UNIVERSIDAD NEBRIJA

Escuela Politécnica Superior

Diego Contreras, Diego González, Javier Zorrilla
Asignatura: Sistemas Distribuidos
Profesor: Andrés Bravo Montes

PRÁCTICA 4

UNIVERSIDAD NEBRIJA

1. Introducción:

Esta actividad se basa en el trabajo con un repositorio de Adrián Pradilla sobre *Simple blockchain in ruby*, es decir, entender cómo funciona un programa simple de cadenas de bloques empleando el lenguaje de programación de Ruby. Para ello, se ha empleado el sistema operativo de Ubuntu y el IDE de Visual Studio Code. Antes de ejecutar el correspondiente programa, se ha instalado *ruby* por comandos en la terminal de Ubuntu.

Primeramente, se deberá entender el código del repositorio provisto en el enunciado mediante el análisis del funcionamiento de cada archivo por separado para, posteriormente, entender el programa global de los tres archivos juntos.

Una vez entendido el código, se deberán de responder a una serie de cuestiones básicas sobre las cadenas de bloques o *blockchain*: qué es el bloque génesis, qué elementos tienen cada bloque, qué es *Proof of Work* (PoW) y qué realiza la función *compute_hash_with_proof_of_work* en el archivo de *block.rb*.

Finalmente, se modificará el código del repositorio para realizar una conexión entre cliente y servidor para mejorar la escalabilidad, ya que permitirá que los nodos de los clientes puedan realizar operaciones sin que sea necesario el procesamiento ni la validación de la transacción de cada una de las realizadas. Además, se ha implementado encriptación para aumentar la seguridad de la cadena de bloques con las distintas transacciones realizadas por el usuario.

2. *Blockchain con ruby*:

a) Ejecución del blockchain:

Para ejecutar la blockchain del repositorio: https://github.com/apradillap/simple-blockchain-in-ruby, se utiliza el comando ***ruby blockchain.rb***:

```
javier@javier-ASUS-TUF-Gaming-F15-FX506LH-FX506LH:~/Escritorio/Practica4_SistemasDistribuidos/simple-blockchain-in-ruby-master$ ruby blockchain.rb
========================

Welcome to Simple Blockchain In Ruby !

This program was created by Anthony Amar for and educationnal purpose

Wait for the genesis (the first block of the blockchain)

.........

========================
#<Block:0x0000563fe740fc18
 @hash="00c573e268e6c024946759a4b1360ecc1e77f7af19ff03fc85221664510b3a0f",
 @index=0,
 @nonce=178,
 @previous_hash="0",
 @timestamp=2023-05-04 10:04:01.147169829 +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>
"========================"
```

Primer bloque creado, bloque génesis

Al ejecutar este programa, se crea el primer bloque de la cadena de bloques (bloque génesis) automáticamente y, después, se le pregunta al usuario si desea añadir otra transacción, a lo que el usuario decidirá si añade otra o no.

b) Análisis del funcionamiento:

Este repositorio, previamente citado, se divide en tres archivos principalmente. Por lo tanto, se explicará el funcionamiento de cada archivo individualmente y, posteriormente, el desempeño global, es decir, teniendo en cuenta los tres archivos.

- ***block.rb*:** La clase *Block* es la principal del blockchain.

Los atributos de cada bloque vienen definidos por *attr_reader* que define un método de acceso getter para cada uno de ellos. Estos son un índice, un timestamp (denota la hora y la fecha en la que ocurrió la transacción), una lista de transacciones, un contador de transacciones, una tabla hash anterior del bloque, un nonce (número aleatorio usado una sola vez para la autenticación de transferencia de datos entre dos o más pares) y una tabla hash del bloque. Entonces, los valores de estos atributos sólo se pueden leer, no se pueden modificar directamente desde fuera de la clase.

```
attr_reader :index, :timestamp, :transactions,
            :transactions_count, :previous_hash,
            :nonce, :hash
```

Atributos de la clase Block

El método *initialize* es el constructor de la clase y es llamado cuando se crea una nueva instancia de la clase. Toma como parámetros el índice, la lista de transacciones y el hash anterior del bloque. Inicializa los atributos del bloque con valores dados, entre los que destacan el *timestamp* con la fecha y hora en la que se realiza la transacción; el contador de transacciones que, como su propio nombre indica, almacena cuántas transacciones se han hecho y; el *nonce* y *hash*, que se inicializa como *compute_hash_with_proof_of_work*.

```
def initialize(index, transactions, previous_hash)
  @index              = index
  @timestamp          = Time.now
  @transactions       = transactions
  @transactions_count  = transactions.size
  @previous_hash      = previous_hash
  @nonce, @hash       = compute_hash_with_proof_of_work
end
```

Método 'initialize'

El método *compute_hash_with_proof_of_work* es el algoritmo de prueba de trabajo que se utiliza para calcular la tabla hash del bloque. Toma como parámetro la dificultad que, por defecto es "00", que se usa para establecer el número de ceros iniciales que debe tener la tabla hash calculada para que se considere válida. Se emplea un *loop* para incrementar el nonce hasta que se encuentre un hash válido, además del requerimiento de los ceros al principio.

```
def compute_hash_with_proof_of_work(difficulty="00")
  nonce = 0
  loop do
    hash = calc_hash_with_nonce(nonce)
    if hash.start_with?(difficulty)
      return [nonce, hash]
    else
      nonce +=1
    end
  end
end
```

Método 'compute_hash_with_proof_of_work'

El método *calc_hash_with_nonce* se emplea para calcular el hash del bloque utilizando un *nonce* dado y mediante la función de hash *Digest::SHA256*.

```ruby
def calc_hash_with_nonce(nonce=0)
  sha = Digest::SHA256.new
  sha.update( nonce.to_s +
              @index.to_s +
              @timestamp.to_s +
              @transactions.to_s +
              @transactions_count.to_s +
              @previous_hash )
  sha.hexdigest
end
```

Método 'calc_hash_with_nonce'

El método *self.first* se usa para crear el primer bloque de la cadena, el llamado bloque génesis. Toma como parámetro una lista de transacciones y devuelve una nueva instancia de la clase *Block* con el índice cero, las transacciones dadas y el hash anterior como cero.

```ruby
def self.first( *transactions )    # Create genesis block
  ## Uses index zero (0) and arbitrary previous_hash ("0")
  Block.new( 0, transactions, "0" )
end
```

Método 'self.first'

El método *self.next* se usa para crear el siguiente bloque de la cadena. Toma como parámetros el bloque anterior y una lista de transacciones para el nuevo bloque. Devuelve una nueva instancia de la clase *Block* con el índice del bloque anterior más uno, el hash anterior del bloque anterior y las transacciones dadas.

```ruby
def self.next( previous, transactions )
  Block.new( previous.index+1, transactions, previous.hash )
end
```

Método 'self.next'

- ***blockchain.rb***: Este programa crea una cadena de bloques simple, permitiendo al usuario agregar nuevas transacciones a la cadena.

Primeramente, se define una constante *LEDGER* vacía, que contendrá los bloques del blockchain. Luego, se definen una serie de métodos:

- *create_first_block*: Crea el primer bloque de la cadena, conocido como génesis, y lo almacena en *LEDGER*. También crea dos transacciones ficticias y las agrega al primer bloque. Luego, llama al método *add_block*.

```ruby
def create_first_block
    i = 0
    instance_variable_set( "@b#{i}",
                                    Block.first(
                                        { from: "Dutchgrown", to: "Vincent", what: "Tulip Bloemendaal Sunset", qty: 10 },
                                        { from: "Keukenhof", to: "Anne", what: "Tulip Semper Augustus", qty: 7 } )
                                )
    LEDGER << @b0
    pp @b0
    p "=========================="
    add_block
end
```

Método 'create_first_block'

- *add_block*: Es un bucle infinito que solicita al usuario información sobre una transacción y crea un nuevo bloque para almacenarla. Este nuevo bloque se agrega a *LEDGER* y se imprime por terminal. Luego, se solicita al usuario que ingrese información sobre otra transacción y así sucesivamente.

```ruby
def add_block
    i = 1
    loop do
        instance_variable_set("@b#{i}", Block.next( (instance_variable_get("@b#{i-1}")), get_transactions_data))
        LEDGER << instance_variable_get("@b#{i}")
        p "============================"
        pp instance_variable_get("@b#{i}")
        p "============================"
        i += 1
    end
end
```

Método 'add_block'

- *launcher*: Es el punto de entrada del programa e imprime un mensaje de bienvenida y, luego, llama al método de crear el primer bloque.

```ruby
def add_block
    i = 1
    loop do
        instance_variable_set("@b#{i}", Block.next( (instance_variable_get("@b#{i-1}")), get_transactions_data))
        LEDGER << instance_variable_get("@b#{i}")
        p "============================"
        pp instance_variable_get("@b#{i}")
        p "============================"
        i += 1
    end
end

def launcher
    puts "=========================="
    puts ""
    puts "Welcome to Simple Blockchain In Ruby !"
    puts ""
    sleep 1.5
    puts "This program was created by Anthony Amar for and educationnal purpose"
    puts ""
    sleep 1.5
    puts "Wait for the genesis (the first block of the blockchain)"
    puts ""
    for i in 1..10
        print "."
        sleep 0.5
        break if i == 10
    end
    puts ""
    puts ""
    puts "=========================="
    create_first_block
end
```

Método 'launcher'

- ***transaction.rb***:

La función *get_transactions_data* permite al usuario ingresar información de transacciones, para agregar a un nuevo bloque en la cadena de bloques. Primeramente, crea una matriz vacía *transactions_block* y una *blank_transaction* (transacción vacía con todas sus propiedades vacías, es decir, el remitente, el destinatario, el mensaje y la cantidad que se va a transferir). Luego, comienza un ciclo infinito, que pide al usuario que ingrese información sobre una nueva transacción. El usuario deberá ingresar su nombre, el mensaje que desea enviar, la cantidad que quiere enviar y el destinatario de la transacción. Después de recopilar toda la información, se crea un nuevo hash que contiene la información de la transacción y lo agrega a la matriz de transactions_block.

Posteriormente, se le pregunta al usuario si desea agregar otra transacción a este bloque. Si el usuario responde una “y”, el usuario decide realizar otra transacción y si, por el contrario, responde una “n”, no desea realizar ninguna transacción más y devuelve la matriz de *transactions_block* con todas las transacciones ingresadas en este bloque y sale del loop.

```ruby
def get_transactions_data

    transactions_block ||= []
    blank_transaction = Hash[from: "", to: "",
                             what: "", qty: ""]
    loop do
        puts ""
        puts "Enter your name for the new transaction"
        from = gets.chomp
        puts ""
        puts "What do you want to send ?"
        what = gets.chomp
        puts ""
        puts "How much quantity ?"
        qty  = gets.chomp
        puts ""
        puts "Who do you want to send it to ?"
        to   = gets.chomp

        transaction = Hash[from: "#{from}", to: "#{to}",
                           what: "#{what}", qty: "#{qty}"]
        transactions_block << transaction

        puts ""
        puts "Do you want to make another transaction for this block ? (Y/n)"
        new_transaction = gets.chomp.downcase

        if new_transaction == "y"
            self
        else
            return transactions_block
            break
        end
    end
end
```

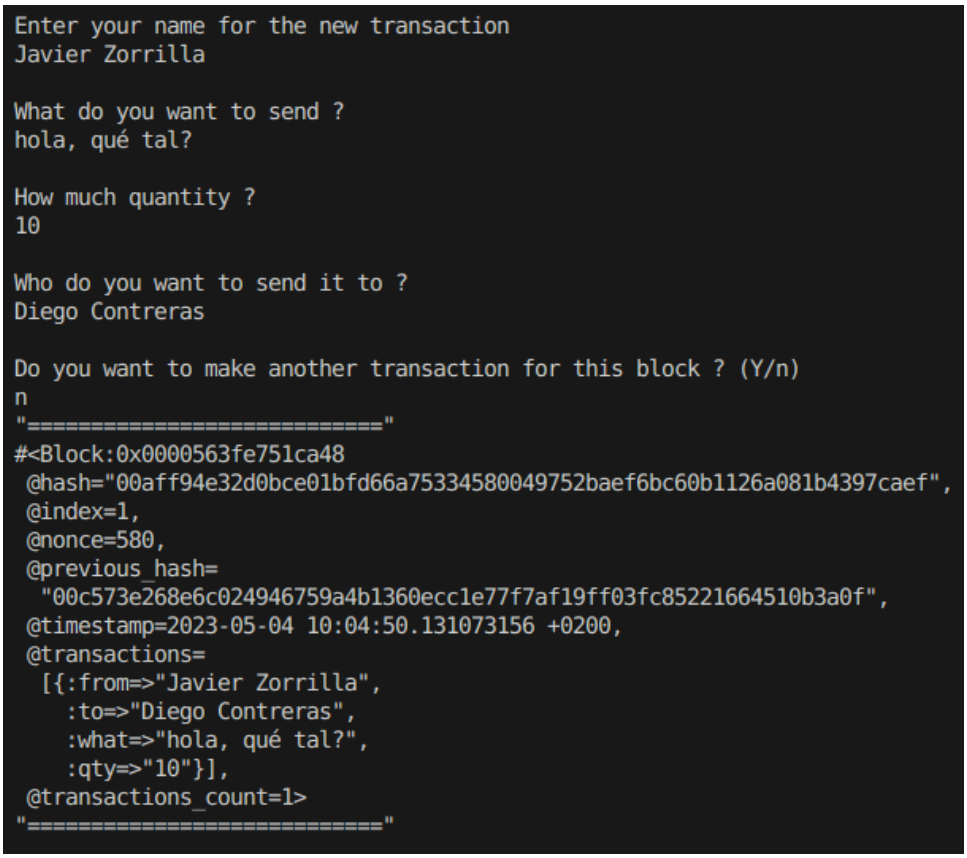
Único método de 'transaction.rb'

Los archivos trabajan juntos para permitir que el usuario envíe mensajes de una cantidad determinada a un receptor determinado en la cadena de bloques. Las transacciones realizadas por el usuario se añaden a la lista de transacciones pendientes. Por último, el nuevo bloque se agrega a la lista de bloques, permitiendo un correcto funcionamiento del sistema de cadena de bloques.

El proceso comienza con la creación de una instancia de la clase *Blockchain* que, inicialmente, no contiene bloques ni transacciones. Luego, se instancia *transaction* mediante una tabla hash, que contiene el remitente, el destinatario, el mensaje y la cantidad; para representar la transacción que el usuario desea realizar. Ésta tiene un remitente, un destinatario y una cantidad que se va a transferir determinados.

La creación del bloque y la adición a la cadena de bloques ocurren en todo momento, es decir, no hay número mínimo de transacciones pendientes establecido, por lo que no se garantiza que los bloques en la cadena de bloques contengan suficientes transacciones para hacer que el sistema sea eficiente y seguro.

Finalmente, el programa solicita al usuario que ingrese un mensaje de una cantidad determinada a un receptor cualquiera. Entonces, se crea una transacción que será agregada a la lista de transacciones pendientes. Luego, se agregará esta transacción a un nuevo bloque en la cadena de bloques.



Transacción realizada con éxito

Una vez que se ha creado el primer bloque, bloque génesis, se solicita al usuario que introduzca un nombre para la nueva transacción, el mensaje que quiere enviar, el destinatario y la cantidad.

Entonces, se tienen el bloque creado inicialmente y la nueva transacción realizada:

- Cada ***hash*** es correcto, puesto que comienza con dos ceros, gracias al algoritmo empleado *compute_hash_with_proof_of_work*, posteriormente explicado.

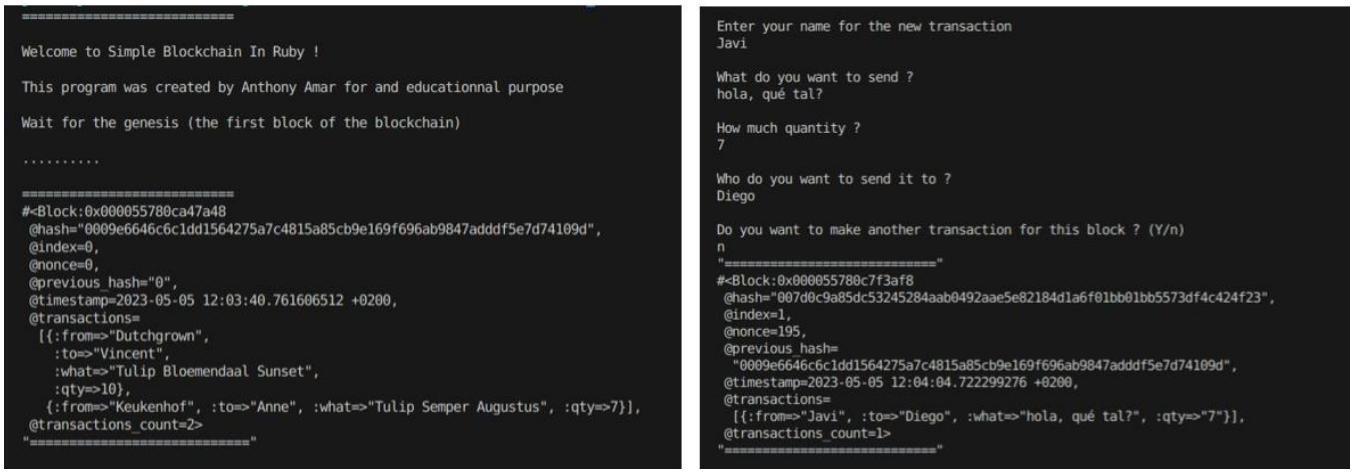
- Cada ***index*** es contiguo al anterior, el bloque génesis tiene índice de 0 y el siguiente de 1.

- Cada ***nonce*** representa un número aleatorio.

- El ***previous_hash*** del bloque génesis es 0, puesto que no es posible crear un bloque antes que él mismo. Sin embargo, en la primera transacción muestra el hash del bloque anterior.

- El ***timestamp*** en ambos bloques muestra la fecha y la hora en que han sido creados y, obviamente, son distintos, ya que se crean secuencialmente, uno detrás de otro.

- En ***transactions***, se tiene el remitente, el destinatario, el mensaje para enviar y la cantidad introducidas por el usuario.



Par de bloques de la cadena de bloques (Bloque génesis y nueva transacción creada)

3. Bloque génesis:

El **bloque génesis** representa el primer bloque de una cadena (*blockchain*) y es fundamental para la integridad y seguridad de la red. Es por ello que no tiene un bloque predecesor. Se crea al iniciar una red de criptomonedas o al realizar un *fork* de esta cadena. Suele ocurrir cuando hay novedades que pueden alterar el correcto funcionamiento de la red y, entonces, se opta por realizar el *fork*. Otro motivo puede ser porque la red se caiga como ocurrió con *terra luna*, la cual perdió la paridad con el dólar. Sin embargo, se hizo un *fork* de ésta que, actualmente tiene un precio de 1’15$, mientras que la original tuvo un precio de 0’000104$ (*terra classic*).

Este bloque contiene información crítica para el funcionamiento de la red, como fecha y hora de creación, número máximo de monedas que se pueden crear, el algoritmo usado para la validación de transacciones (PoW, PoS, PoA, PoST), el número de tokens emitidos, la versión del protocolo utilizado y demás configuraciones relevantes para la red. Una vez creado el génesis, la cadena se extiende mediante la creación de bloques cada cierto número de transacciones que se conectan entre sí.

La creación de este bloque es un proceso crítico porque cualquier error puede dañar toda la cadena de bloques. Entonces, se debe garantizar que el bloque sea creado de manera precisa y segura.

Cabe destacar que el bloque génesis es especial, puesto que es el punto de inicio de todas las transacciones que se realicen posteriormente en la cadena de bloques.

4. Elementos de cada bloque:

Un bloque de blockchain contiene varios elementos importantes que se usan en la validación y el registro de las transacciones en la red:

- **Identificador del bloque**: Cada bloque se identifica mediante un hash generado mediante el algoritmo de hash criptográfico *SHA256* en el encabezado del bloque. Este identificador es único de cada bloque. Entonces, cada bloque contiene el hash de su padre dentro de su propio encabezado.

  La secuencia de hash que vincula cada bloque con su padre crea una cadena que va hasta el primer bloque creado, es decir, el bloque génesis.

- **Datos de las transacciones**: Cada bloque contiene información sobre las transacciones que se han realizado en la red desde el último bloque. Esta información incluye el destinatario, el remitente, el mensaje y la cantidad a transferir de cada una de las transacciones realizadas hasta el momento.

- ***Timestamp***: Cada bloque contiene un *timestamp* que indica cuándo se creó el bloque en cuestión. Es importante llevar este registro correctamente en el tiempo para mantener en un orden cronológico adecuado las transacciones realizadas en el blockchain.

- **Validación**: Cada bloque de la cadena de bloques contiene una serie de transacciones y, antes de que el bloque se agregue a la cadena, se deben validar si estas transacciones son legítimas. Para validar cada una de las transacciones se deben presentar estas transacciones a la red para su debida validación. Los demás nodos de la red verifican que la solución sea correcta y que todas las demás transacciones se realicen correctamente.

  Este elemento de cada bloque es esencial para garantizar y mantener la seguridad y la integridad de la cadena de bloques. Cabe destacar que, gracias a la validación, la cadena de bloques se vuelve más resistente a ataques maliciosos y a la manipulación de datos.

- **Referencia al bloque anterior**: Cuando se agrega un nuevo bloque a la cadena de bloques, su referencia apunta al hash del bloque anterior en la cadena. Así, se crea un enlace criptográfico entre los bloques, y cualquier intento de modificar un bloque afectará al hash del bloque siguiente en la cadena. Es importante garantizar la integridad de la cadena y evitar la manipulación de datos, como también es clave en la validación.

5. Método “def compute_hash_with_proof_of_work”:

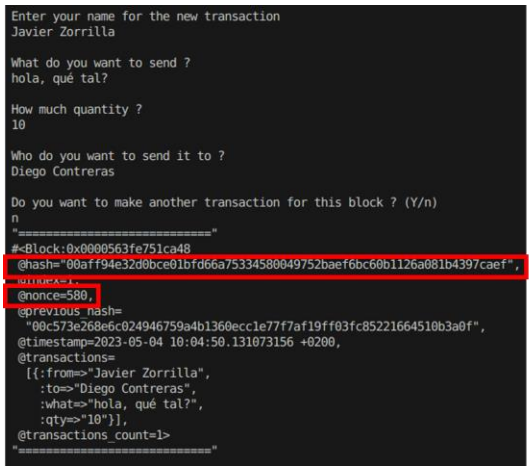
Proof of Work es un algoritmo basado en complejas operaciones matemáticas que deben ser realizadas por un equipo informático. Requiere de una elevada carga computacional y sirve para disuadir a atacantes maliciosos que deseen lanzar un ataque de denegación de servicio.

Se emplea para la confirmación de las transacciones y la generación de nuevos bloques, haciendo uso de la validación, previamente explicada. El problema a solucionar debe ser computacionalmente difícil, pero no demasiado complejo como para ralentizar la generación de bloques. Si fuera demasiado difícil, las transacciones se amontonarían y eso haría que la red fuera absurdamente lenta. Si los bloques no se pueden generar en el tiempo previsto, la red se congestionará y el coste de usarla (comisiones por transacciones) se disparará.

El método ***compute_hash_with_proof_of_work*** es una Implementación de *Proof of Work* que busca garantizar que los nodos de la red que validan las transacciones hayan dedicado una cantidad significativa de esfuerzo computacional para hallar una solución a un problema matemático difícil. Al hacerlo, se agregan nuevos bloques al blockchain.

Toma como parámetro la dificultad, que especifica cuántos ceros deben tener los primeros caracteres del hash de la transacción para que se considere válida. Este proceso de PoW comienza con un *nonce* (número aleatorio) igual a cero y se repite hasta que se encuentra un hash que cumpla con la dificultad indicada. En cada iteración, se calcula un hash para el *nonce* actual y se verifica si cumple con la dificultad, si no es así, se incrementa el *nonce* y se vuelve a intentar.

Cuando se encuentra el hash que cumple con la dificultad indicada, se devuelve una tupla de valores que contiene el *nonce* y el hash encontrado, y se agrega a la cadena de bloques como nuevo bloque.



'hash' y 'nonce' de las transacciones realizadas

En ambas transacciones, se puede comprobar que el hash comienza con la dificultad deseada de dos ceros (00…) y, además, el número aleatorio, *nonce*, se muestra correctamente.

6. *Fork* de la práctica con cambios:

Se ha realizado un *fork* de la práctica con las modificaciones llevadas a cabo, como se puede observar en el siguiente enlace al repositorio de GitHub:

https://github.com/Diegopower199/simple-blockchain-in-ruby

Para aumentar la seguridad de nuestra cadena de bloques se ha decidido implementar una encriptación con el objetivo de que si alguien se mete en el blockchain, no pueda coger información fácilmente. Además, se ha implementado una estructura cliente – servidor para mejorar la escalabilidad, al permitir que los nodos de los clientes puedan realizar operaciones sin que sea necesario el procesamiento ni la validación de la transacción de cada una de las que haga.

Se han añadido dos archivos nuevos para definir el funcionamiento del servidor y el cliente:

- ***server.rb***: Clase del servidor.

Primero, se debe requerir el módulo *socket* que se necesita para poder crear aplicaciones con la red. Se crea la variable *server* que inicializa el objeto *TCPServer* localmente, es decir, en el *localhost* y el puerto 3000. Además, se debe configurar *stdout* de manera sincronizada para enviar información inmediatamente al cliente y que no quede almacenada en el búfer.

Luego, se entra en un bucle infinito y se crea la variable *client*. Cada vez que se introduce un nuevo cliente se mete en un bucle nuevo, diferente al anterior. Gracias a *until client.eof?* se puede saber cuándo ha finalizado la conexión con el cliente, si este valor resulta verdadero. En caso contrario, este valor es falso y se continúa con la comunicación. Si esta comunicación no ha terminado, es decir, el cliente sigue activo, se hace un *get* del mensaje del cliente para almacenarlo en *msg*. Finalmente, se envía al cliente el mismo mensaje que ha enviado al servidor.

```ruby

require 'socket'

server = TCPServer.new('localhost', 3000)

$stdout.sync = true

loop do
    client = server.accept

    puts "#{client.eof?}"
    until client.eof?
        msg = client.gets

        client.write(msg)
        puts msg
        puts "#{client.eof?}"
    end

end
```

Clase Servidor

- ***client.rb***: Clase del cliente.

Primero, se debe requerir los módulos *socket* que se necesita para poder crear aplicaciones con la red y *./blockchain.rb* porque se quiere llamar desde este archivo a una función del fichero de blockchain.

A continuación, se lleva a cabo un manejo de excepciones con *begin* (try) y *rescue* (catch). Dentro del *begin*, se crea la variable *socket* y se inicializa con conexión TCP en el puerto 3000 y en el *localhost*, es decir, localmente. Después, se muestra el contenido de *socket*. Si la conexión en la variable *socket* se ha realizado correctamente, significa que se ha podido comunicar adecuadamente y se escribe un mensaje en el *socket* de salida y, luego, se muestra la respuesta del *socket* de entrada.

Por último, se llama a la función *launcher* que está definida en el archivo *blockchain.rb* y ahí se ejecutará el código que contenga dicho método.

Si se ha producido algún error en el bloque de *begin* se ejecutará el *rescue* y mostrará el error ocurrido. Justo después de que la conexión con el servidor haya sido realizada correctamente se debe cerrar el *socket*.

```ruby

require 'socket'
require './blockchain.rb'


    begin
        socket = TCPSocket.new('localhost', 3000)
        puts socket
        socket.write("Funciona bien la llamada al cliente\n")
        puts socket.gets

        launcher

    rescue
        puts "Se ha producido un error: #{error.message}"
    end
    socket.close
```

Clase Cliente

También se ha modificado el archivo ***transaction.rb***, en el que se debe requerir del módulo *openssl*, ya que es una librería con la que se puede encriptar y desencriptar de maneras diferentes. Dentro de este fichero, se tienen tres funciones distintas: *get_transactions_data, encrypt* y *decrypt.*

La función ***get_transactions_data*** permite al usuario ingresar distinta información acerca de las transacciones que se vayan realizando para agregarla a un nuevo bloque en la cadena de bloques. Primeramente, se crean dos matrices vacías, *transactions_block* y una *blank_transaction*, siendo la segunda de ellas una transacción vacía que engloba todas las propiedades vacías: el remitente, el destinatario, el mensaje y la cantidad que se va a transferir). A continuación, se crea una clave de encriptación aleatoria de 256 bits mediante el objeto *OpenSSL::Cipher* con el algoritmo de cifrado de AES-256-CBC y, esto mismo, también con el vector de inicialización.

```ruby
def get_transactions_data

    transactions_block ||= []
    blank_transaction = Hash[remitente: "", destinatario: "", what: "", qty: ""]
    loop do
        puts ""
        puts "Enter your name for the new transaction"
        remitente = gets.chomp
        puts ""
        puts "What do you want to send ?"
        what = gets.chomp
        puts ""
        puts "How much quantity ?"
        qty  = gets.chomp
        puts ""
        puts "Who do you want to send it to ?"
        destinatario = gets.chomp

        # Creamos una clave de encriptación
        key = OpenSSL::Cipher.new('AES-256-CBC').random_key

        # Creamos un vector de inicialización para la encriptación
        iv = OpenSSL::Cipher.new('AES-256-CBC').random_iv


        # Encriptamos los nombres del remitente y del destinatario
        encrypted_remitente = encrypt(remitente, key, iv)
        encrypted_destinatario = encrypt(destinatario, key, iv)
```

```ruby
        # Imprimimos los nombres encriptados y la clave y vector de inicialización
        puts "Remitente encriptado: #{encrypted_remitente}"
        puts "Destinatario encriptado: #{encrypted_destinatario}"
        puts "Clave de encriptación: #{key.unpack('H*').first}"
        puts "Vector de inicialización: #{iv.unpack('H*').first}"

        decrypted_remitente = decrypt(encrypted_remitente, key, iv)
        decrypted_destinatario = decrypt(encrypted_destinatario, key, iv)


        puts "Remitente desencriptado: #{decrypted_remitente}"
        puts "Destinatario desencriptado: #{decrypted_destinatario}"




        transaction = Hash[remitente: "#{encrypted_remitente}", destinatario: "#{encrypted_destinatario}",
                           what: "#{what}", qty: "#{qty}"]
        transactions_block << transaction

        puts ""
        puts "Do you want to make another transaction for this block ? (Y/n)"
        new_transaction = gets.chomp.downcase

        if new_transaction == "y"
            self
        else
            return transactions_block
            break
        end
    end
end
```

Función 'get_transactions_data'

Luego, se encriptan los nombres del remitente y el destinatario por seguridad, llamando a la función "**encrypt**", que toma tres parámetros (el nombre del remitente, la clave de encriptación creada aleatoriamente y el vector de inicialización). Dentro de esta función, se crea el objeto *cipher* a partir de la instancia de *OpenSSL::Cipher* (clase que proporciona *openssl* para el cifrado y descifrado simétrico. Permite una gran variedad de algoritmos de encriptación, aunque se ha elegido emplear AES (*Advanced Encryption Standard*) y, con ello, se crea un objeto que toma un par de parámetros (el tamaño de bits, en este caso, 256 bits, y el modo de operación de cifrado simétrico en bloque que se usa con AES).

El *cipher* es el responsable de encriptar y se establecen los valores de "key" y "iv" a los objetos de *cipher*. Finalmente, se crea la variable *encrypted* que se devolverá y los encriptados mediante las funciones siguientes: "cipher.update" (se le pasa el texto que se desea encriptar por parámetro) y "cipher.final". A continuación, se imprime esto para saber que se ha encriptado correctamente, aunque esto no debería mostrarse por terminal, pero es para asegurarse que funciona como se quiere.

```
3   def encrypt(texto_sin_encriptar, key, iv)
4       cipher = OpenSSL::Cipher::AES.new(256, :CBC)
5       cipher.encrypt
6       cipher.key = key
7       cipher.iv = iv
8       encrypted = cipher.update(texto_sin_encriptar) + cipher.final
9       return encrypted
10    end
```

Función 'encrypt'

Luego, se desencripta el remitente y el destinatario llamando a la función "**decrypt**", que se le pasa por parámetros el remitente o el destinatario encriptado, la *key* y el *iv*. Si no se tiene ni *key* ni *iv*, no hay forma de poder desencriptar la información. Dentro de esta función, se crea el objeto *cipher* a partir de la instancia de *OpenSSL::Cipher* (clase que proporciona *openssl* para el cifrado y descifrado simétrico) con dos parámetros, el tamaño de 256 bits y el modo de operación de cifrado simétrico en bloque (CBC).

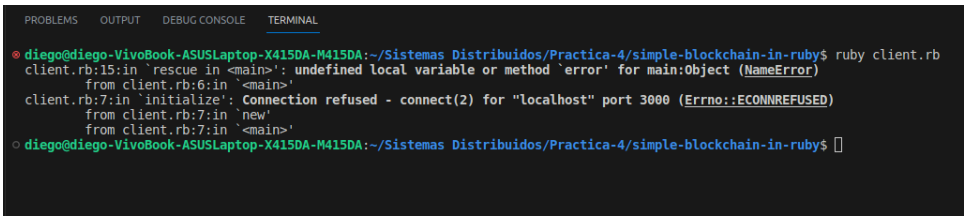
En este momento, el *cipher* es el encargado de desencriptar y se establecen los valores de "key" y "iv" a los objetos de *cipher*. Finalmente, se crea la variable *decrypted* que se devolverá y se desencripta usando las funciones: "cipher.update" (se le pasa el texto que se desea desencriptar por parámetro) y "cipher.final". Por último, se muestra por pantalla el remitente y el destinatario desencriptados.

Cabe destacar que la clave de encriptación y el vector de inicialización son importantes, ya que son necesarios para desencriptar el texto encriptado que, en nuestro caso, son el remitente y/o el destinatario.

```
12  def decrypt(texto_encriptado, key, iv)
13      cipher = OpenSSL::Cipher::AES.new(256, :CBC)
14      cipher.decrypt
15      cipher.key = key
16      cipher.iv = iv
17      decrypted = cipher.update(texto_encriptado) + cipher.final
18      return decrypted
19  end
```

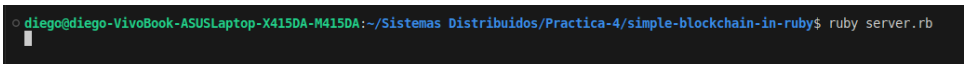
Función 'decrypt'

Si no se tiene una terminal que ejecute el servidor, resultará este error porque se carece de conexión.

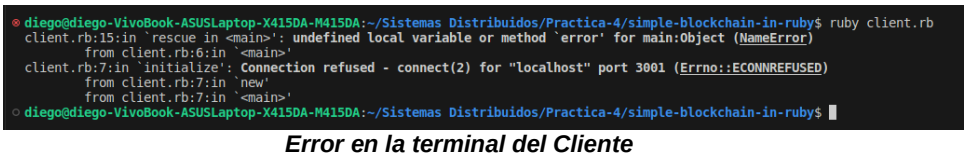


Error de conexión en la terminal del Cliente

Además, si el servidor está activo, pero el puerto es distinto en el cliente saldrá otro error.



Se inicia el Servidor desde su terminal

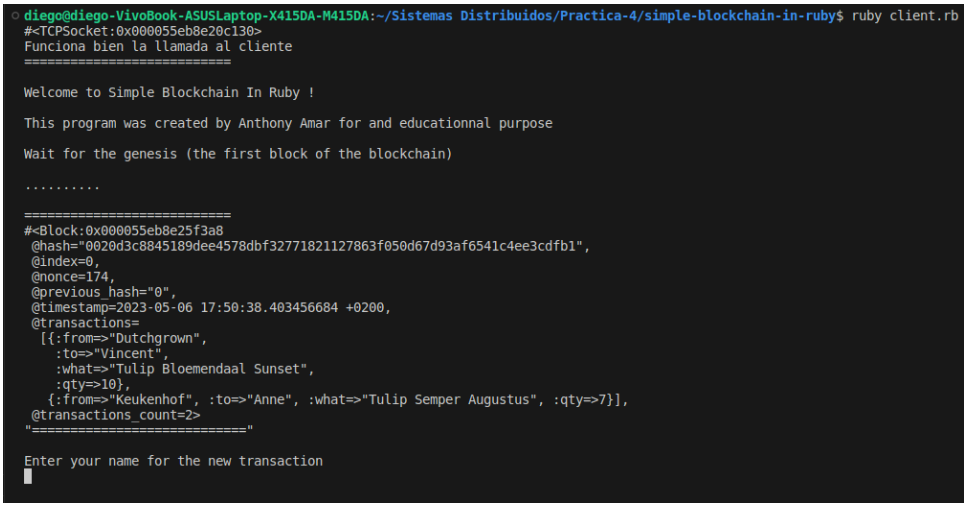


Error en la terminal del Cliente

Después de citar y evidenciar todos los errores posibles que pueden ocurrir en este programa y sus correspondientes soluciones, se verá el funcionamiento cuando no hay fallos de por medio.

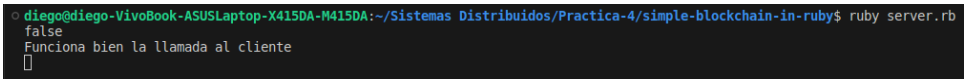
Se ejecutan el servidor y el cliente con dos terminales diferentes, una para cada uno.

Terminal del Servidor



Terminal del Cliente

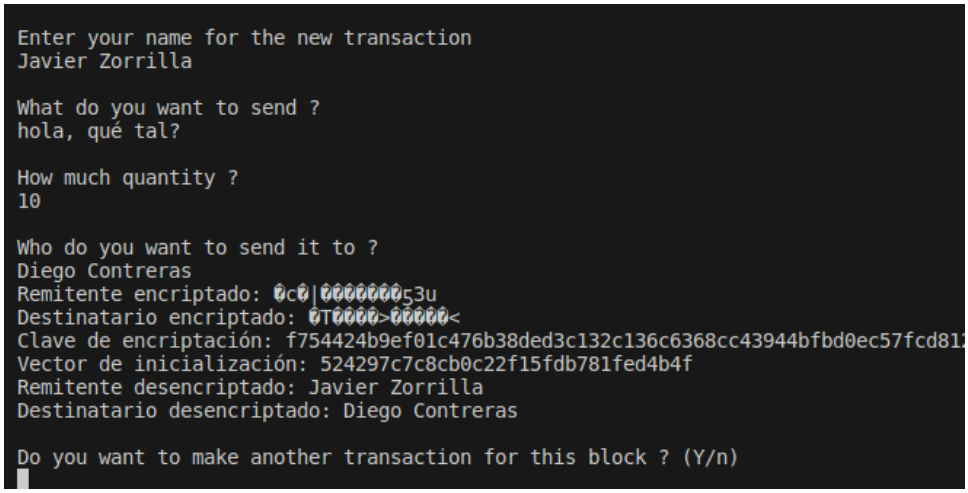
En la terminal del servidor, aparecen dos mensajes.



Terminal del Servidor

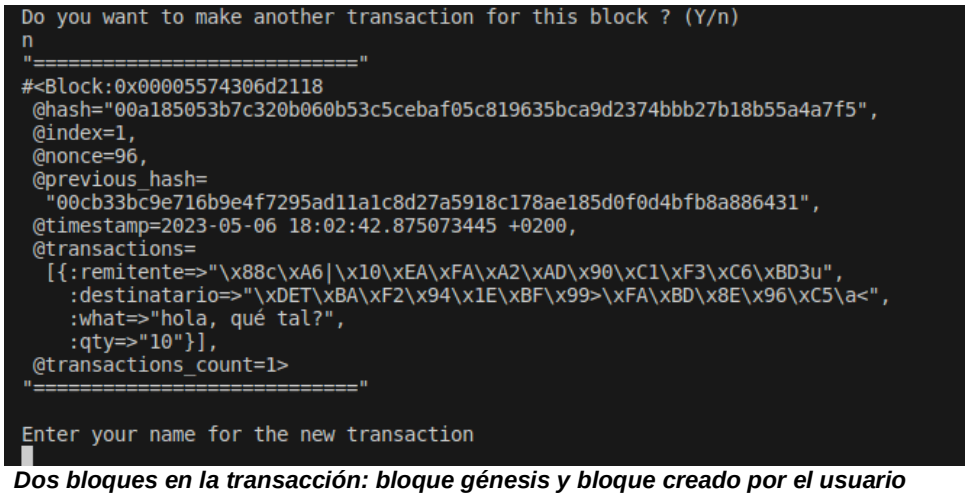
El primero de ellos, *false*, indica que la llamada del cliente aún está en proceso, y el segundo, es para saber que la comunicación se está realizando correctamente.

En la terminal del cliente, se introducen los valores necesarios para llevar a cabo una transacción y, se muestra por pantalla, el remitente y el destinatario encriptados, la clave de encriptación y el vector de inicialización para la correspondiente encriptación y, finalmente, el remitente y el destinatario desencriptados.



Cifrado y descifrado de los valores de cada transacción

A continuación, sale un mensaje que indica al usuario si desea realizar otra transacción. Si responde que sí, tendrá que introducir los datos requeridos en las distintas transacciones que quiera agregar a la cadena de bloques. Si la opción elegida es no seguir introduciendo transacciones en ese bloque, sale toda la información que se ha introducido previamente en ese bloque.

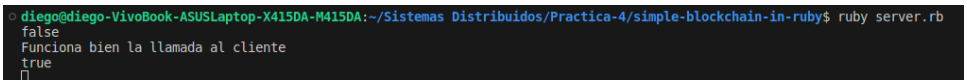


Dos bloques en la transacción: bloque génesis y bloque creado por el usuario

Si se para la ejecución del cliente en el servidor sale una nueva línea de código que significa que ya se ha finalizado la comunicación con el cliente.

Terminal Cliente finalizada

Se puede observar en la terminal del servidor que aparece el mensaje de *true*, es decir, que se ha terminado la comunicación con el cliente correctamente.



Terminal Servidor finalizado

7. Conclusiones:

En esta práctica se ha comprendido el concepto de ***Blockchain***, para qué sirve y por qué a día de hoy es algo necesario y bastante interesante. Además de entender los conceptos básicos acerca de la cadena de bloques como el bloque génesis, los elementos que tiene cada bloque y las funciones que realizan. Por último, la comprensión del código en el lenguaje de *Ruby* ha sido algo complejo, ya que es un lenguaje nuevo, y las implementaciones nuevas que se han añadido al repositorio mediante un *fork*.

8. Bibliografía:

- **Qué es el bloque génesis** -> https://academy.bit2me.com/que-es-bloque-genesis/
- **Elementos de un bloque** -> https://www.itdo.com/blog/que-son-los-bloques-en-la-tecnologia-blockchain/
- **Qué es *Proof of Work*** -> https://academy.bit2me.com/que-es-proof-of-work-pow/
- **Qué es PoW** -> https://www.profesionalreview.com/2021/07/31/que-es-proof-of-work/#Que_es_Proof-of-Work