuanto todos los validadores lo firman.

Si algunos de los validadores seleccionados no están disponibles para completar el bloque, entonces se selecciona el siguiente bloque ganador, se elige un nuevo grupo de validadores, y así sucesivamente, hasta que un bloque recibe la cantidad correcta de firmas. Los honorarios se dividen entre el minero y los validadores que firmaron el bloque.

Prueba de quemado

Este algoritmo suele ser usado en criptomonedas. Es muy parecido a la prueba de participación, pero con una sutil diferencia, mientras que en la prueba de participación tiene más posibilidades de ser validador quien tiene más monedas, en este algoritmo tiene más posibilidades quien invierte o “quema” más criptodivisas. Las monedas que han sido quemadas se pierden, es decir, una vez que han sido invertidas no se recuperan.

Prueba de capacidad

Como hemos visto, la mayoría de estos protocolos alternativos emplean algún tipo de esquema de pago. La prueba de capacidad no es diferente, pero aquí se "paga" con espacio en el disco duro. Cuanto más espacio en el disco duro tengas, más posibilidades tendrás de minar el siguiente bloque y ganar la recompensa del bloque.

Antes de minar en un sistema de prueba de capacidad, el algoritmo genera grandes conjuntos de datos conocidos como "parcelas", que los mineros

almacenan en su disco duro. Cuantas más parcelas tenga, más posibilidades tendrá de encontrar el siguiente bloque de la cadena. Al invertir en terabytes de espacio en el disco duro, te compras una mayor posibilidad de crear bloques duplicados y bifurcar el sistema.

10. Creación de una Blockchain en Python

En este apartado vamos a ver cómo podemos crear una cadena de bloques en Python. Para ello, usaremos un framework de dicho lenguaje denominado Flask.

Lo primero que debemos de hacer es instalar las librerías que vamos a usar para nuestro programa.

```
from flask import Flask
import hashlib
import json
from datetime import datetime
import sys
import random
import math
```

Ahora, lo primero que vamos a hacer es crear la clase cadena_bloque, el cual en el constructor iniciaremos una lista vacía donde guardaremos los bloques que se vayan creando. En la programación orientada a objeto el constructor es el método que es llamado cada vez que creamos una nueva instancia de la clase. En Python se crea con el método __init__.

```
class cadena_bloque:
    def __init__(self):
        self.cadena = []
```

Esta clase va a tener una serie de métodos de instancia, los cuales siempre toman el parámetro self como primer parámetro. El parámetro self representa la instancia del método.

El primer método se encargará de crear los bloques e introducirlos en nuestra cadena, la lista creada anteriormente. Los bloques serán diccionarios, que como hemos visto en este documento, tendrán el número del bloque, un identificador tipo timestamp, el cual nos indicará en segundos cuando fue creado, el nonce y el hash del bloque anterior y su propio hash.

El bloque génesis deberemos de tratarlo de forma especial al ser el primer bloque.

```
    def crear_bloque(self, nonce, hash_previo,hash=0):
            posicion = len(self.cadena)
            id_tiempo = str(datetime.today().timestamp())
            if posicion == 0:
                bloque_genesis = {'indice' : posicion,
                        'timestamp' : id_tiempo,
                        'nonce' : nonce,
                        'hash_previo' : hash_previo}
                hash = self.generar_hash(bloque_genesis)
                bloque = {'indice' : posicion,
                        'timestamp' : id_tiempo,
                        'nonce' : nonce,
                        'hash_previo' : hash_previo,
                        'hash' : hash}
                self.cadena.append(bloque)
                return bloque
            else:
                bloque = {'indice' : posicion,
                        'timestamp' : id_tiempo,
                        'nonce' : nonce,
                        'hash' : hash_previo,
                        'hash_bloque_anterior': hash}
                self.cadena.append(bloque)
                return bloque
```

Este método será llamado cada vez que creemos una instancia de tipo cadena_bloque, por lo tanto, debemos añadirlo al constructor de la clase. Le tendremos que pasar como parámetros el valor del nonce, el cual puede ser un número aleatorio, y el valor del hash del bloque previo, este valor será 0 debido a que es el bloque génesis.

```
class cadena_bloque:
    def __init__(self):
        self.cadena = []
        self.genesis=self.crear_bloque(0,'0')
```

Añadimos un nuevo método a nuestra clase que se va a encargar de obtener el valor del último bloque de la cadena.

Crearemos un método igual que el anterior que nos devolverá el nonce del bloque previo.

```
    def obtener_bloque_previo(self):
        ultimo_bloque = self.cadena[-1]
        return ultimo_bloque

    def obtener_nonce_previo(self):
        ultimo_bloque = self.cadena[-1]
        nonce_previo = ultimo_bloque['nonce']
        return nonce_previo
```

El siguiente método se va a denominar prueba_de_trabajo y va a ser el encargado de ir probando nuevos nonce hasta dar con el correcto. Este método va a recibir como parámetro el nonce anterior, a continuación, veremos para que lo necesitamos.

Lo primero que vamos a hacer es inicializar 2 variables, nuevo_nonce, el cual será un número entero positivo entre todos los números disponibles del sistema, y nonce_correcto, un booleano que iniciaremos en False y nos informará si el nuevo_nonce es correcto, en ese caso pasará a True.

Iniciamos un bucle while que estará activo mientras nonce_correcto sea False, es decir, parará cuando encontremos un nonce correcto. Dentro de este bucle creamos nuestra operación matemática la cual no puede ser simétrica y debe involucrar nonce_nuevo con el nonce_previo, parámetro de entrada del método. En mi caso he elegido como operación el algoritmo de la resta de las raíces cuadradas de los nonces.

Nuestra operación será pasada a la función de hash sha256 y convertida a hexadecimal. Por último, marcamos nuestro objetivo de hashes válidos, en este caso he elegido que los hashes deben iniciar por lo menos en 5 ceros. Si quisiéramos dificultar el algoritmo incrementaríamos el número de 0.

Si el hash está en el objetivo la variable nonce_correcto pasará a True y saldremos del bucle, en caso contraría probaremos con otro número aleatoriamente.

```
    def prueba_de_trabajo(self,nonce_previo):
        nuevo_nonce = random.randint(0,sys.maxsize)
        nonce_correcto = False
        while(not nonce_correcto):
            operacion = str(math.log((math.sqrt(nuevo_nonce)-math.sqrt(nonce_previo)))).encode()
            hash_generado = hashlib.sha256(operacion).hexdigest()
            if hash_generado[0:5] == '00000':
                nonce_correcto = True
            else:
                nuevo_nonce = random.randint(0,sys.maxsize)

        return nuevo_nonce
```

El siguiente método va a ser el encargado de generar el hash de un bloque. Recibirá como parámetro de entrada el propio bloque. Dentro de esta función convertiremos el bloque, de tipo diccionario, a un string conforme al estándar json. A continuación, generaremos el hash con la librería correspondiente.

Debemos tener en cuenta que en Python los diccionarios no tienen un orden, esto podría provocar un intercambio de la posición de los datos generando hashes diferentes para un mismo bloque, es por ello que ordenaremos los diccionarios por el valor de las claves.

```
    def generar_hash(self, bloque):
        bloque_json = json.dumps(bloque, sort_keys=True).encode()
        hash = hashlib.sha256(bloque_json).hexdigest()
        return hash
```

Por último, vamos a añadir un método que va a comprobar la validez de la cadena. Esta función devolverá verdadero si todo está correcto o falso si ha encontrado algún fallo. Para ello, recorrerá toda la lista de bloques y comprobará que el hash previo de los bloques coincide con el hash del bloque anterior y si el hash de los bloques cumple el objetivo del número de ceros por el que debe empezar.

```
    def validar_cadena(self, cadena):
        indice_posicion_previa = 0
        for bloque in cadena[1:]:
            if bloque['hash_previo'] != self.generar_hash(cadena[indice_posicion_previa]):
                return False

            nonce_previo = cadena[indice_posicion_previa]['nonce']
            nonce_actual = bloque['nonce']
            operacion =str(math.log((math.sqrt(nonce_actual)-math.sqrt(nonce_previo)))).encode()
            hash_generado = hashlib.sha256(operacion).hexdigest()
            if hash_generado[0:5] != '00000':
                return False

            indice_posicion_previa = indice_posicion_previa + 1

        return True
```

Ahora sólo nos queda desarrollar la aplicación web con Flask.

```
app = Flask(__name__)
cadena = cadena_bloque()
@app.route("/minado",  methods=["POST", "GET"])
def minado():
    if request.method == 'POST':
        nonce_previo = cadena.obtener_nonce_previo()
        nonce = cadena.prueba_de_trabajo(nonce_previo)
        bloque_previo = cadena.obtener_bloque_previo()
        hash_previo = cadena.generar_hash(bloque_previo)
        cadena.crear_bloque(nonce, hash_previo, bloque_previo['hash'])
        cadena.transacciones.append(request.form['text'])
        return render_template('index.html', cadena=cadena.cadena,
transacciones = cadena.transacciones)
    else:
        return render_template('index.html', cadena=cadena.cadena,
transacciones = cadena.transacciones)
```

Tras aplicar las correspondientes hojas de estilos y la creación del html este sería el resultado. Tras introducir las transacciones se minarán los bloques y quedarán inmutables.

En el caso de que el lector quisiera probar la aplicación podrá conseguirla con esta imagen Docker: jesusrubiomartin/blockchain

Ejecutando el siguiente comando tendrá la aplicación en el puerto 80 de su máquina anfitriona:

```
docker run -d -p 80:5000 jesusrubiomartin/blockchain
```

11. El Rango del Nonce

Como ya hemos comentado anteriormente el nonce es el único campo que los mineros pueden ir alterando para dar con un hash que se encuentre dentro del objetivo. Ahora la pregunta que nos surge es: ¿cuál es el valor máximo que puede tomar el nonce?

El valor del nonce no es infinito, es un número de 32 bits sin signo. Esto nos da lugar a que el número mínimo que puede tomar el nonce es 0 y el máximo es 4294967295, algo más de cuatro mil millones.

Ahora vamos a ver como de grande es este número comparando con la potencia que tienen los dispositivos encargados de encontrar este número.

Para hacer estos cálculos vamos a utilizar como ejemplo datos sobre el protocolo bitcoin ya que es uno de los más conocidos hoy en día:

- Cantidad de números hexadecimales de 64 caracteres: 16^{64}
  Si cada dígito puede tener 16 caracteres distintos del 0 a F deberemos multiplicar 16 por los 64 caracteres dando como resultado aproximadamente 10^{77}
- Cantidad de hashes dentro del objetivo: 16^{64-19}
  Actualmente para que un bloque sea válido en bitcoin su hash debe empezar como mínimo por 19 ceros a la izquierda. Es por ello, que las posibilidades que tenemos ahora se ven reducidas drásticamente pues esos 19 caracteres son fijos y sólo podemos jugar con los 45 restantes. Esta cantidad es aproximadamente 10^{54}
- Probabilidad de encontrar un hash válido: 10^{-23}
  Por lo tanto, la probabilidad de encontrar un hash valido será el cociente de la cantidad de hashes válidos entre la cantidad de hashes disponibles. Una probabilidad del 0.00000000000000000000001%.

Como ya hemos comentado disponemos aproximadamente de $4x10^{9}$ nonces. Si asumimos que no hay colisiones, que un nonce equivale al cálculo de un único hash, dispondríamos de $4x10^{9}$ hashes. Con estos datos podríamos calcular la probabilidad de que un nonce sea el correcto:

- Probabilidad de que un nonce sea válido: $4x10^{-14}$
  Dicho número sale de la probabilidad de que un hash disponible a partir de la cantidad de nonce que tenemos multiplicado por la probabilidad de que un hash sea válido, lo cual nos da una probabilidad del 0.000000000000004%.

Lo que podemos concluir de la última probabilidad que hemos calculado es lo siguiente: si recorremos los cuatro mil millones de nonce que disponemos, la probabilidad de que un nonce dé con un hash correcto es aproximadamente $4x10^{-14}$. Como ya hemos visto es una probabilidad tremendamente baja. Esto podría provocar que recorriéramos todos los nonce disponibles y no diéramos con ningún hash válido.

Otra pregunta que nos puede surgir es respecto a la velocidad de computación de la que disponemos actualmente. Podemos poner un ejemplo de la velocidad a la que probaría nonce un minero con un equipo muy básico de dos gráficas Nvidia RTX 3060 las cuales son capaces de calcular 50 millones de hashes por segundo, es decir, el equipo de este minero podría recorrer los cuatro mil millones de hashes en tan sólo 40 segundos. Entonces supongamos que este minero intenta minar un bloque, recorre todos los nonce generando todos los hashes disponibles y se encuentra con que ningún hash está dentro del objetivo. Por esta razón deberemos añadir un nuevo campo a nuestro bloque: el timestamp.

El timestamp es la fecha (hora, minuto, segundo, día, mes y año) en el que el bloque está siendo minado. Y este campo se irá actualizando cada segundo que pasa cogiendo como estándar el unix timestamp, el cual cuenta cada segundo desde 1970.

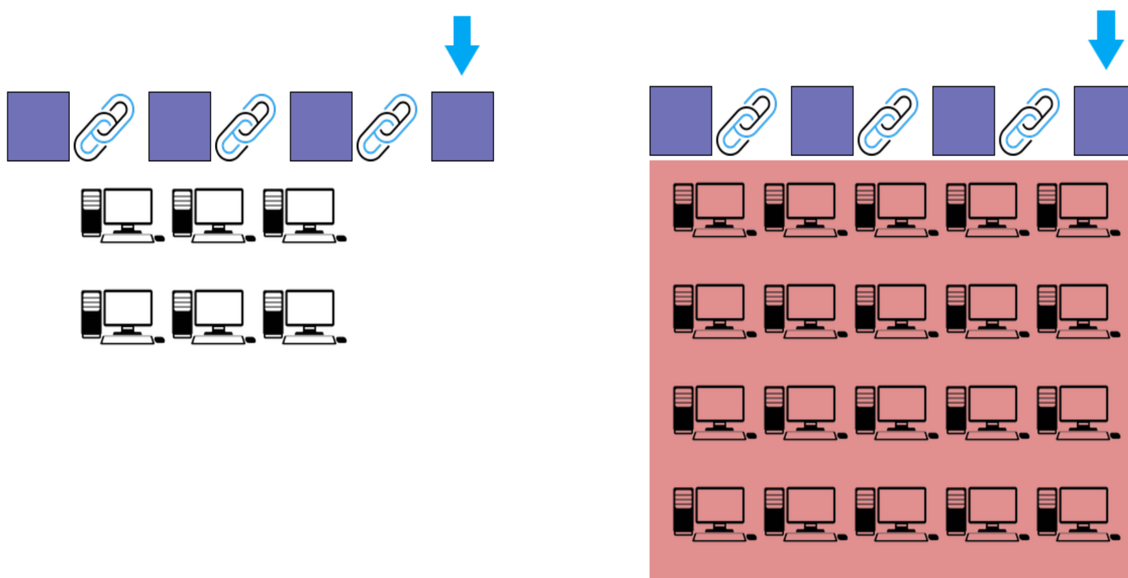
El cambio del timestamp cada segundo modificará el problema constantemente ya que como hemos visto a lo largo de este documento, un mínimo cambio en la información del bloque modificaría el valor del hash.

Introduciendo este nuevo campo, que se actualizará cada segundo, hemos solucionado el problema de no tener ningún nonce válido. Ahora el minero podrá probar el mayor número de nonce posibles en cada segundo hasta encontrar un hash que se encuentre dentro del objetivo.

12. Ataque del 51%

Como ya sabemos ningún sistema ni tecnología es 100% segura y blockchain no es una excepción. En este apartado vamos a explicar una de las más famosas vulnerabilidades que presentan las cadenas de bloques. El ataque del 51% es una vulnerabilidad teórica debido a que hoy día no se ha logrado realizar con éxito.

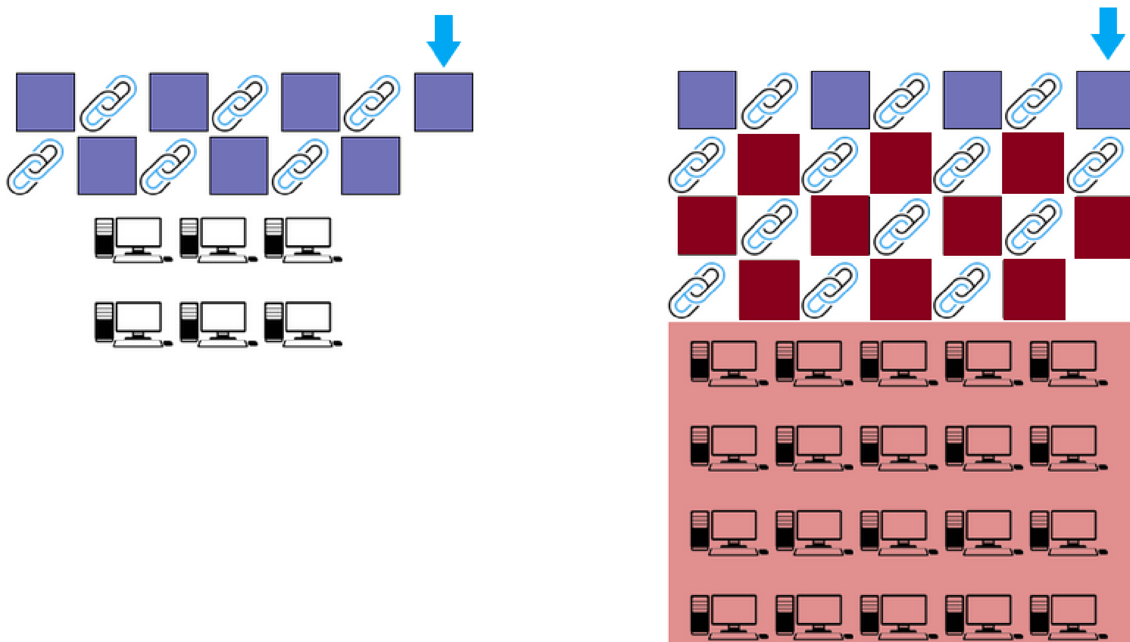
Imaginemos que tenemos en nuestra red dos grupos de mineros: legítimos y maliciosos. Para que este ataque sea posible el grupo malicioso deberá ser estrictamente mayor que el grupo de mineros legítimos. Nuestros mineros legítimos se encontraban minando y formando nuestra cadena de manera segura, pero de repente los mineros maliciosos se unen a nuestra red.



Lógicamente nuestro software no tiene ningún sistema para detectar si uno de los nodos que se conectan a la red tiene la intención de modificar nuestra cadena maliciosamente por lo tanto lo que hará este grupo de nodos mayoritarios será hacer una copia de la cadena en el punto actual. Una vez que cada uno de los nodos tiene una copia llega uno de los pasos importantes, todos y cada uno de los mineros maliciosos se desconectan de nuestra red.

A partir de ahora, los mineros legítimos seguirán aportando bloques a nuestra cadena, pero los mineros maliciosos también harán aumentar nuestra cadena a partir de la copia que realizaron, esta cadena maliciosa irá aumentando, pero no seremos conscientes debido a que como hemos comentado se desconectaron de nuestra red.

Es por esto la importancia de que el número de maliciosos tiene que ser mayor al número de mineros que actúan de buena fe, ahora la cadena maliciosa aumentará en un menor tiempo al tener una mayor potencia computacional que el grupo de nodos legítimos. Puede ser que en el tiempo que nuestra cadena mina un bloque la cadena maliciosa mine cuatro bloques más.



Llegados a este punto los mineros maliciosos volverán a conectarse a nuestra red llevándose a cabo la corrupción de la cadena. Ahora el sistema se encuentra con dos cadenas en competición: nuestra cadena y la maliciosa.

El sistema al encontrarse con dos cadenas seguirá una de las reglas principales, es decir, debe continuar la cadena más larga y la otra debe desapareces. En este punto nuestro sistema habrá caído.

Por lo tanto, el punto débil de este ataque es el poder computacional con el que cuenta una cadena de bloque. A mayor fuerza de minado disponga una cadena más difícil será conseguir el 51% de los nodos. Por suerte, hoy en día este ataque no ha tenido éxito en ninguno de los protocolos importantes.

13. Claves Públicas y Privadas

Como ya hemos comentado, blockchain y criptografía son tecnologías que van prácticamente juntas. Es por ello que vamos a explicar en este punto que son las claves públicas y privadas, ya que son términos que podemos encontrar a menudo en el mundo del blockchain.

El fin por el que surge la criptografía de clave pública (PKC) o cifrado asimétrico es simple, facilitar una comunicación segura y privada mediante firmas digitales en un canal público donde pueden existir vulnerabilidades que pueden aprovechar los atacantes. Por lo tanto, ya tenemos un método para saber que una transacción ha sido realizada por quién dice ser quién la ha realizado.

En el cifrado asimétrico cada participante posee una especie de "llavero" con dos claves: la clave privada (no puede ser mostrada a nadie) y la clave pública.

Existen dos variantes dentro de la criptografía de clave pública dependiendo de con que clave cifremos el mensaje:

- <u>Cifrado de clave pública</u>: en este caso el mensaje será cifrado con la clave pública de la persona a la que queremos enviar nuestra forma. De esta forma sólo él podría descifrar nuestro mensaje. Una buena analogía sería la siguiente:
  Podemos tratar a la clave pública del receptor como un candado y su clave privada como una llave. De esta forma nosotros pondremos el candado (clave pública) a nuestro mensaje y sólo el receptor podrá abrirlo con su llave (clave privada).
- <u>Firmas digitales</u>: en el caso de las firmas digitales el mensaje será cifrado con nuestra clave privada para garantizar a los receptores, los cuales tienen nuestra clave pública, que hemos sido nosotros quién ha escrito ese mensaje y nadie ha suplantado nuestra identidad.

Entre las ventajas fundamentales de la criptografía asimétrica podemos encontrar:

- No repudio, podemos demostrar o probar la participación de las partes (origen y destino, emisor y receptor, remitente y destinatario), mediante su identificación.
- Poder enviar mensajes a través de canales abiertos sin que nadie sepa su contenido.
- Dispone de una alta confidencialidad.
- Alta seguridad.

Mientras, su principal desventaja es su velocidad y su costo computacional.

14. Smart Contracts

Los contratos inteligentes son simplemente programas almacenados en una cadena de bloques que se ejecutan cuando se cumplen unas condiciones predeterminadas. El código de estos programas es visible por cualquier participante. Suelen utilizarse para automatizar la ejecución de un acuerdo, de modo que todos los participantes puedan estar inmediatamente seguros del resultado, sin que intervenga ningún intermediario ni se pierda tiempo.

El nombre de contrato proviene del significado de dicha palabra ya que un contrato es un acuerdo en el que se definen una serie de reglas las cuales no pueden ser incumplidas.

En el caso de usar smarts contracts en nuestra cadena de bloque, se añadirán dos parámetros de información más, por ende, al estar en los bloques también se almacenarán dentro de todos los nodos de nuestra red.

En primer lugar, almacenaremos el historial de todos los smarts contracts, es decir, si había contratos inteligentes en bloques anteriores también se encontrarán en todos los nodos. Por lo tanto, todos los contratos inteligentes de nuestra cadena se encontrarán corriendo en todas las instancias.

En segundo lugar, también se almacenarán los estados actuales de todos los smarts contracts. Al final un Smart Contract no es más que un simple programa escrito en un lenguaje de programación por lo tanto puede estar en diversos estados: fallido, ejecutándose, con éxito, …

Entre las ventajas de los smarts contracts podemos encontrar:

- **Velocidad y precisión**: una vez que se cumple una condición, el contrato se ejecuta inmediatamente. Además, una vez que hemos creado nuestro Smart Contract eliminaremos el factor humano.
- **Confianza y transparencia**: en los contratos inteligentes todas las transacciones quedan registradas en la cadena de bloques por lo tanto no hace falta manejar dicha información por alguna empresa o persona.
- **Seguridad**: como ya hemos comentado a lo largo de este documento las cadenas de bloques son muy difíciles de corromper por lo tanto los smarts contracts también.
- **Ahorro**: los smarts contracts conllevan abaratamiento de costes debido a la eliminación de los intermediaros que cobrarían tarifas como en tantos otros sectores.

15. Smart Contract en Solidity

En este punto vamos a explicar paso por paso la realización de un Smart Contract en el lenguaje solidity, el cual es un lenguaje de programación de alto nivel diseñado para este fin. Este contrato inteligente será el encargado de manejar una empresa de renting car. Veremos que una vez programado el contrato la empresa podrá organizarse prácticamente sola sin necesidad de interactuar con personas. Todo podría ser automático, incluso aunque la empresa quebrara el contrato inteligente podría seguir funcionando.

En primer lugar, en solidity debemos declarar la versión del compilador con la que vamos a trabajar. Podemos añadir un rango de versiones como vamos a hacer en nuestro caso. Además, añadiremos la librería ABIEncoderV2, la cual nos ayudará a crear hashes del conjunto de información que introduzcamos como parámetros.

```
pragma solidity >=0.4.4 < 0.7.0;
pragma experimental ABIEncoderV2;
```

Ahora sí, ya podemos crear nuestro contrato. En solidity un contrato es el bloque de construcción más básico. Todas las funcionalidades que queremos implementar deberán estar dentro de este bloque. Para crearlo usaremos la palabra reservada contract seguido del nombre que queramos darle.

```
contract renting {

}
```

Ahora declararemos una variable que guardará la dirección, los últimos 20 bytes del hash que devolverá la función Keccak-256 de la clave pública, del director de la empresa de renting. La declaramos, es un tipo de variable address y también la haremos pública para que pueda ser accesible desde fuera del contrato.

```
contract renting {

    address public director;

}
```

Una vez hemos declarado la variable director debemos crear el constructor. En solidity dentro del constructor debemos especificar las propiedades que definen el contrato. Constructor va a inicializar las variables de estado del contrato, estas variables se almacenarán en una parte del contrato conocida como storage. En nuestro caso la variable director será igual a la dirección de la persona que haya desplegado nuestro contrato en la cadena de bloques. Para ello, usaremos la función global msg.sender.

```
contract renting {


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

A continuación, vamos a relacionar el hash del dni de cada cliente con una lista de string en la que se encontrarán la matrícula del vehículo y que fecha debe devolverlo. Para ello usaremos la función maping.

```
    mapping(bytes32 => string[]) user_car;
```

También crearemos un array con los clientes que pidan hacer una revisión del coche alquilado en un taller.

```
  string [] reclamacion_taller;
```

Podemos ya crear nuestra primera función que se denominará alquiler y tendrá como parámetros de entrada el dni del cliente junto con un array donde en primer lugar introduciremos la matrícula del vehículo y en segunda posición la fecha en la que se deberá devolver el vehículo. Esta función lógicamente sólo podrá ser usada por el director de la empresa por ello añadiremos un modificador que nos permitirá esta opción.

```
function Alquiler(string memory dnicliente, string[] memory 
cliente_fecha) public exclusivo_director(msg.sender){
        bytes32 hash_dni = keccak256(abi.encodePacked(dnicliente));
        user_car[hash_dni] = cliente_fecha;
    }
```

Dentro de nuestra función hashearemos el dni de nuestro cliente, de esta manera no estará expuesto en las transacciones de nuestra cadena de bloque, y usando el map creado anteriormente lo relacionaremos con la lista de matrícula y fecha.

A continuación, creamos el modificador exclusivo_director en el que comprobaremos si la dirección que ha lanzado el contrato es igual a la que quiere ejecutar la función.

```
modifier exclusivo_director(address direccion){
        require(direccion == director, "Permisos insuficientes.");
        _;
    }
```

Otra función que podríamos configurar para nuestros clientes sería ver qué día deben entregar el coche. Esto es tan sólo es un ejemplo ya que en un caso real podríamos añadir muchísimos más datos ahorrando facturas o papeles innecesarios los cuales pueden perderse. En el caso de nuestro Smart contract el cliente sólo necesitaría su dni.

```
function dia_entrega(string memory dnicliente) public view returns(string
 memory) {
        bytes32 hash_dni = keccak256(abi.encodePacked(dnicliente));
        string [] memory fecha = user_car[hash_dni];
        return fecha[1];
    }
```

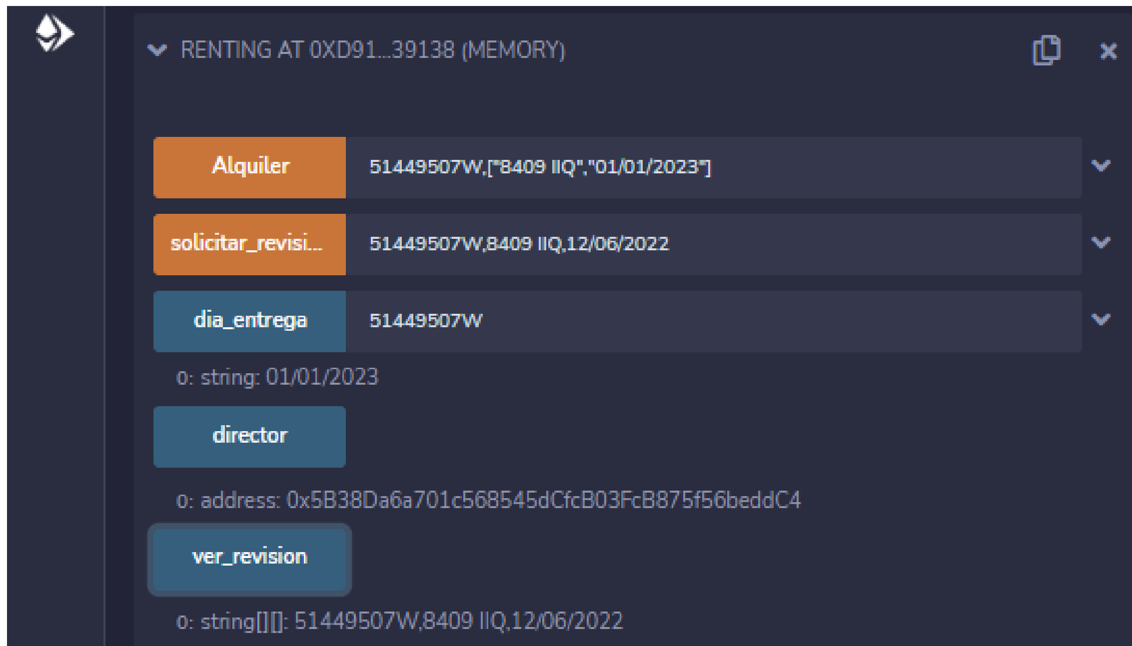
Añadimos una función más para que los clientes puedan solicitar citas para revisiones en el taller y otra función para que la empresa pueda ver las solicitudes de revisiones.

```
    string[][] revisiones;

    function solicitar_revision(string memory dnicliente, string memory
matricula, string memory fecha) public {
        revisiones.push([dnicliente,matricula,fecha]);
    }

    function ver_revision() public view exclusivo_director(msg.sender)
returns (string [][] memory){
        return revisiones;
    }
```

Por lo tanto, nuestro Smart Contract quedaría de la siguiente forma:

```
pragma solidity >=0.4.4 < 0.7.0;
pragma experimental ABIEncoderV2;

contract renting {

    address public director;
    constructor () public{
        director = msg.sender;
    }

    mapping(bytes32 => string[]) user_car;
    string [] reclamacion_taller;

    function Alquiler(string memory dnicliente, string[] memory
cliente_fecha) public exclusivo_director(msg.sender){
        bytes32 hash_dni = keccak256(abi.encodePacked(dnicliente));
        user_car[hash_dni] = cliente_fecha;
    }

    modifier exclusivo_director(address direccion){
        require(direccion == director, "Permisos insuficientes.");
        _;
    }

    function dia_entrega(string memory dnicliente) public view
returns(string memory) {
        bytes32 hash_dni = keccak256(abi.encodePacked(dnicliente));
        string [] memory fecha = user_car[hash_dni];
        return fecha[1];
    }

    string[][] revisiones;

    function solicitar_revision(string memory dnicliente, string memory
matricula, string memory fecha) public {
        revisiones.push([dnicliente,matricula,fecha]);
    }

    function ver_revision() public view exclusivo_director(msg.sender)
returns (string [][] memory){
        return revisiones;
    }

}
```

Para probarlo podemos desplegarlo en remix, esta página nos permitirá probar nuestro código tan solo desde el navegador web.



Ahora todos nuestros datos estarían almacenados en forma de transacciones en nuestra blockchain, por lo tanto, serían inmutables y quedarían reflejados para siempre.

16. Conclusión

En definitiva, la tecnología blockchain es uno de los sectores con más futuros debido a su seguridad y descentralización. Dentro de 10 o 20 años esta tecnología estará implementada en prácticamente todos los ámbitos de nuestra vida, desde votaciones electorales hasta registro de viviendas.

Además, la evolución de los smarts contracts nos darán la posibilidad de que empresas funcionen de manera descentralizada y sin prácticamente intervención humana. También llegaremos a ver empresas en las que se combinen la inteligencia artificial junto con los smarts contracts llegando a obtener por ejemplo empresas de transportes en las que no necesitaremos ni conductores. A base de los contratos inteligentes los vehículos sabrán cuando parar a repostar, que ruta tiene menos tráficos, la parada con más usuarios en espera, …

Es por todo esto que las cadenas de bloques es uno de los campos con mayor área de desarrollo y que revolucionará el mundo tal y como hoy lo conocemos.