Programación de Sistemas Distribuidos Práctica 4: Blockchain con ruby

Carlos Bermúdez Expósito
Arturo Requejo Portilla
Augusto Messina

UNIVERSIDAD NEBRIJA

Tabla de contenido

I. Enunciados

Esta práctica es grupal, para la entrega, tenéis que daros de alta como grupo en la asignatura de prácticas y que entregue la práctica el responsable del grupo. Si algún miembro del grupo no participa, el responsable debe comunicarlo en la entrega.

A. Ejercicio 1

Del siguiente repositorio https://github.com/apradillap/simple-blockchain-in-ruby

a) Ejecuta la blockchain
b) Analiza cómo funciona y añade capturas de pantalla para mostrar evidencias de la compresión de la ejecución.

B. Ejercicio 2

¿Qué es el bloque génesis?

C. Ejercicio 3

¿Qué elementos tiene cada bloque?

D. Ejercicio 4

¿Qué hace el método def compute_hash_with_proof_of_work? ¿Qué significa Proof of work?

E. Ejercicio 5

Haz un fork de la práctica. ¿Te atreves a realizar algún cambio? Se valorarán más las propuestas ingeniosas.

II. Introducción

Blockchain es un libro mayor compartido e inalterable que facilita el proceso de registro de transacciones y de seguimiento de activos en una red de negocios. Un activo puede ser tangible o intangible. Prácticamente cualquier cosa de valor, puede rastrearse y comercializarse en una red de blockchain, reduciendo así el riesgo y los costes para todos los involucrados.

Consta de 3 elementos principales:

- **Tecnología de libro mayor distribuido:** Todos los participantes de la red tienen acceso al libro mayor distribuido y a su registro inalterable de transacciones. Así, las transacciones se registran solo una vez, eliminando la duplicación del esfuerzo, típica en las redes de negocios tradicionales.
- **Registros inalterables:** Una vez grabada en el libro mayor compartido, nadie puede modificar una transacción. El registro de una transacción incluye un error, se añadirá una nueva transacción para revertir el error, pero ambas transacciones serán visibles.
- **Contratos inteligentes:** Para acelerar las transacciones, un conjunto de reglas, llamado contrato inteligente, se almacena en el blockchain y se ejecuta automáticamente.

El funcionamiento principal de blockchain es que cada una de las transacciones que tienen lugar se registran como un "bloque" de datos. Estas transacciones muestran el movimiento de un activo. El bloque de datos registra la información que elijamos: quién, qué, cuándo, dónde, cuánto e incluso el estado, es decir, se almacena lo que le pongamos, lo más conocido a nivel mundial son las transacciones donde se almacenan cosas tales como la fecha, la cantidad enviada, el emisor, receptor, …

Cada bloque está conectado al bloque anterior y al posterior donde a medida que un activo se mueve, estos bloques forman una cadena de datos. Los bloques de la cadena verifican el tiempo exacto como la secuencia de las transacciones, y se unen de forma segura, evitando que se produzcan modificaciones o inserciones entre dos bloques existentes, para añadir seguridad y evitar posibles ataques.

Las transacciones se unen y forman una cadena irreversible llamada blockchain dónde el bloque adicional refuerza la verificación del anterior y, por lo tanto, de todo el blockchain. Esto hace que dicha cadena esté a prueba de manipulaciones, lo que conforma la ventaja principal de la inalterabilidad. Esto evita que alguien modifique la cadena y crea un libro mayor distribuido de transacciones, en el que nosotros y otros miembros de la red puedan confiar.

Blockchain es un sistema distribuido total porque la información que se almacena en la cadena de bloques se encuentra en múltiples nodos o computadoras interconectadas en la red. Esto significa que cada nodo en la red tiene una copia idéntica de la hilera de bloques.

Cada transacción que se realiza en la red de blockchain es validada y verificada por varios nodos de la red. Estos nodos están ubicados en diferentes partes del mundo, y utilizan algoritmos de consenso para comprobar que la transacción es válida y que la confesión registrada en la hilera de bloques es precisa. Una vez que la transacción ha sido validada y verificada por los nodos, se agrega un nuevo bloque a la cadena de bloques, y este nuevo bloque se convierte en parte del registro histórico inmutable de la red, para albergar en él la información que se requiera.

El hecho de que la información esté distribuida en múltiples nodos hace que la red sea más resistente a los ataques de ciberseguridad, ya que un atacante tendría que tomar el control de la mayoría de los nodos en la red para alterar la información en la cadena de bloques. Esto lo hace muy seguro y confiable para muchas aplicaciones diferentes, especialmente para aquellas que involucran transferencias de valor o información sensible.

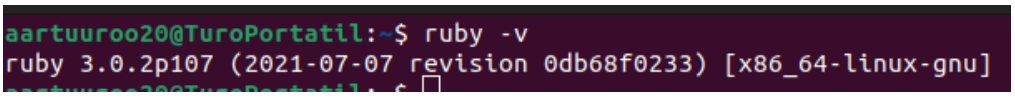
III. Resultados

Para poder realizar un correcto funcionamiento del repositorio proporcionado primero necesitaremos instalar ruby en nuestro ordenador, en nuestro caso lo hemos instalado en linux ubuntu. Para ello, empezaremos instalando ruby mediante el siguiente comando:

sudo apt-get install ruby-full

Una vez instalado comprobaremos si se ha realizado bien la instalación mediante el siguiente comando:

ruby -v

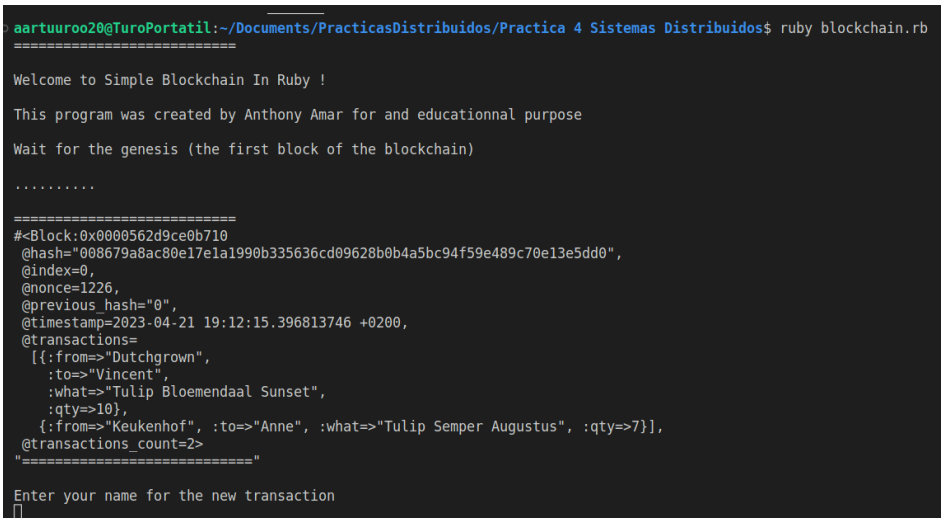


A. Ejercicio 1

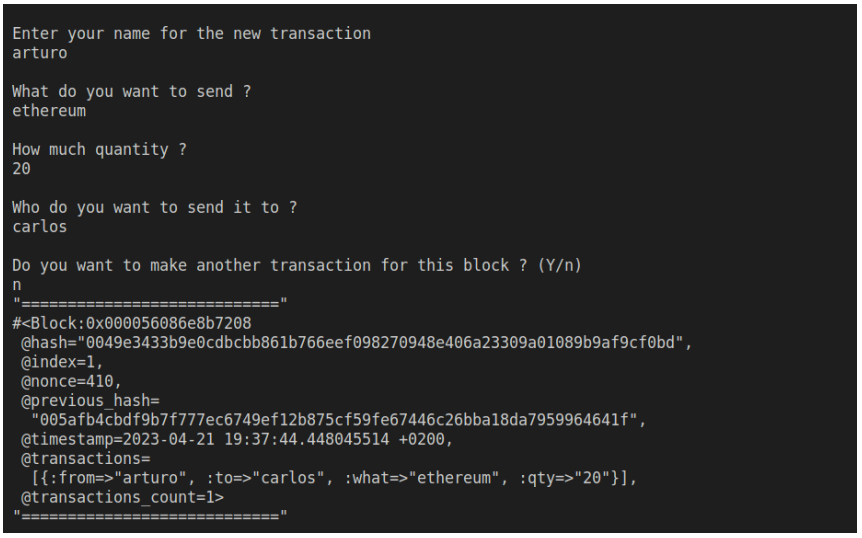
a) Para realizar la ejecución del repositorio proporcionado ejecutaremos el siguiente comando en la terminal:

ruby blockchain.rb

Al ejecutarlo nos muestra lo siguiente:



También, como vemos en la parte inferior, el programa comienza a realizar una serie de preguntas donde los resultados se añadirán como información que se almacenará en el nuevo bloque, también podemos crear varias transacciones en un mismo bloque.



```
Enter your name for the new transaction
arturo

What do you want to send ?
ethereum

How much quantity ?
20

Who do you want to send it to ?
carlos

Do you want to make another transaction for this block ? (Y/n)
n
"============================"
#<Block:0x000056086e8b7208
 @hash="0049e3433b9e0cdbcbb861b766eef098270948e406a23309a01089b9af9cf0bd",
 @index=1,
 @nonce=410,
 @previous_hash=
  "005afb4cbdf9b7f777ec6749ef12b875cf59fe67446c26bba18da7959964641f",
 @timestamp=2023-04-21 19:37:44.448045514 +0200,
 @transactions=
  [{:from=>"arturo", :to=>"carlos", :what=>"ethereum", :qty=>"20"}],
 @transactions_count=1>
"============================"
```

```
Enter your name for the new transaction
augustus

What do you want to send ?
bitcoin

How much quantity ?
10

Who do you want to send it to ?
carlos

Do you want to make another transaction for this block ? (Y/n)
n
"============================"
#<Block:0x000056000da4b3a8
 @hash="00b232d77e23b6ed190b04442db025529f6301a1509502c8cedf3790316ffd8a",
 @index=1,
 @nonce=99,
 @previous_hash=
  "001eb4f86c28306caf03f1e87bf5247fe3c35d900ddac8c29dcda8b13baf0d85",
 @timestamp=2023-04-21 19:39:24.639915277 +0200,
 @transactions=
  [{:from=>"arturo", :to=>"carlos", :what=>"ethereum", :qty=>"20"},
   {:from=>"augustus", :to=>"carlos", :what=>"bitcoin", :qty=>"10"}],
 @transactions_count=2>
```

b) Analiza cómo funciona y añade capturas de pantalla para mostrar evidencias de la compresión de la ejecución.

Como podemos ver en las capturas adjuntas, este código explicado a muy alto nivel lo que hace es crear nuevos bloques en una red de blockchain. Esto se realiza de la siguiente manera, para crear un bloque ejecutamos ***ruby blockchain.rb,*** en primer lugar se crea el primer bloque automáticamente (bloque génesis) y a este se de adherirá el primer bloque creado por nosotros que sería el segundo de la cadena.

Para crear nuestro bloque tendremos que responder una serie de preguntas, esta va ser la información que almacene el bloque creado, albergando de estas como información más importante las transacciones, donde se pueden añadir varias. Al finalizar la creación de nuestro bloque nos dirá si queremos crear otro más y así en bucle hasta crear todos los que queramos, estos se conectarán con su anterior mediante el hash único que tiene cada uno.

B. Ejercicio 2

Un bloque génesis es el primer bloque en una cadena de bloques blockchain, siendo este el punto de partida de la cadena y no teniendo ningún bloque anterior al se refiere

Se crea durante la creación inicial de la cadena de bloques, conteniendo así información especial siendo parecido a una marca de tiempo única y datos únicos que son necesarios para el correcto funcionamiento.

El bloque génesis es importante porque establece una base de la cadena de bloques y cualquier otro bloque creado se basará en este mismo manteniendo la comunicación entre dos nodos sólo pudiendo emparejarse aquellos nodos con el mismo bloque génesis. Mantener la cadena de bloques es crucial para garantizar la integridad de la cadena de bloques en su conjunto. El bloque génesis del proyecto es el siguiente:

```
==========================
#<Block:0x000055ef2a69b058
 @hash="00fd775dd779570d747b3b8d856d56e5243690dfd82f3c9735ce2fd01d50dc78",
 @index=0,
 @nonce=15,
 @previous_hash="0",
 @timestamp=2023-04-21 20:23:27.877493684 +0200,
 @transactions=
  [{:from=>"Dutchgrown",
    :to=>"Vincent",
    :what=>"Tulip Bloemendaal Sunset",
    :qty=>10},
   {:from=>"Keukenhof", :to=>"Anne", :what=>"Tulip Semper Augustus", :qty=>7}],
 @transactions_count=2>
"==========================="
```

C. Ejercicio 3

Cada bloque contiene los siguientes elementos:

- **hash**: Es el identificador del bloque el cual está encriptado mediante SHA256 , establece un identificador único que permite diferenciar a unos bloques de otros y confirmar su validez, este puede cambiar al actualizar o cambiar la información de este.
- **index**: Se refiere al número de secuencia asignado al bloque dentro de la cadena de bloques, este número comienza con 0 en el bloque génesis y según se vayan creando se irán asignando.
- **nonce**: Es un número aleatorio que genera el propio protocolo que se utiliza para evitar que hash que antiguos puedan ser reutilizados.
- previous_hash: Hash del bloque previo al actual, formando así un encadenamiento.
- **timestamp**: Es la fecha de creación del bloque, normalmente representado en formato URT o GMT.
- **transactions**: Información de las transacciones realizadas en el bloque, incluyendo el remitente, destinatario, qué se ha enviado y la cantidad.

- **transactions_count**: Se refiere a la cantidad de transacciones que se incluyen en el bloque esta variable puede cambiar dependiendo de cuantas transacciones haya realizado el bloque.

D. Ejercicio 4

La función compute_hash_with_proof_of_work se encarga de crear un nuevo hash a partir de un nonce, que inicialmente es 0. Para ello, se inicia un bucle en el que se crean nuevos hashes con el nonce dado, y si el hash obtenido supera la dificultad dada, se considerará como válido y saldrá del bucle. Sin embargo, si el hash no es válido, se volverá a crear un hash nuevo a partir de otro nonce distinto.

Proof of work es un protocolo utilizado en algunos sistemas blockchain para validar transacciones y crear nuevos bloques en la cadena de bloques. El objetivo de esto es resolver el problema de la confianza y la seguridad en un entorno descentralizado sin necesidad de una autoridad central.
En este protocolo los nodos de la red de blockchain compiten para encontrar una solución criptográfica a un problema complejo, llamado problema de Prueba de Trabajo. Este problema es difícil de resolver, pero fácil de verificar una vez que se encuentra la solución. La solución se encuentra mediante la realización de cálculos complejos utilizando una gran cantidad de energía computacional.

El primer nodo en encontrar la solución al problema es recompensado con nuevas criptomonedas y, a cambio, agrega un nuevo bloque a la cadena de bloques, validando así las transacciones contenidas en el bloque. El problema de este sistema es que consume mucha energía y requiere una gran cantidad de recursos computacionales.

Otro concepto importante a la hora de entender la función compute_hash_with_proof_of_work es la "dificultad", la dificultad en un bloque de blockchain indica lo difícil que es resolver un complejo rompecabezas criptográfico de "prueba de trabajo" requerido para agregar un nuevo bloque a la cadena de bloques.

La dificultad de minar nuevas unidades aumenta o disminuye con el tiempo, dependiendo del número de mineros de la red. A dificultad más alta la red es más segura y requiere más esfuerzo computacional agregar nuevos bloques, una dificultad más baja indica que la red es menos segura, consta de una mayor velocidad de procesamiento y reduce costos de energía.

En nuestro caso, la dificultad dada es que el hash empiece con la secuencia “00”, pero puede incrementarse la complejidad modificando esta secuencia.

E. Ejercicio 5

Link al repositorio original -> https://github.com/apradillap/simple-blockchain-in-ruby
Link al repositorio -> https://github.com/augusMessina/simple-blockchain-in-ruby

En el programa original, los datos para las transacciones se recogen a través de la consola, de la misma forma, la función para crear nuevos bloques se ejecuta tras ordenarlo por la consola. Solo se puede visualizar e interactuar con la blockchain desde el propio ordenador en el que se está ejecutando el código.

Como ampliación de la práctica se nos ocurrió permitir que varios ordenadores cliente se comuniquen con un servidor para interactuar con la blockchain. Además, en el código original hay 1 única función para añadir bloques, en la que se piden las transacciones por consola y se crea el bloque. Sin embargo, nosotros separamos el proceso en 2 acciones distintas. Por una parte, se añaden transacciones y se guardan en un array llamado "pending_transactions", y por otra parte, se crean bloques y en estos se almacenan las transacciones que han ido añadiendo. Durante este proceso, se añade de forma adicional una nueva transacción de parte del propio blockchain, y con el usuario que ha creado el bloque como destinatario. Esta acción tiene como objetivo simular la actividad de minar bloques, y los usuarios que crean los bloques son los mineros, que son recompensados cada vez que logran crear un bloque.

Para llevar a cabo la funcionalidad descrita, levantamos un servidor web usando Ruby, con ayuda de Sinatra, que es una "gema" de Ruby. Las gemas son módulos que ofrecen una funcionalidad en concreto, y pueden encontrarse en Ruby Gems. Se pueden descargar estos módulos definiendo un archivo Gemfile, con el siguiente aspecto:

```
2
3     source "https://rubygems.org"
4
5     gem 'sinatra'
6
7     gem 'thin'
```

Con "source" se especifica la fuente de donde se obtendrán las gemas, y con "gem" se indica la gema que se utilizará. Además de Sinatra, también descargamos Thin, que es necesaria para el correcto funcionamiento de la anterior.

Para descargar estos paquetes es necesario descargarse bundle, que es una herramienta de desarrollo de Ruby que nos permite descargar e interactuar con las Ruby Gems. Puede descargarse con el comando:

sudo gem install bundler

Tras la instalación con la herramienta, podemos dirigirnos al directorio donde se encuentra nuestro programa y ejecutar:

sudo bundle install

Este comando instalará todas las gemas especificadas en el Gemfile, y creará un archivo Gemfile.lock, donde se especifican dependencias y versiones de las gemas. Los módulos descargados se instalarán por defecto en el directorio /var/lib/gems, y para ejecutar nuestro programa y que este pueda interactuar con las gemas, utilizaremos el comando:

sudo bundle exec ruby blockchain.rb

Como ya hemos mencionado, en nuestro proyecto empleamos la gema Sinatra. Esta gema permite levantar un servidor web capaz de manejar solicitudes HTTP de manera rápida y eficaz. Si hemos añadido la línea *require 'sinatra'* a nuestro archivo, podremos tener acceso a sus métodos y funciones, y para crear un endpoint, una URL mediante la cual interactuar con el servidor mandando peticiones, simplemente tenemos que definir el siguiente código en nuestro archivo:

```
get '/endpoint' do

   **funcionalidad**

end
```

En este caso estamos creando un endpoint que responde a peticiones que emplean el método GET. Existen varios métodos posibles además de este, los principales son:

- GET: el cliente no manda información al servidor, sino que es el servidor el que envía la información.
- POST: el cliente envía información nueva al servidor y este trabaja con ella.
- PUT: el cliente envía datos con el fin de modificar información que ya estaba en el servidor.
- DELETE: el cliente ordena el borrado de algún tipo de información en el servidor.

Tanto en el POST como en el PUT o el DELETE, el servidor puede o no enviar información de vuelta al cliente.

En el servidor que hemos levantado, definimos 4 endpoints distintos:

- **/chain**: este endpoint de tipo GET devuelve un JSON con toda la cadena del blockchain, especificando los detalles de cada bloque. Para ello, utilizamos la

variable LEDGER, que es un array definido en el código original, y que almacena las referencias de todos los bloques en orden. Usando el método map() propio de los arrays, transformamos cada referencia del array en un hash con la información del bloque, y este nuevo array es el que se devuelve.

- **/transactions**: en este endpoint, también de tipo GET, se devuelve un JSON con el array de transacciones pendientes.

- **/newTransaction**: este endpoint es de tipo POST. En él, el cliente enviará un JSON con la siguiente información: remitente, destinatario, divisa y cantidad. Si el JSON recibido por el servidor no cumple con las especificaciones, se enviará de vuelta un *status 400*, indicando que la petición está mal formada. Por otra parte, si la petición es correcta, se hará un push en el array pending_transactions con la información enviada.

- **/mine**: por último, mediante este endpoint, también de tipo POST, el cliente es capaz de minar un nuevo bloque. Para ello, primero se comprueba si existen transacciones pendientes. En caso negativo, se devolverá un *status 503*, que indica que el servicio no está disponible en estos momentos. Sin embargo, en el caso de que sí haya transacciones con las que trabajar, se ordenará la creación de un nuevo bloque, pasando como parámetros las transacciones pendientes y la IP del cliente minero. El bloque se creará siguiendo el procedimiento original, llamando al método next() de la clase Block, que crea un nuevo bloque con un index posterior al del bloque anterior, e indicando el hash de este bloque previo. La diferencia que hemos añadido es que a la hora de calcular el hash y el nonce, es decir, cuando se realiza el Proof or work, si se completa la prueba exitosamente, se añade al array de transacciones una nueva transacción con la IP del minero como destinatario, simulando de esta forma las recompensas que obtienen los mineros por crear nuevos bloques.

En el caso de que queramos permitir el acceso a nuestro servidor desde otras direcciones IP de la misma red, tendremos que añadir además la siguiente línea en alguna parte de nuestro código:

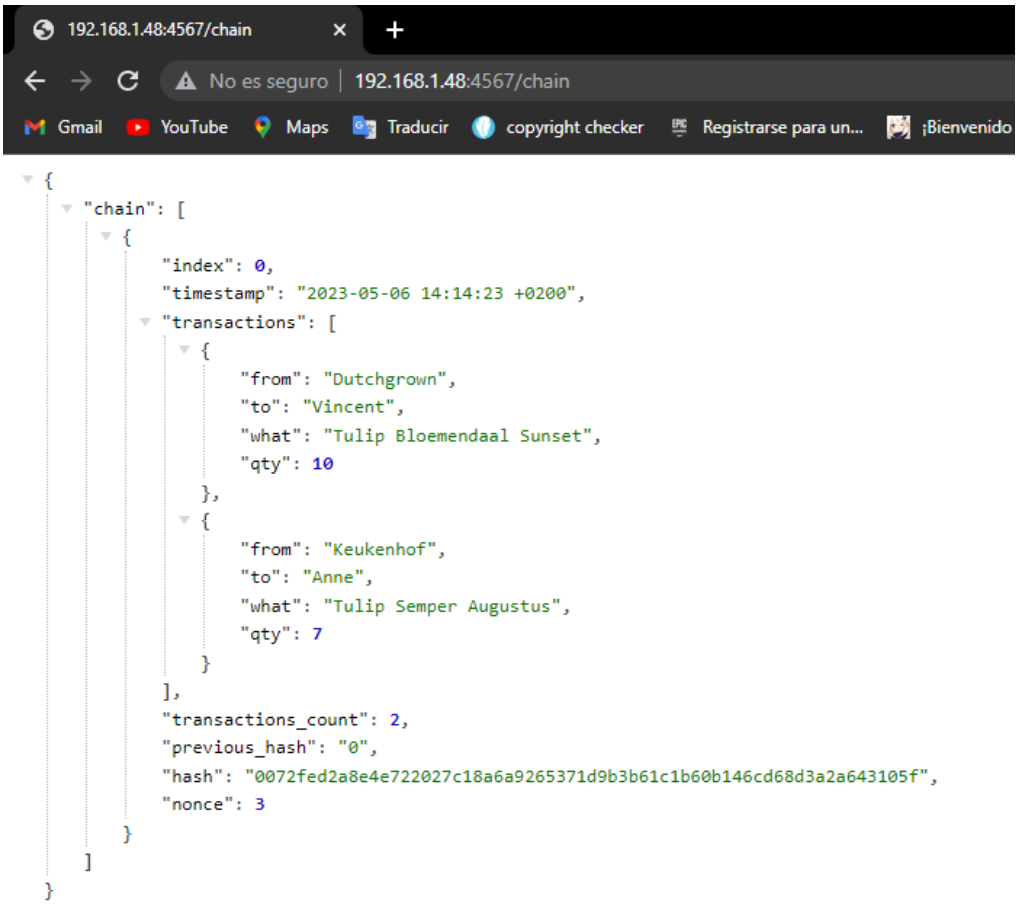
```
set :bind, 'tu IP'
```

De esta forma, le decimos a Sinatra que la IP del servidor mediante la cual los clientes enviarán sus peticiones, es la IP privada de nuestro ordenador. En caso de no poner esta línea, se tomará como IP 127.0.0.1, que nos permitirá trabajar de forma local, pero se podrá acceder desde una IP distinta.

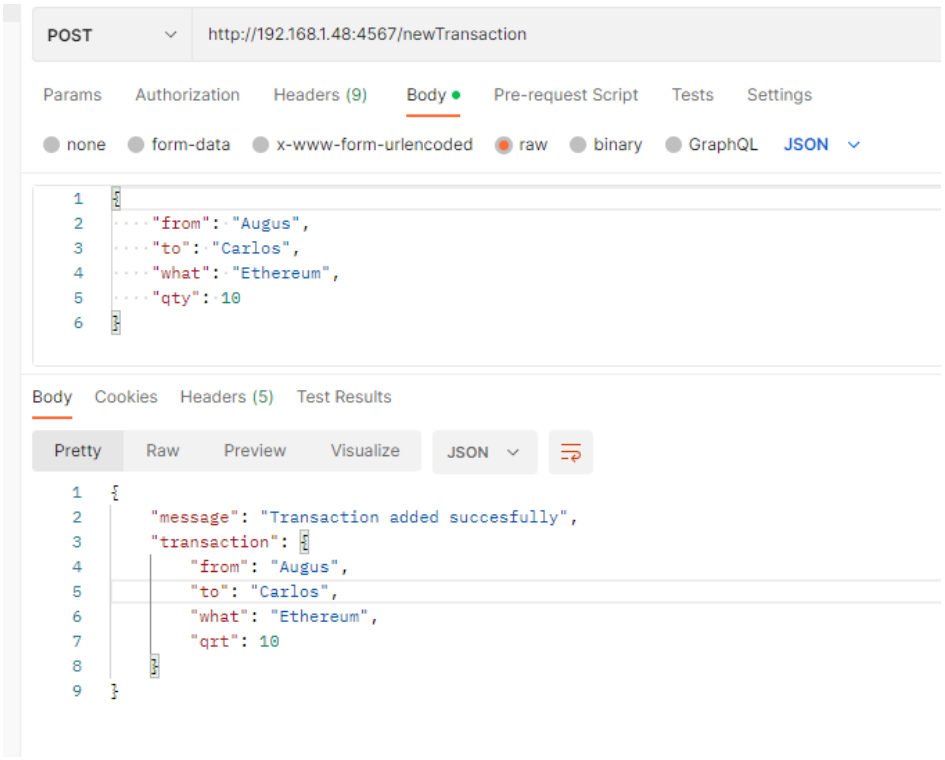
Con esto ya quedaría definido nuestro servidor web, si utilizamos el comando de ejecución de nuestro programa, el servidor quedará a la escucha de peticiones.

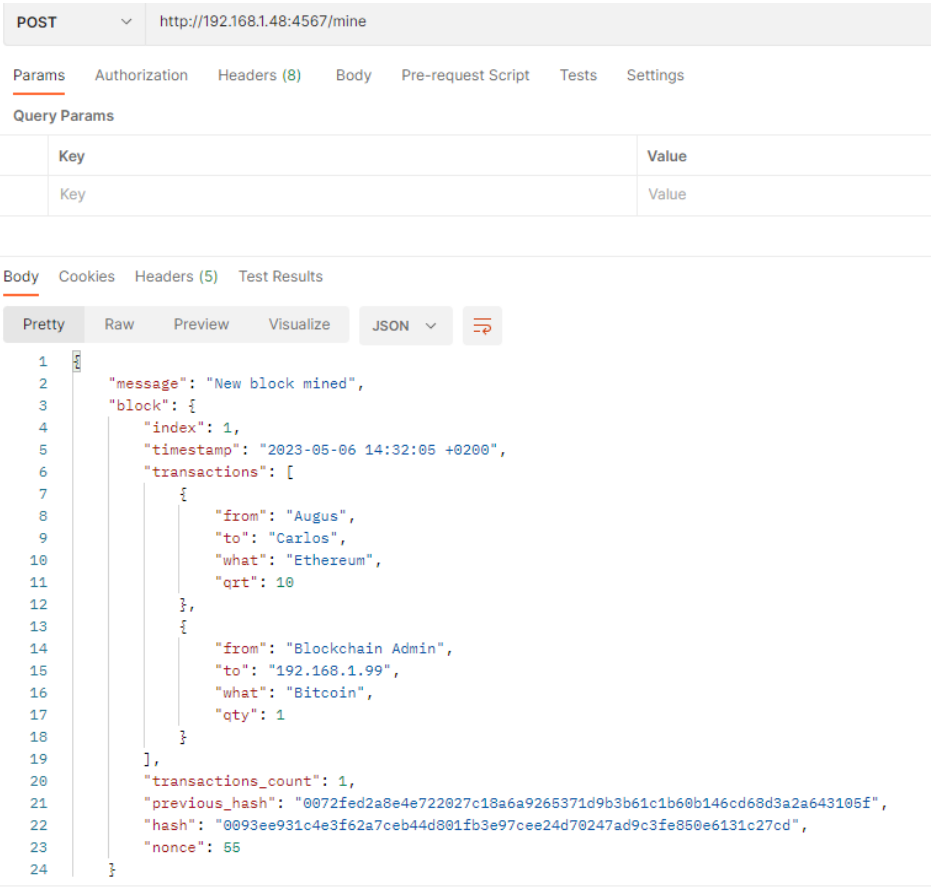
```
augusto@augusto-MSI:~/Documents/GitHub/simple-blockchain-in-ruby$ sudo bundle exec ruby blockchain.rb
== Sinatra (v3.0.6) has taken the stage on 4567 for development with backup from Thin
2023-05-06 14:12:17 +0200 Thin web server (v1.8.2 codename Ruby Razor)
2023-05-06 14:12:17 +0200 Maximum connections set to 1024
2023-05-06 14:12:17 +0200 Listening on localhost:4567, CTRL+C to stop
```

Para la demostración, realizamos las peticiones desde dos computadoras, una con sistema operativo de Windows y otra con Ubuntu.



Al principio, la cadena únicamente incluye el bloque génesis, pero podemos añadir nuevas transacciones y bloques usando los endpoints /newTransaction y /mine.


POST
http://192.168.1.48:4567/newTransaction
Params
Authorization
Headers (9)
Body
Pre-request Script
Tests
Settings
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
{
"from": "Augus",
"to": "Carlos",
"what": "Ethereum",
"qty": 10
}
Body
Cookies
Headers (5)
Test Results
Pretty
Raw
Preview
Visualize
JSON
{
"message": "Transaction added succesfully",
"transaction": {
"from": "Augus",
"to": "Carlos",
"what": "Ethereum",
"qrt": 10
}
}


POST
http://192.168.1.48:4567/mine
Params
Authorization
Headers (8)
Body
Pre-request Script
Tests
Settings
Query Params
Key
Value
Key
Value
Body
Cookies
Headers (5)
Test Results
Pretty
Raw
Preview
Visualize
JSON
{
"message": "New block mined",
"block": {
"index": 1,
"timestamp": "2023-05-06 14:32:05 +0200",
"transactions": [
{
"from": "Augus",
"to": "Carlos",
"what": "Ethereum",
"qrt": 10
},
{
"from": "Blockchain Admin",
"to": "192.168.1.99",
"what": "Bitcoin",
"qty": 1
}
],
"transactions_count": 1,
"previous_hash": "0072fed2a8e4e722027c18a6a9265371d9b3b61c1b60b146cd68d3a2a643105f",
"hash": "0093ee931c4e3f62a7ceb44d801fb3e97cee24d70247ad9c3fe850e6131c27cd",
"nonce": 55
}

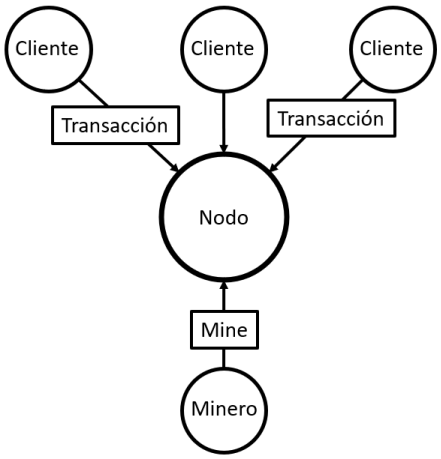
Tras minar bloques desde 2 clientes distintos, podemos ver que la cadena tiene el siguiente aspecto:


```
{
    "index": 1,
    "timestamp": "2023-05-06 14:32:05 +0200",
    "transactions": [
        {
            "from": "Augus",
            "to": "Carlos",
            "what": "Ethereum",
            "qrt": 10
        },
        {
            "from": "Blockchain Admin",
            "to": "192.168.1.99",
            "what": "Bitcoin",
            "qty": 1
        }
    ],
    "transactions_count": 1,
    "previous_hash": "0072fed2a8e4e722027c18a6a9265371d9b3b61c1b60b146cd68d3a2a643105f",
    "hash": "0093ee931c4e3f62a7ceb44d801fb3e97cee24d70247ad9c3fe850e6131c27cd",
    "nonce": 55
},
{
    "index": 2,
    "timestamp": "2023-05-06 14:37:14 +0200",
    "transactions": [
        {
            "from": "Arturo",
            "to": "Augus",
            "what": "Bitcoin",
            "qrt": 5
        },
        {
            "from": "Carlos",
            "to": "Augus",
            "what": "Ethereum",
            "qrt": 16
        },
        {
            "from": "Blockchain Admin",
            "to": "192.168.1.48",
            "what": "Bitcoin",
            "qty": 1
        }
    ],
    "transactions_count": 2,
    "previous_hash": "0093ee931c4e3f62a7ceb44d801fb3e97cee24d70247ad9c3fe850e6131c27cd",
    "hash": "0073e5a9b3f116087c4a60ff3ac1bbf8b41f5e28fe20186501d6a7f1a5eb07ea",
    "nonce": 10
}
]
```

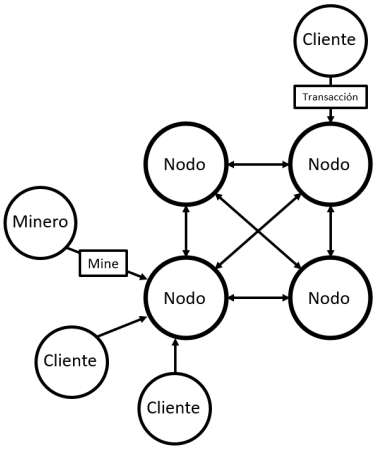
Como podemos ver, los bloques se añaden correctamente, y las recompensas a los mineros quedan grabadas con la IP de cada cliente como destinatario.

Sabemos que esta implementación no es del todo óptima debido a que sigue un modelo cliente-servidor centralizado. Para poder corregir este modelo y volverlo descentralizado deberemos levantar varios nodos, en lugar de solo 1, e implementar un algoritmo de consenso. Este algoritmo permite que todos los nodos se pongan de acuerdo para realizar acciones con una cadena de bloques.

De una forma más visual, pasaríamos de este modelo:

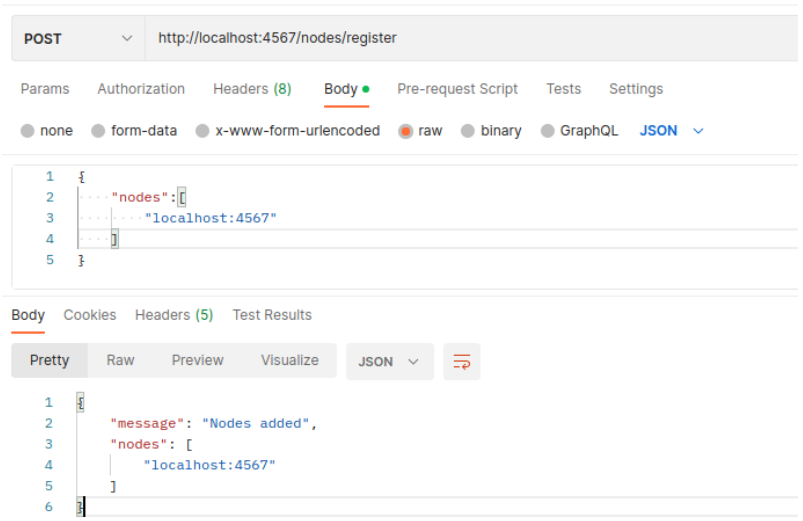


Al siguiente:

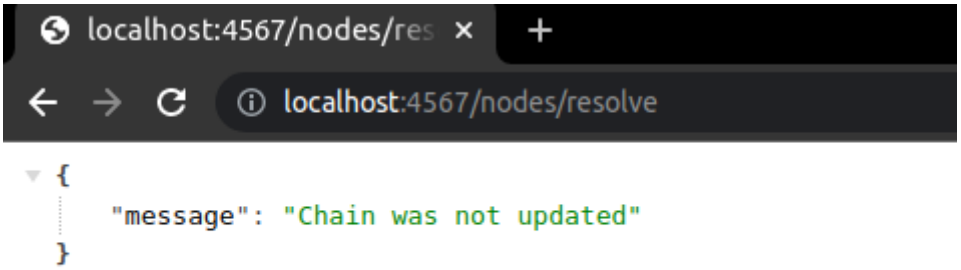


Para ello, primero crearemos un nuevo array llamado “nodes” en el que guardaremos una lista de todos los nodos vecinos que tenemos, es decir, las IPs de estos. Seguidamente, crearemos 2 endpoints adicionales con los que interactuar con esta nueva variable:

- /nodes/register -> Este endpoint sería un POST que permite introducir direcciones IPs de nodos al sistema. Se envía un array, donde se especifican las direcciones de los nodos del sistema (eg: [“IP1:PORT1”, “IP2:PORT2”])



- /nodes/resolve -> Este endpoint sería un GET que llamará a la función “resolve_nodes”. Esta función recorrerá el array *nodes*, e irá llamando al método */chain* de cada uno. Con la cadena recibida de cada nodo, se comparará la longitud de estas con la cadena del propio nodo. Si alguna de estas cadenas es más larga, se considerará que está más actualizada, por lo que se sustituirá la cadena del nodo por la recibida.



Como el nodo especificado como vecino ha sido simplemente el propio nodo, la cadena no ha sido actualizada.

De esta forma, cada vez que realicemos una petición al endpoint /nodes/resolve de un nodo, este realizará peticiones al resto de nodos en busca de una cadena más actualizada. Además, para hacer que el sistema sea más eficiente, podemos llamar también a la función *resolve_nodes* cuando vayamos a minar un nuevo bloque, así nos aseguramos de minar en una cadena válida.

IV. Conclusión

En esta práctica hemos aprendido el funcionamiento de blockchain, en que se basa en una base de datos distribuida y descentralizada, donde la información se registra en bloques que están conectados entre sí mediante algoritmos criptográficos. En cada bloque se almacenan transacciones verificadas y validadas por toda la red. Blockchain ofrece numerosas ventajas como transparencia, inmutabilidad y seguridad.

También hemos aprendido a crear un sistema blockchain que permite la conexión de distintos nodos a un servidor, esto funciona a través de un servidor web. Sabemos que una red blockchain no consta de una arquitectura centralizada pero hemos tratado de realizar una simulación de cómo funciona una red blockchain básica.

V. Bibliografía

-Installing Ruby
-¿Qué es el bloque génesis de Bitcoin?
-¿Qué es la tecnología Blockchain? - IBM Blockchain | IBM
-Qué son los bloques en la tecnología blockchain - Dinero y Trabajo
-What Is Proof of Work (PoW) in Blockchain?
-¿Qué es Prueba de trabajo / Proof of Work (PoW)?
-¿Qué significa la dificultad de minado? — Bitpanda Academy
-Usar Bundler para instalar Ruby gems
-Learn Blockchains by Building One | HackerNoon