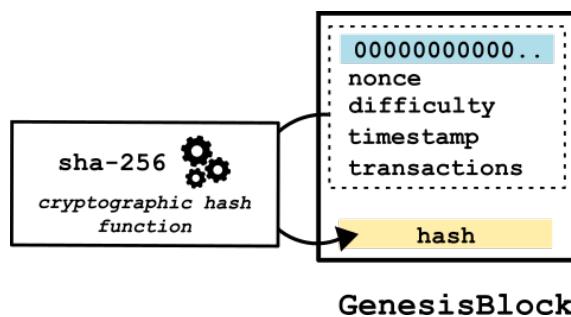


Desarrollo de una criptomoneda con Blockchain

Sara Pérez García

Formacion Profesional en Desarrollo de Aplicaciones Multiplataforma
IES Islas Filipinas



Junio 2022

Tutora:
María Isabel Márquez Carpio

Índice

1. Introducción	2
1.1. Descripción del problema	2
1.1.1. ¿Qué es Blockchain?	2
1.1.2. ¿Qué es una criptomoneda?	2
1.2. Objetivos	2
1.3. Motivación	3
2. Tecnologías y Herramientas utilizadas	3
3. Estimación de Recursos y Planificación	4
4. Análisis	4
5. Desarrollo de la Cadena de Bloques	6
5.1. Componentes del Blockchain	6
5.2. Minado de Bloques	8
5.3. Red descentralizada y sincronización de la cadena de bloques	10
6. Desarrollo de la criptomoneda	11
6.1. Transacciones	11
6.2. Wallet y firmas digitales	12
6.3. Contenedor de transacciones y minado	12
6.4. Implementación de la criptomoneda	13
7. Conclusiones	14
8. Anexo	16
8.1. Informe de Tests	16

Índice de figuras

1. Características del Blockchain más relevantes a diferentes aplicaciones industriales.	3
2. Planificación - Diagrama de Gantt	4
3. Caso de Uso: Crear una transacción y añadirla al contenedor de transacciones	5
4. Caso de Uso: Minar el contenedor de transacciones en la cadena de bloques.	5
5. Diagrama de clases	6
6. Componentes de la cadena de bloques	7
7. Flujo requerido para la correcta resolución del PoW durante el minado de bloques en este proyecto.	9
8. Red P2P y la importancia de la sincronización de los peers evitando posibles ataques a la cadena de bloques	11
9. Cartera virtual que contiene la clave privada y publica requeridas para poder crear y recibir transacciones.	12
10. Bloques minados en la cadena de bloques de la criptomoneda Bitcoin a día 3 de Junio de 2022	13
11. Recompensa obtenida por minar el bloque 739039 del blockchain en el que se sustenta la criptomoneda Bitcoin.	13

Desarrollo de una criptomoneda con Blockchain

Sara Pérez García

Resumen

REHACERBlockchain es una palabra que escuchada en nuestro día a día cada vez más. Con este proyecto se ha querido aprender a utilizar esta tecnología con aplicaciones reales. Me he centrado en la implementación de una criptomoneda con Node.js, Express, PubNub, React y Git. REHACER

1. Introducción

1.1. Descripción del problema

1.1.1. ¿Qué es Blockchain?

Blockchain (cadena de bloques en español) es un conjunto de tecnologías que permiten llevar un registro seguro, descentralizado, sincronizado y distribuido de las operaciones digitales, sin necesidad de la intermediación de terceros [1]. Blockchain es un libro mayor compartido e inmutable que facilita el proceso de registro de transacciones y de seguimiento de activos en una red de negocios. Un activo puede ser tangible (una casa, un auto, dinero en efectivo, terrenos) o intangible (propiedad intelectual, patentes, derechos de autor, marcas). Prácticamente cualquier cosa de valor puede ser rastreada y comercializada en una red de blockchain, reduciendo el riesgo y los costos para todos los involucrados [2].

Con el Blockchain se consigue tener un control de acceso sobre los datos almacenados, seguridad sobre los datos de que no van a ser modificados en posibles ataques; transparencia, ya que todo el mundo puede acceder directamente a la cadena de bloques y consultar la información que contenga. Se almacena un historial de las transacciones lo que mantiene la trazabilidad de estas. Dado que se encuentra distribuida no existen intermediarios y nadie puede ser dueño de la información que esta cadena contiene. Esto genera una reducción de costes, más eficiencia y que cada uno pueda ser dueño de sus datos.

1.1.2. ¿Qué es una criptomoneda?

Una criptomoneda (o 'cripto') es un activo digital que puede circular sin necesidad de una autoridad monetaria central como un gobierno o un banco. En cambio, las criptomonedas se crean utilizando técnicas criptográficas que permiten a las personas comprarlas, venderlas o intercambiarlas de forma segura mediante el uso del Blockchain [3].

1.2. Objetivos

Los objetivos de este proyecto es familiarizarse con la tecnología del Blockchain mediante la programación de una criptomoneda.

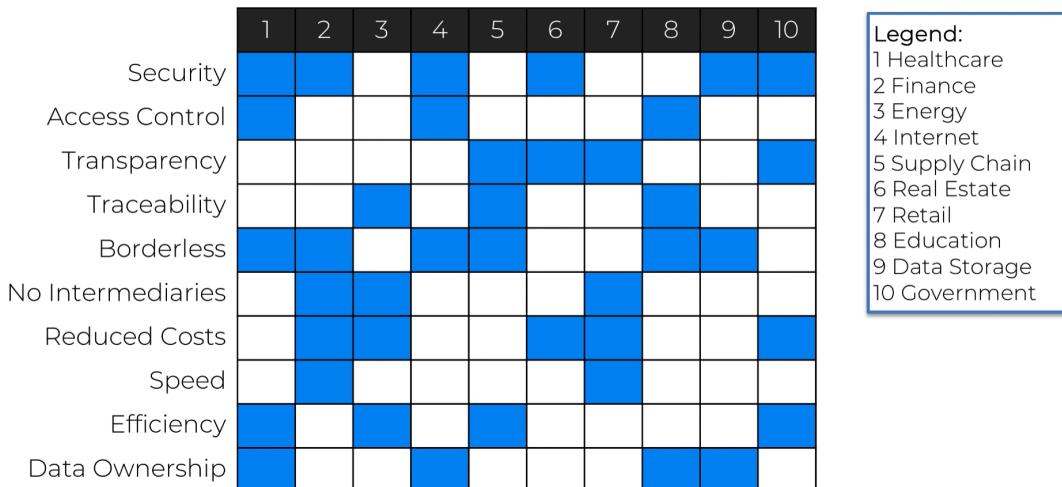


Figura 1. Características del Blockchain más relevantes a diferentes aplicaciones industriales.

- Ser capaz de implementar las características que diferencian al blockchain de otras tecnologías.
- Aprender cómo generar una red peer-to-peer (P2P) y que los usuarios accedan y modifiquen la información de la cadena de bloques.
- Asegurar que las transacciones que se llevan a cabo no están duplicadas y se sincronizan en todas las copias de las cadenas de la red distribuida.
- Ser capaz de desarrollar carteras digitales para los usuarios y que puedan crear transacciones.
- Desarrollar e implementar el minado de bloques de la cadena y generar los API endpoints necesarios para la comunicación entre peers.

1.3. Motivación

Considero que el Blockchain es una nueva forma de comprender el almacenamiento de datos y me interesa comprender cómo funciona esta tecnología. Creo que hoy en día se confunden los conceptos de criptomonedas con el blockchain pero realmente esta segunda es el esqueleto en el que se sustenta la aplicación de las 'cripto'.

A futuro, me gustaría crear una pequeña prueba de concepto de un sistema de votaciones descentralizado, seguro y transparente, basado en el Blockchain. Esto permitiría automatizar la votación en unas elecciones haciendo que cada persona pueda votar de forma segura desde cualquier dispositivo. Existen muchas más aplicaciones (**Figura 1**) que se le puede dar al Blockchain pero actualmente la más accesible para aprender es el desarrollo de una criptomoneda.

2. Tecnologías y Herramientas utilizadas

- **Javascript:** es un lenguaje de programación interpretado, dialecto del estándar ECMAScript. Se define como orientado a objetos, basado en prototipos, imperativo, débilmente tipado y dinámico.

- **Node.js**: Ideado como un entorno de ejecución de JavaScript orientado a eventos asíncronos, Node.js está diseñado para crear aplicaciones network escalables [4].
- **npm**: npm es el sistema de gestión de paquetes por defecto para Node.js, un entorno de ejecución para JavaScript, bajo Artistic License 2.0 [5].
- **Express**: Express es una infraestructura de aplicaciones web Node.js mínima y flexible que proporciona un conjunto sólido de características para las aplicaciones web y móviles. Lo usaré para el desarrollo de las API endpoints [6].
- **PubNub**: es una plataforma de comunicación en tiempo real. La utilizará para la generación de la red p2p [7].
- **Postman**: Postman es una aplicación que nos permite realizar pruebas API. Es un cliente HTTP que nos da la posibilidad de testear 'HTTP requests' a través de una interfaz gráfica de usuario, por medio de la cual obtendremos diferentes tipos de respuesta que posteriormente deberán ser validados. Con esta aplicación podremos testear las API endpoint que vaya desarrollando [8].
- **LaTeX**: es un sistema de composición de textos, orientado a la creación de documentos escritos que presenten una alta calidad tipográfica. Por sus características y posibilidades, es usado de forma especialmente intensa en la generación de artículos y libros científicos que incluyen, entre otros elementos, expresiones matemáticas [9].
- **Git**: Git es un software de control de versiones diseñado por Linus Torvalds, pensando en la eficiencia, la confianza y compatibilidad del mantenimiento de versiones de aplicaciones cuando estas tienen un gran número de archivos de código fuente [10].
- **GitHub**: es una forja para alojar proyectos utilizando el sistema de control de versiones Git. Se utiliza principalmente para la creación de código fuente de programas de ordenador [11].

3. Estimación de Recursos y Planificación

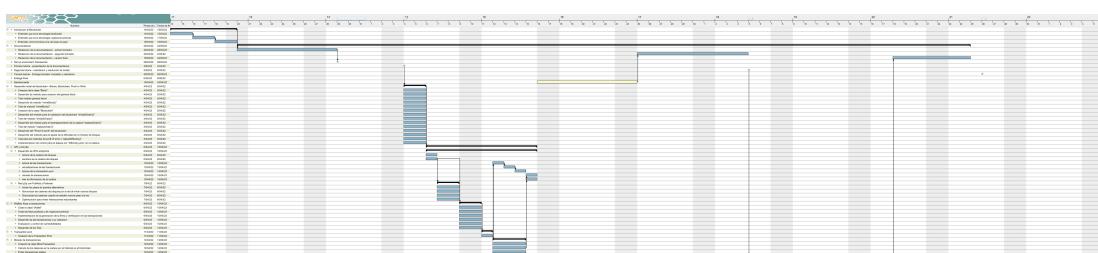


Figura 2. Planificación - Diagrama de Gantt

4. Análisis

En la **Figura 3**, se puede observar el principal caso de uso de la aplicación del proyecto 'Cryptochain': crear transacciones de dinero virtual. El usuario debe crearse una cartera virtual, con sus claves privadas y públicas. Con esta cartera, se podrán

crear transacciones que se validarán previo a ser incluidas en el contenedor de transacciones.

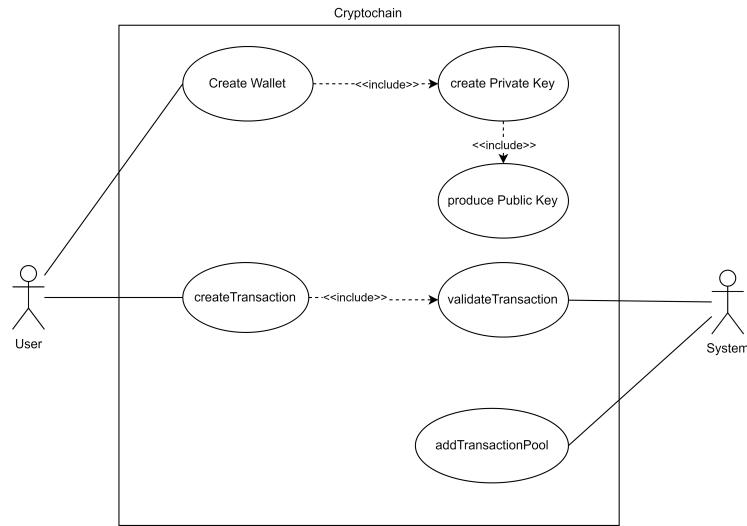


Figura 3. Caso de Uso: Crear una transacción y añadirla al contenedor de transacciones

En la **Figura 4**, se ilustra el caso de uso para poder minar el contenedor de transacciones a la cadena de bloques. El 'minero' recibirá una cantidad de criptomonedas como premio por el gasto computacional requerido en el minado. Finalmente se limpiará el contenedor de transacciones.

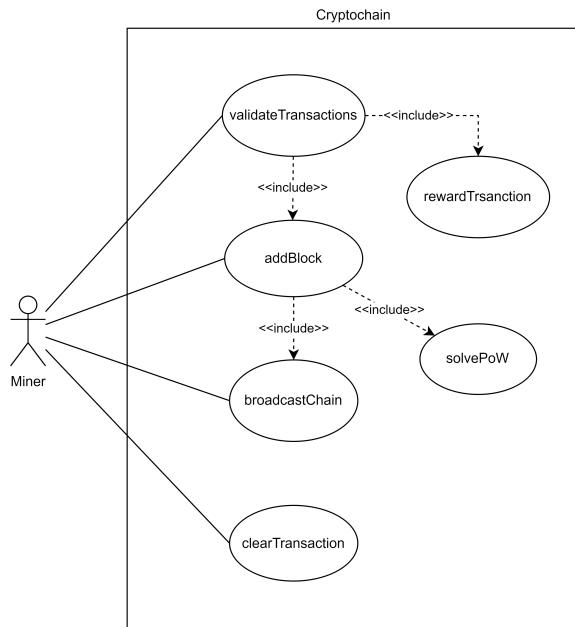


Figura 4. Caso de Uso: Minar el contenedor de transacciones en la cadena de bloques.

En la **Figura 5** se ilustra el diagrama de las clases necesarias para el desarrollo de esta aplicación.

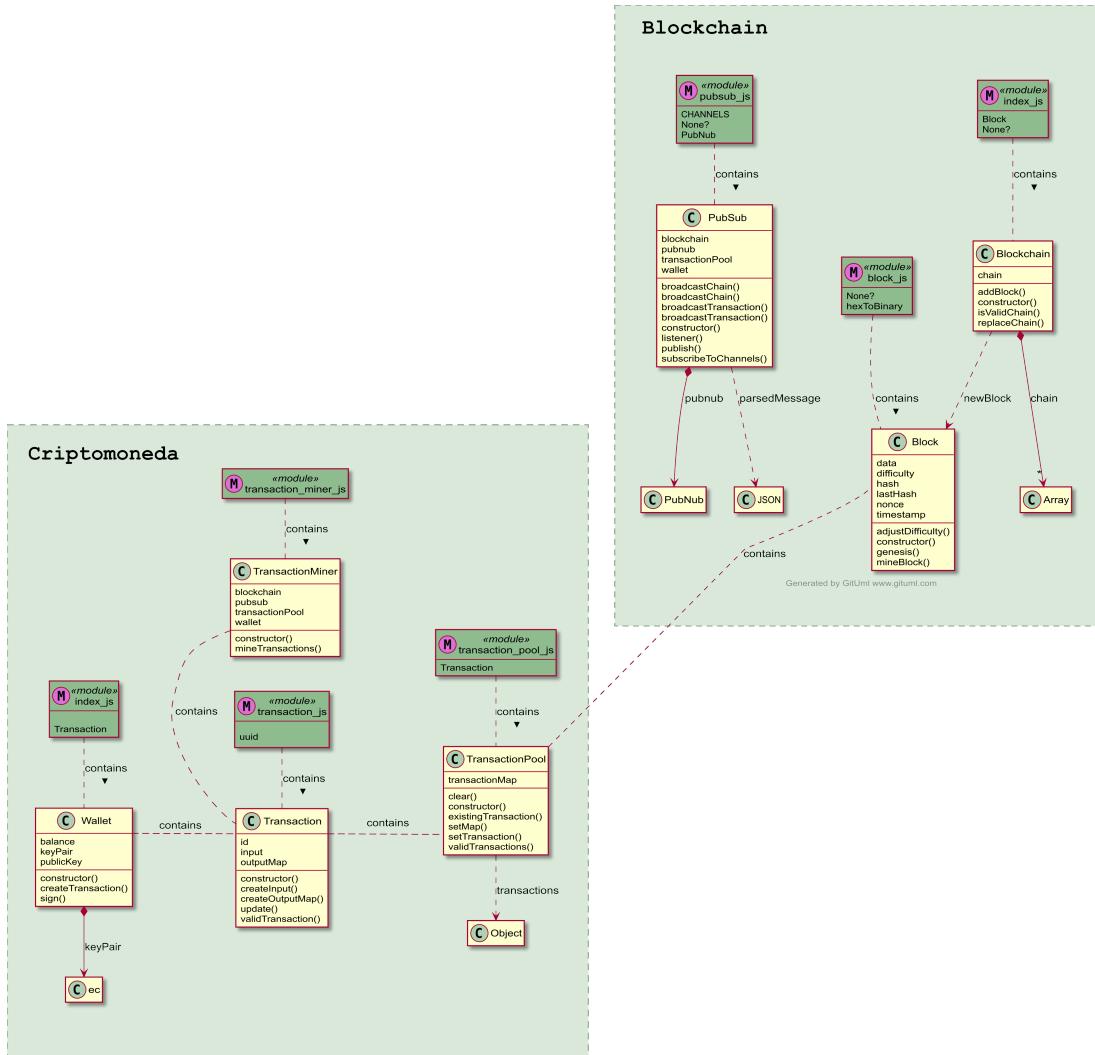


Figura 5. Diagrama de clases

5. Desarrollo de la Cadena de Bloques

5.1. Componentes del Blockchain

Para el desarrollo de la cadena de bloques es necesaria la creación de dos clases fundamentales: 'Block' y 'Blockchain' (Figura 5). Un bloque representa la unidad fundamental de almacenamiento en la cadena de bloques. Los bloques contienen la siguiente información (Figura 6.a):

- **lastHash**: es un identificador de 65 caracteres que hace referencia al *hash* del bloque anterior.
- **timestamp**: momento en el cual el bloque es creado.
- **nonce**: numero entero que es modificado una unidad para cumplir con el 'Proof of Work' (*PoW*). En este caso el PoW utilizado para mantener la integridad de la cadena es un numero consecutivos de ceros en el *hash* determinado por el parámetro *difficulty*. Ese concepto importante es relevante en el minado de bloques y se explicará más adelante en la sección 5.2.

- **difficulty**: número entero que representa el total de ceros a nivel binario consecutivos al inicio del *hash* tras encriptación del contenido del bloque.
- **data**: representa la información que se quiere almacenar en la cadena de bloques. En el caso de este proyecto, esa información sera el conjunto de transacciones que se dan entre usuarios de la cadena. Este concepto de transacciones se explicará más adelante en la sección 6.1.

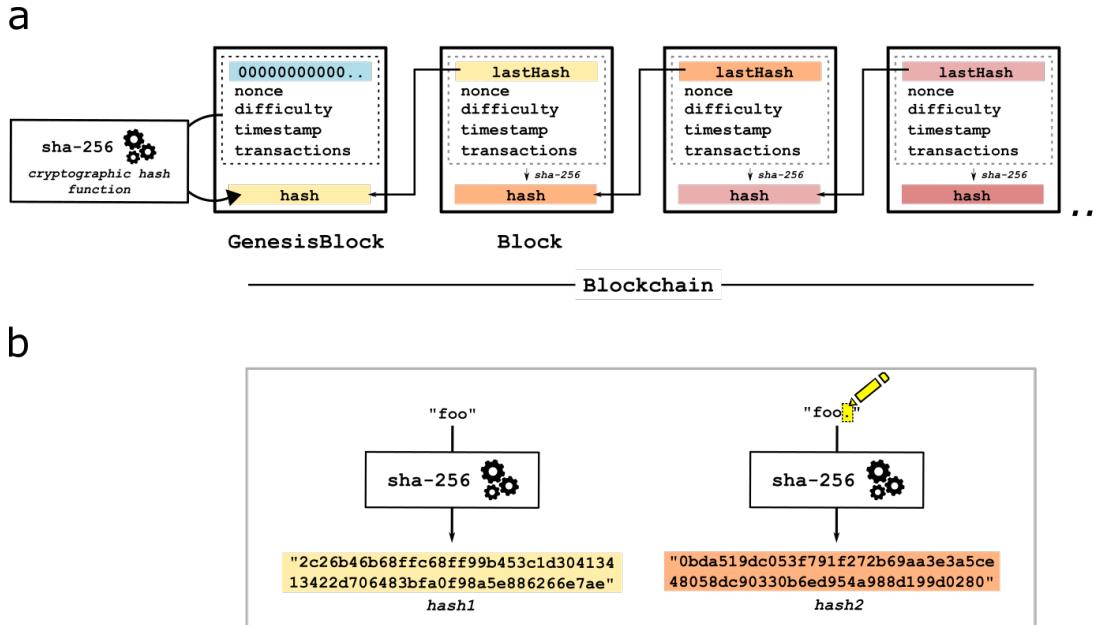


Figura 6. Componentes de la cadena de bloques a) Bloques de las cadenas con sus atributos, cuya información es encriptada por el algoritmo SHA-256. Cada bloque se enlaza con el siguiente gracias a los punteros *lastHash*. b) Cambios mínimos en los datos generan una modificación drástica en el *hash*.

Todos estos atributos del bloque se encriptan mediante el algoritmo de encriptación **SHA-256**. Tras procesar la información de los atributos, se genera un identificador único para el bloque, denominado *hash*.

- **hash**: identificador único del bloque creado a partir de toda la información de sus atributos. Nótese, que se incluye el *hash* al bloque anterior (*lastHash*). Al incluir este dato, creamos un puntero que, apunta al bloque anterior. Así se consiguen enlazar unos bloques con otros conformando la cadena de bloques: 'BlockChain'.

El algoritmo es extremadamente sensible a cualquier modificación en la información a encriptar. En la **Figura 6.b** se puede apreciar que la adición de un punto “.” en el string “foo” modifica drásticamente el *hash*. Si se genera alguna modificación en los datos del bloque por algún ataque, esto producirá un cambio completo del *hash* haciendo que la referencia del bloque siguiente a este se rompa.

Resumiendo esta primera sección de componentes, la cadena de bloques está formada por bloques que contienen información. Esta información es encriptada, generando los identificadores únicos *hash*. Cada bloque que se añade a la cadena, apunta al bloque anterior incluyendo su *hash* en su atributo *lastHash*. Añadir un bloque a la cadena es también conocido como “Minar un Bloque” donde entran en juego los conceptos de *PoW*, *nonce* y *difficulty*.

5.2. Minado de Bloques

El potencial de la tecnología 'Blockchain' reside, entre otros, en la seguridad de que los datos no van a ser modificados por ataques externos. Para poder mantener esa seguridad en la cadena además de utilizar el algoritmo SHA-256 se utilizan otro mecanismo de contención: *PoW*. Este *PoW* no es más que un pequeño puzzle u operación a resolver que está relacionado con el *hash* del bloque a minar. Es necesario hacer una pequeña introducción en el generación del *hash* para poder comprender el *PoW*. Como sabemos el *hash* se genera de forma eficiente mediante el algoritmo SHA-256. Este *hash* es un conjunto de 256 caracteres en valor binario transformados a 65 caracteres en valor hexadecimal. Por ejemplo, para el string:

```
"hello world"
```

Su *hash* en binario tras la encriptación es:

```
110100011100110010101101001001000111100110011111010000011  
010110001011110111100001001000011101010110000110101010001110  
001100110000110011110011110011110100001101010011110000111001  
110101100000111000110101000001111111001101101000111101100101  
000
```

Mientras que en hexadecimal sería:

```
68e656b251e67e8358bef8483ab0d51c6619f3e7a1a9f0e75838d41ff368f728
```

Un *PoW* conocido es el de concatenar 16 ceros consecutivos (*difficulty* = 16) en la forma hexadecimal del *hash*. Por ejemplo, el siguiente *hash* resuelve este PoW y sería valido como identificador del bloque:

```
000000000000000058bef8483ab0d51c6619f3e7a1a9f0e75838d41ff368f728
```

Para poder resolver este puzzle sin modificar la información del bloque, se añade una variable que va a estar constantemente cambiando. Esta variable es el atributo conocido como *nonce* en el bloque. El valor que toma esta variable es un número entero que va aumentando en una unidad hasta que el *PoW* se resuelva. Este puzzle se resuelve mediante la 'fuerza bruta' ya que no se puede cambiar el encriptado del *hash* de otra forma para resolver el *PoW* de forma más óptima.

A nivel computacional esto tiene un alto coste ya que como sabemos '0000' en binario equivale a '0' en hexadecimal. En este ejemplo concatenar 16 ceros en hexadecimal equivaldría a concatenar 64 ceros consecutivos en binario. Resolver este *PoW* supone una alta capacidad de CPU haciendo que el tiempo promedio para minar un bloque sea de 10 minutos (*MINE_RATE* = 10 min). Si se quisiese modificar algún valor de los atributos del bloque sucedería lo siguiente:

- **Modificación del propio *hash* del bloque:** dado que se modifica la información contenida en el bloque, esta al ser procesada por el sensible algoritmo SHA-256 generaría un *hash* distinto. Este nuevo *hash* es altamente probable que no cumpla con el *PoW* implementado. Por lo tanto, tendría que resolver de nuevo el *PoW* (10 minutos de media) mediante la modificación del *nonce* hasta resolver la *difficulty* decidida.

- **Pérdida del puntero entre bloques:** Aunque se resolviese el puzzle del *PoW*, el *hash* se ha modificado a un *nuevoHash*. Este *nuevoHash* es distinto al *lastHash* presente en el el bloque siguiente. Es decir, se rompe la cadena de bloques en uno de sus eslabones. Esto se puede observar en la **Figura 8** representado como un 'Cyber Attack'. Romper la cadena implica tener un número menor de bloques enlazados. Este concepto de tamaños de cadenas es relevante y se explicara más en detalle en la siguiente sección (5.3).

En este proyecto, el *PoW* consiste en resolver a nivel binario un número *difficulty* de ceros concatenados. Para no almacenar identificadores tan grandes (256 caracteres), el *hash* se guarda en hexadecimal (65 caracteres), aunque el *PoW* se teste en el paso anterior del encriptado. Esta decisión se toma para reducir el esfuerzo computacional. El flujo descrito se ilustra en la **Figura 7**.

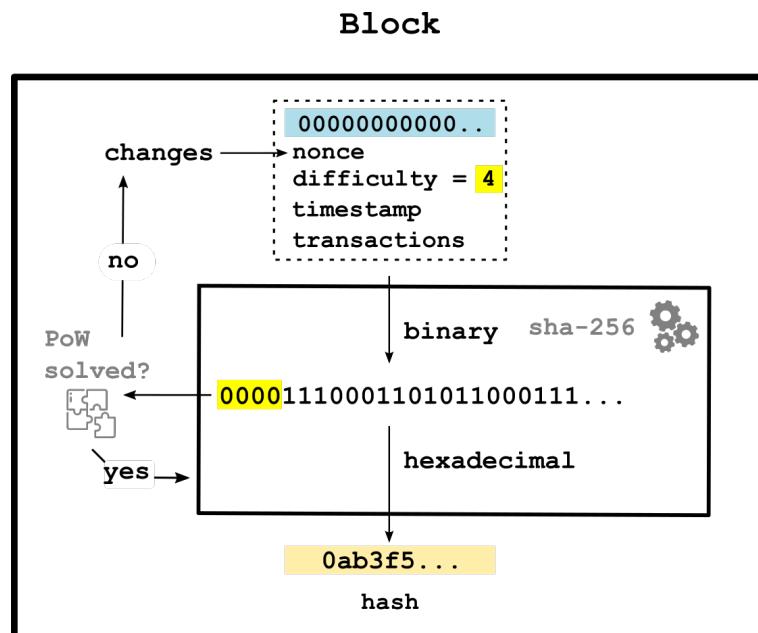


Figura 7. Flujo requerido para la correcta resolución del PoW durante el minado de bloques en este proyecto. Como ejemplo se utiliza el valor 4 para el atributo *difficulty*, aunque este puede variar en función del *MINE_RATE*.

Finalmente, un ultimo nivel de seguridad es añadido en la cadena de bloques gracias a la ya introducida constante *MINE_RATE*. El parámetro *difficulty* es dinámico. Esto quiere decir que se adapta a la velocidad de minado de la cadena de bloques. Cuanto más rápido se mine un bloque más aumentará el valor del *difficulty*. Cada vez será más complicado añadir un bloque a la cadena. Este parámetro dinámico permite que, en el caso de un ataque a la cadena, no se pueda reconstruir todos los *hash* de todos los bloques. Para ello, requerimos de un nuevo parámetro de configuración que es el *MINE_RATE*.

El *MINE_RATE* es una constante global que establece el tiempo medio de minado. Si se minan demasiado rápido los bloques, que el tiempo de minado es inferior a esta constante, entonces se aumenta el atributo *difficulty*. Por otro lado, si el tiempo de minado es muy alto, por lo tanto superior al *MINE RATE*, el parámetro *difficulty* disminuye. Esto permite adaptarse a posibles ataques que, como se ha explicado anteriormente, quieran reescribir todos los *hash* de la cadena de bloque.

Recapitulando, minar un bloque equivale a añadir un bloque nuevo a la cadena. Este proceso tiene que realizarse resolviendo un *PoW* que requiera un mínimo de

dificultad y esfuerzo computacional. Los *PoW* son puzzles sencillos pero que se resuelven mediante 'fuerza bruta' modificando el parámetro *nonce*. La dificultad del *PoW* es variable en función de la constante *MINE_RATE* y la variable *difficulty* establecidos. Este minado requiere una cierta capacidad de computo, por lo tanto, un coste económico de hardware, software y electricidad. Los 'mineros' de bloques son recompensados con un premio o *reward* que se explicará en la sección [6.3](#) para compensar esos gastos y motivar al mantenimiento de la cadena de bloques.

5.3. Red descentralizada y sincronización de la cadena de bloques

Cualquier aplicación basada en Blockchain es por defecto distribuida. Eso significa que la cadena de bloques esta distribuida entre los usuarios que la conforman, acceden y contribuyen a ella. Existe una comunidad de colegas ('peers' en inglés) que se benefician de esta aplicación y contribuyen a su correcto funcionamiento. Con esto en mente, podemos deducir que no existe ningún servidor central en el cual se almacena la cadena de bloques (red centralizada), si no que se almacena en una red de colega-a-colega o 'peer-to-peer' (P2P). Cada peer puede tener su propia copia de la cadena y acceder sin necesidad de ningún intermediario a la información que esta contiene. Adicionalmente, puede tomar el rol de 'minero' y añadir nuevos bloques a la cadena.

En esta situación podemos pensar que se pueden dar inconsistencias entre peers si cada uno almacena su propia copia de la cadena y añade los bloques que vea conveniente. Es por esto por lo que entra el juego el papel de la comunidad y la importancia de la sincronización de los peers.

En este proyecto se hará uso de la herramienta PubNub para solventar la sincronización entre peers. Nos permite crear aplicaciones que requieren sincronización en tiempo real. Es una herramienta que funciona con un sistema de publicaciones y suscripciones. Similar a la suscripción a una revista, pero donde son los propios lectores los que escriben la revista y la publican. De esta forma, con PubNub se pueden enviar mensajes (publicar) en tiempo real a aquellos peers suscritos. Todos los datos que son publicados se replican automáticamente generando un sistema descentralizado. En este proyecto se crean dos canales de suscripción en PubNub. Uno para la propia cadena de bloques y otro para las transacciones almacenadas en el contenedor de transacciones (sección [6.3](#)). En la clase 'PubSub()' (**Figura 5**) se implementó el envío de mensajes al canal de suscripción 'BLOCKCHAIN'.

Gracias a este sistema, estamos recibiendo en tiempo real cualquier publicación que se da en la cadena, por lo tanto cualquier nueva adición de un bloque. Con este sistema de publicaciones y suscripciones podemos crear una lógica sencilla para que nuestra cadena este siempre actualizada. Si recibimos un mensaje en la red de que hay una nueva cadena, y comprobamos que esta cadena contiene un mayor número de bloques tenemos que sustituirla por nuestra cadena actual ya que la nuestra estará desactualizada.

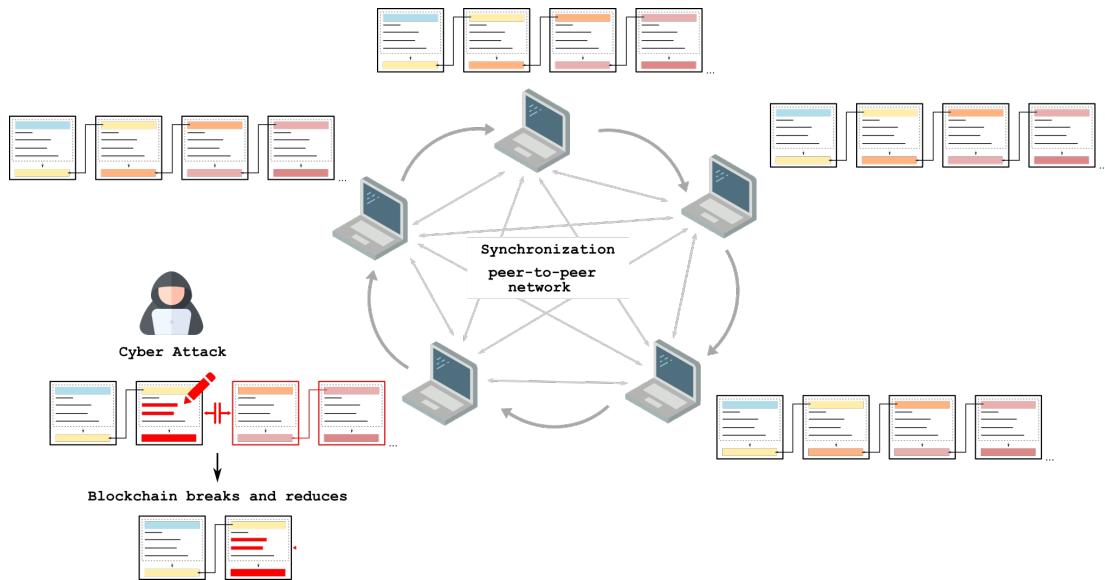


Figura 8. Red P2P y la importancia de la sincronización de los peers evitando posibles ataques a la cadena de bloques

En el caso en el que alguien quisiese modificar la información de uno de los bloques, como sabemos, esa cadena se rompería por la modificación de los *hash* derivando en una cadena con un menor numero de bloques. Al estar en constante escucha de la red, se detectaría que esa cadena no es correcta ya que es más corta que la recibida y se actualizaría por la cadena más larga (**Figura 8** y **Figura 5**).

En un mundo ideal, podríamos confiar en todos aquellos peers que minasen un nuevo bloque y felizmente añadirla a nuestra cadena de bloques. Pero desafortunadamente, tenemos que hacer una validación previa de los bloques minados para comprobar que es una cadena valida. La validación tiene varios test, entre ellos comprobar que empiezan por el bloque Génesis, que cada bloque tiene los mismos atributos o que la dificultad del *PoW* no se haya modificado bruscamente en algún ataque.

6. Desarrollo de la criptomoneda

6.1. Transacciones

La criptomoneda es un activo digital similar al dinero digital que almacenamos en nuestro bancos. La diferencia principal con las monedas es que las cripto, residen en una cadena de bloques donde los usuarios pueden realizar transacciones sin depender de ningún intermediario. También, la creación de una criptomoneda depende de desarrolladores con conocimientos de programación. Para la creación de una moneda nacional hay que seguir una serie de procedimientos burocráticos diferentes.

Para la creación de una criptomoneda es necesario el uso del Blockchain, ya que es en los datos de los bloques donde se van a almacenar esas transacciones que realizan los usuarios de las criptomonedas. Hablamos de transacciones y es un concepto similar a una transferencia bancaria entre dos personas. Una transacción consta de la siguiente información básica:

- **input:** información de la persona que envía la transacción (*sender*); número identificativo del *sender* (como un DNI); cantidad que se quiere enviar; número identificativo de la persona a la que se le quiere enviar esa cantidad; y una firma del *sender* que confirma que ha sido esa persona la que envía la cantidad.

- **output:** es una confirmación de la cantidad que se envía al receptor y el dinero restante que se le quedaría al *sender* en su 'cuenta bancaria' también denominado balance.

De esta forma se puede almacenar a quien, cuánto, que 'dinero' tiene en su cuenta y si la persona que ha enviado el 'dinero' ha dado su consentimiento mediante la firma.

6.2. Wallet y firmas digitales

Para realizar una firma digital de una transacción es necesario tener una cartera digital o *wallet*. Dentro de esta cartera se almacenan dos claves:

- **clave privada:** es una llave que permite encriptar información, en este caso el output de la transacción. La única forma de desencriptarla es haciendo uso de su clave pública en combinación de la clave privada el receptor.
- **clave pública:** tiene dos papeles fundamentales. Como se ha explicado arriba, sirve para comprobar que la información recibida proviene del *sender*. Si no proveniese del *sender*, no se podría desencriptar el output. Por otro lado, también actúa como 'DNI' público o identificador único de nuestra persona. En el input de la transacción hacemos uso de esta clave privada para identificarnos e identificar a la persona a la que le queremos enviar nuestras criptomonedas.

Estas dos claves se almacenan en la wallet. Además es en la wallet donde tenemos la capacidad de crear transacciones, sería similar a nuestra tarjeta de débito.

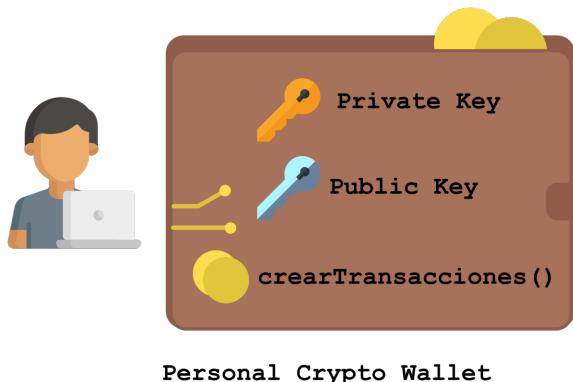


Figura 9. Cartera virtual que contiene la clave privada y pública requeridas para poder crear y recibir transacciones.

6.3. Contenedor de transacciones y minado

El contenedor de transacciones o Transaction Pool es donde se almacenan las transacciones previo a introducirlas en un bloque de la cadena. En otras palabras, es donde se guardan las transacciones antes de que uno de los mineros resuelva el PoW y las pueda incorporar en la cadena.

Como se puede dar la situación de que hagamos más de una transacción antes de que se incorporen los datos a un nuevo bloque, es necesario poder actualizar la transacción. De igual modo que con la cadena de bloques, tenemos que estar suscritos a un canal de PubNub por si algún usuario realiza o actualiza una transacción. En PubNub coordinamos dos canales de subscripción y publicación: 'TRANSACTIONS' y 'BLOCKCHAIN'. Esto permite estar constantemente escuchando posibles cambios

dentro del transaction pool (canal TRASANCTIONS) y cambios en la cadena de bloques ('BLOCKCHAIN'). Si un minero añade un bloque a la cadena, entonces mandaría un mensaje por el canal 'TRANSACTIONS' haciendo que se limpie esa transaction pool. También, si algún usuario de la criptomonedas realiza una transacción, la añadiría a la nueva transaction pool y así sucesivamente. La transaction pool puede contener un número variable de transacciones no definidos. Como se puede observar en la **Figura 10** existen un total de 739038 bloques en la cadena de bloques que almacena las transacciones de Bitcoin.

Últimos bloques			
Los bloques minados más recientemente			
Altura	Minado	Minador	Tamaño
739038	22 minutos	F2Pool	1.120.001 bytes
739037	25 minutos	F2Pool	1.602.410 bytes
739036	27 minutos	Unknown	1.653.164 bytes
739035	33 minutos	Unknown	1.447.730 bytes
739034	53 minutos	SlushPool	1.606.731 bytes
739033	56 minutos	SlushPool	1.558.874 bytes

[Ver todos los bloques →](#)

Figura 10. Bloques minados en la cadena de bloques de la criptomonedas Bitcoin a día 3 de Junio de 2022

Cada bloque se mina en un rango aproximado de entre 20 minutos y 1 hora. Aparece el usuario que ha minado los bloques y el tamaño de los bloques de hasta 1GB. Podemos acceder al último bloque minado, que almacena el transaction pool, y también la recompensa obtenida por haber minado el bloque. En este ejemplo, el último bloque minado es el [Bloque 739039](#).

Recompensa de bloque	6.25000000 BTC
----------------------	----------------

Figura 11. Recompensa obtenida por minar el bloque 739039 del blockchain en el que se sustenta la criptomonedas Bitcoin.

Finalmente, cada vez que un 'minero' mina un bloque en la cadena, este tiene que recibir una recompensa que se traduce en una transacción de premio o TransactionMiner. En este caso, los mineros reciben una transacción cada vez que minan un bloque y son recompensados 'económicamente' con una cantidad de criptomonedas. En el ejemplo anterior del Bloque 739039 (**Figura 11**), 6.25 BTC equivalen actualmente a 176.770,96€ (3 de Junio de 2022).

6.4. Implementación de la criptomonedas

En la **Figura 5** se pueden observar las clases 'Wallet()', 'Transaction()', 'TransactionPool()' y 'TransactionMiner()' que contienen los atributos y métodos necesarios

para llevar a cabo todas y cada una de los procesos necesarios para la correcta implementación de la moneda. En la clase 'PubSub()' se implementó el envío de mensajes al canal de subscripción 'TRANSACTION'.

En este proyecto, se creó adicionalmente un módulo de configuración ('config.js') en el cual se fijaban las constantes de STARTING_BALANCE, INITIAL_DIFFICULTY, GENESIS_DATA, MINING_REWARD, REWARD_INPUT. Estas constantes son necesarias para configurar las características de cada moneda.

La aplicación de criptomonedas se desarrolló en Express y Node.js. Se creó un entorno de desarrollo 'dev' para correr la aplicación en local en el puerto '3000'. Además, se generó una variable de entorno de puerto aleatorio entre los valores '3000' y '4000' para la simulación de los 'peers' en local. Este segundo entorno se denominó 'dev-peer'. Diferentes Application Programming Interface (API) fueron desarrolladas para recibir la cadena de bloques y las transacciones, así como poder postear nuevos bloques a la cadena o añadir transacciones a la transaction pool de la aplicación. Una última API fue desarrollada para almacenar la transaction pool en el nuevo bloque minado y posteriormente limpiar el contenedor de transacciones.

Por otro lado, cada clase y módulo desarrollado en el proyecto tiene un test de pruebas unitario para poder testear el código y que cumpla con los estándares de la tecnología Blockchain. Cada módulo de Node.js tiene asociado un archivo de test ('module.test.js') desarrollado con 'jest' ([Sección 8.1](#)).

El código de este proyecto se puede encontrar en el repositorio <https://github.com/sara-perezg/Blockchain>.

7. Conclusiones

Se ha conseguido desarrollar una prueba de concepto de una criptomoneda en este proyecto. El trabajo se dividió en dos partes, la implementación de la cadena de bloques y la implementación de la criptomoneda. Durante el desarrollo del proyecto se han comprendido y aplicado conceptos complejos y actuales de la tecnología Blockchain. Adicionalmente, se podrían realizar mejoras entre las que destacan, solucionar bugs, crear una red P2P en un entorno de producción y no en una simulación en local. También, se podría añadir una interfaz gráfica de usuario que facilitase la accesibilidad a los usuarios. Con todo esto, considero que se han cumplido los objetivos del proyecto (sección 1.2) y que podría aplicar los conocimientos adquiridos en un futuro.

Referencias

- [1] Solunion, “¿qué es y para qué sirve la tecnología blockchain?” Oct 2021. [Online]. Available: <https://www.solunion.cl/blog/que-es-y-para-que-sirve-la-tecnologia-blockchain/>
- [2] IBM, “¿qué es la tecnología de blockchain? - ibm blockchain,” 2021. [Online]. Available: <https://www.ibm.com/es-es/topics/what-is-blockchain>
- [3] nerdWallet, “what is cryptocurrency? here's what investors should know,” 2022. [Online]. Available: <https://www.nerdwallet.com/article/investing/cryptocurrency>
- [4] Node.js, “acerca de node.js 2022,” 2022. [Online]. Available: <https://nodejs.org/es/about/>
- [5] npm, jun 2022. [Online]. Available: <https://www.npmjs.com>

- [6] “infraestructura de aplicaciones web node.js 2022,” 2022. [Online]. Available: <https://expressjs.com/es/>
- [7] “real-time in-app chat and communication platform 2022,” 2022. [Online]. Available: <https://www.pubnub.com>
- [8] postman, “postman,” 2022. [Online]. Available: <https://www.postman.com>
- [9] Latex, jun 2022. [Online]. Available: <https://www.latex-project.org>
- [10] git, jun 2022. [Online]. Available: <https://git-scm.com>
- [11] github, jun 2022. [Online]. Available: <https://github.com>

8. Anexo

8.1. Informe de Tests

El reporte de test se realizo con el paquete [jest-html-reporter](#).

Test Report

Started: 2022-06-04 21:06:34

Suites (6)	Tests (65)		
6 passed 0 failed 0 pending	65 passed 0 failed 0 pending		
C:\Users\sarap\Github\Blockchain\cryptochain\util\crypto-hash.test.js 2.758s			
<code>cryptoHash()</code>	<i>generates a SHA-256 hashed output</i>	passed	0.009s
<code>cryptoHash()</code>	<i>produces the same has with the same input arguments in any order</i>	passed	0s
<code>cryptoHash()</code>	<i>produces a unique hash when the properties have changed on an input</i>	passed	0.001s
C:\Users\sarap\Github\Blockchain\cryptochain\blockchain\block.test.js 3.362s			
<code>Block</code>	<i>has a timestamp, lastHash, hash, nonce, difficulty and data property</i>	passed	0.008s
<code>Block > genesis()</code>	<i>returns a Block instance</i>	passed	0.001s
<code>Block > genesis()</code>	<i>returns the genesis data</i>	passed	0.001s
<code>Block > mineBlock()</code>	<i>returns a Block instance</i>	passed	0.001s
<code>Block > mineBlock()</code>	<i>sets the 'lastHash' to be the hash of the 'lastBlock'</i>	passed	0.002s
<code>Block > mineBlock()</code>	<i>sets the 'data'</i>	passed	0.003s
<code>Block > mineBlock()</code>	<i>sets a 'timestamp'</i>	passed	0.001s
<code>Block > mineBlock()</code>	<i>creates a SHA-256 'hash' based on the proper inputs</i>	passed	0.001s
<code>Block > mineBlock()</code>	<i>sets a 'hash' that matches the difficulty criteria</i>	passed	0.001s
<code>Block > mineBlock()</code>	<i>adjust the difficulty</i>	passed	0.001s
<code>Block > adjustDifficulty()</code>	<i>raises the difficulty for a quickly mined block</i>	passed	0.001s
<code>Block > adjustDifficulty()</code>	<i>lowers the difficulty for a slowly mined block</i>	passed	0s
<code>Block > adjustDifficulty()</code>	<i>has a lower limit of 1</i>	passed	0s

C:\Users\sarap\Github\Blockchain\cryptochain\blockchain\index.test.js			3.462s
Blockchain	<i>contains a 'chain' Array instance</i>	passed	0.005s
Blockchain	<i>starts with the genesis block</i>	passed	0.003s
Blockchain	<i>adds a new block to the chain</i>	passed	0.002s
Blockchain > isValidChain() > when the chain does not start with the genesis block	<i>returns false</i>	passed	0.002s
Blockchain > isValidChain() > when the chain starts with the genesis block and has multiple blocks > and a lastHash reference has changed	<i>returns false</i>	passed	0.002s
Blockchain > isValidChain() > when the chain starts with the genesis block and has multiple blocks > and the chain contains a block with an invalid field	<i>returns false</i>	passed	0.002s
Blockchain > isValidChain() > when the chain starts with the genesis block and has multiple blocks > and the chain contains a block with a jumped difficulty	<i>returns false</i>	passed	0.001s
Blockchain > isValidChain() > when the chain starts with the genesis block and has multiple blocks > and the chain does not contain any invalid blocks	<i>returns true</i>	passed	0.002s
Blockchain > replaceChain() > when the new chain is not longer	<i>does not replace the chain</i>	passed	0.002s
Blockchain > replaceChain() > when the new chain is not longer	<i>logs an error</i>	passed	0.002s
Blockchain > replaceChain() > when the new chain is longer > when the new chain is invalid	<i>does not replace the chain</i>	passed	0.003s
Blockchain > replaceChain() > when the new chain is longer > when the new chain is invalid	<i>logs an error</i>	passed	0.001s
Blockchain > replaceChain() > when the new chain is longer > when the new chain is valid	<i>replaces the chain</i>	passed	0.003s
Blockchain > replaceChain() > when the new chain is longer > when the new chain is valid	<i>logs about the new chain replacement</i>	passed	0s

C:\Users\sarap\Github\Blockchain\cryptochain\wallet\index.test.js			6.783s
Wallet	<i>has a 'balance'</i>	passed	0.177s
Wallet	<i>has a 'publicKey'</i>	passed	0.136s
Wallet > signing data	<i>verifies a signature</i>	passed	0.78s
Wallet > signing data	<i>does not verify an invalid signature</i>	passed	1.15s
Wallet > createTransaction() > and the amount exceeds the balance	<i>throws an error</i>	passed	0.162s
Wallet > createTransaction() > and the amount is valid	<i>creates an instance of 'Transaction'</i>	passed	0.322s
Wallet > createTransaction() > and the amount is valid	<i>matches the transaction input with the wallet</i>	passed	0.369s
Wallet > createTransaction() > and the amount is valid	<i>outputs the amount the recipient</i>	passed	0.314s

C:\Users\sarap\Github\Blockchain\cryptochain\wallet\transaction-pool.test.js			13.647s
TransactionPool > setTransaction()	<i>adds a transaction</i>	passed	0.319s
TransactionPool > existingTransaction()	<i>returns an existing transaction given an input address</i>	passed	0.262s
TransactionPool > validTransactions()	<i>returns valid transaction</i>	passed	5.041s
TransactionPool > validTransactions()	<i>logs errors for the invalid transactions</i>	passed	3.339s
TransactionPool > clear()	<i>clears the transactions</i>	passed	0.147s
TransactionPool > clearBlockchainTransactions()	<i>clears the pool of any existing blockchain transactions</i>	passed	1.067s

C:\Users\sarap\Github\Blockchain\cryptochain\wallet\transaction.test.js			11.572s
Transaction	<i>has an 'id'</i>	passed	0.362s
Transaction > outputMap	<i>has an 'outputMap'</i>	passed	0.269s
Transaction > outputMap	<i>outputs the amount to the recipient</i>	passed	0.301s
Transaction > outputMap	<i>outputs the remaining balance for the 'senderWallet'</i>	passed	0.317s
Transaction > input	<i>has an 'input'</i>	passed	0.406s
Transaction > input	<i>has a 'timestamp' in the input</i>	passed	0.425s
Transaction > input	<i>sets the 'amount' to the 'senderWallet' balance</i>	passed	0.374s
Transaction > input	<i>sets the 'address' to the 'senderWallet' publicKey</i>	passed	0.33s
Transaction > input	<i>signs the input</i>	passed	0.828s
Transaction > validTransaction() > when the transaction is valid	<i>returns true</i>	passed	0.621s
Transaction > validTransaction() > when the transaction is invalid > and a transaction outputMap value is invalid	<i>returns false and logs an error</i>	passed	0.22s
Transaction > validTransaction() > when the transaction is invalid > and the transaction input signature is invalid	<i>returns false and logs an error</i>	passed	0.794s
Transaction > update() > and the amount is invalid	<i>throws an error</i>	passed	0.224s
Transaction > update() > and the amount is valid	<i>outputs the amount to the next recipient</i>	passed	0.325s
Transaction > update() > and the amount is valid	<i>subtracts the amount from the original sender output amount</i>	passed	0.33s
Transaction > update() > and the amount is valid	<i>maintains a total output that matches the input amount</i>	passed	0.34s
Transaction > update() > and the amount is valid	<i>re-signs the transaction</i>	passed	0.356s
Transaction > update() > and the amount is valid > and another update for the same recipient	<i>adds to the recipient amount</i>	passed	0.456s
Transaction > update() > and the amount is valid > and another update for the same recipient	<i>subtracts the amount from the original sender output amount</i>	passed	0.418s
Transaction > rewardTransaction()	<i>creates a transaction with the reward input</i>	passed	0.254s
Transaction > rewardTransaction()	<i>creates one transaction for the miner with the 'MINING_REWARD'</i>	passed	0.278s