Proof of Stake

Proof of Stake renueva todo el sistema de "carrera de mineros" a "grupo de validadores sin confianza." Proof of Stake fue creado para solventar la principal problemática de Proof of Work (Gasto energético).

Ya no existen los mineros, ahora los nodos serán "validadores". En este consenso, "minar un bloque" pasa a ser "forjar un bloque", que significan lo mismo. Los Validadores pueden ejercer dos roles: Forjador y Testigo.

- El forjador se encarga de verificar las transacciones y agruparlas en un bloque, firmar ese bloque con su llave privada y enviarlo a la red.
- Los Testigos se encargan de revisar que el trabajo del forjador este correcto.
- Cuando el bloque haya sido validador. El forjador anexa el bloque a la Blockchain.

Antes, los mineros probaban su validez en la red con su trabajo. Ahora, prueban su validez con su "liquidez" (o stake). Cada nodo pone su dinero en la red para probar que es de confianza; si el nodo quisiera dañar la red, su dinero (que no es una cantidad pequeña) sería destruido.

En este consenso, entre más dinero tenga un nodo puesto como "seguro" en la red, más probabilidad tiene de ser el siguiente forjador del bloque.

Cada que se escoge un nuevo forjador, se escogen también un número determinado de validadores.

Nos vamos a centrar en todo este mecanismo, poniendo un ejemplo, para después ver como se implementa en una función de nuestra cadena de bloques.

Vamos a ir desarrollando poco a poco, clase por clase

Nodos

Un nodo que está participando para ser elegido como validador o como forjador, tiene que tener si o si una cuenta en la que recibiría las recompensas de la red. Una cuenta puede usar la red sin la necesidad de ser un validador. Un nodo puede ser un validador por dos motivos:

1. Porque el usuario así lo desea.
2. Porque cuenta con dinero suficiente para demostrar que es de confianza.

En Proof of Stake, se necesita un "mínimo" de dinero virtual para ser validador de la red.

Podemos concluir que: Si tienes una cuenta en la Blockchain, y tienes mínimo N cantidad de dinero virtual, puedes ser un validador.

Para la blockchain que vamos a desarrollar, vamos a crear una nueva clase llamada Validator en el mismo archivo en el que tenemos nuestra clase Account.

Esta clase va a tener dos atributos:

- "Tokens" por cuenta.
- La dirección de la cuenta que es Validador.

In [ ]:
```
from bin.tokens import Token
from bin.account import Account

class Validator():
    def __init__(self, account):
        self.account = account
        self.tokens = []

    def set_tokens(self, total_coins):
        """Funcion que por cada moneda en el balance, se instancia un nuevo token e
        for every_coin in range(0, total_coins):
            self.tokens.append(Token(self))

    def get_tokens(self):
        """Funcion que devuelve el numero de tokens en la cuenta."""
        return len(self.tokens)
```

Los **tokens** son objetos dentro de una blockchain que puede tener una utilidad dentro de la misma. En este caso, nos van a ayudar para resolver el algoritmo de selección de forjador dentro de un grupo de validadores. Vamos a crear una clase Token.

Tokens

In [ ]:
```
class Token():
    def __init__(self, owner):
        self.owner = owner
```

La clase Token va a tener una función muy sencilla, pero muy importante. Tener un dueño al cual nombrar ganador (esto se ve mas adelante).

¿Cómo se ve la clase Forjador y Testigo?

Clase Forger

Un forjador es encargado de crear un bloque y anexarlo a red. Funciona de la siguiente manera:

1. Primero agrupa las transacciones y las verifica.
2. Crea un bloque y anexa las transacciones.
3. Firma el bloque con su llave privada y lo envía a los validadores del bloque.
4. Los demás validadores ahora serán testigos, y su trabajo será atestiguar que el nodo hizo bien su trabajo.
5. Si todos llegan a un consenso, el forjador tendrá permiso para anexar el nodo a la red.

Dentro de nuestro directorio de trabajo vamos a crear un nuevo archivo python llamado forger.py.

En el vamos a instanciar una clase llamada Forger.

In [ ]:
```
class Forger():
    pass
```

Para crear a un forjador, se tiene que inicializar con el validador que ejercerá ese rol.

In [ ]:
```
class Forger():
    def __init__(self, validator):
        pass
```

La clase que vamos a usar en nuestra Blockchain va a necesitar los siguientes atributos:

- validator: Guarda la dirección de la cuenta que está realizando la transacción.
- block: Variable que almacena el bloque que se crea.
- block_signature: Variable que almacena la firma de bloque hecha por el forjador.

In [ ]:
```
class Forger():
    def __init__(self, validator):
        self.validator = validator # Cuando se crea un Forger, se vincula con la cu
        self.block = None # Bloque creado por el forjador
        self.block_signature = None # Firma del bloque
```

En cuanto a sus métodos...

Lo primero que hace un forjador es agrupar las transacciones en espera y verificarlas.

In [ ]:
```
    def verify_tx(self, holding_tx):
            """Funcion en la que el forjador verifica que las transacciones en espe
            validas para incluirlas en el bloque. La verificacion es igual que en P
            print('El forjador esta verificando las tx.')
            verified_tx = []
            for tx in holding_tx:
                if tx.verify_signature(): # Utiliza el metodo de la clase Transacti
                    print(tx.to_dict())
                    verified_tx.append(tx) # Si la firma se verifica, se anexa a ur
            return verified_tx
```

Una vez listas las transacciones, el forjador crea un objeto bloque.

In [ ]:
```
    def create_a_block(self, _previous_hash, _verified_tx, _block_number):
        self.block = Block(_previous_hash, _verified_tx, _block_number)
```

Con el bloque creado, y con las transacciones verificadas y almacenadas, el forjador firma el bloque. El proceso es similar a cuando se firma una transacción.

In [ ]:
```
    def sign_block(self) -> bool:
        """El forjador recibe un objeto bloque, lo firma con su llave privada."""
        print("### FIRMANDO EL BLOQUE")
        # Se añade el validador al bloque.
        self.block.forger = self # Antes de firmarse el bloque, se guarda en el blo
        block_header = json.dumps(self.block.get_block_header_pos()).encode() # Se
        block_hashed = SHA256.new(block_header) # El bloque pasa por un algoritmo S
        #
        signer = self.validator.account.signer # Cada forjador es una cuenta con el
        signature = signer.sign(block_hashed)
```

```
        self.block.hash = block_hashed.hexdigest() # Se almacena dentro del bloque
        self.block_signature = signature # El forjado guarda consigo su firma del b
        print('### BLOQUE FIRMADO Y CON SU HASH ---')

    def broadcast_block(self):
        """Funcion que retorna el bloque creado del forjador."""
        return self.block
```

Con la transacción firmada, se envía a la red.

In [ ]:
```
def broadcast_block(self):
    return self.block
```

Clase Attestor

Los Attestors (Testigos en ingles) son los encargados de verificar el trabajo del forjador; "atestiguan" que todo esté bien. Su metodología iría algo así:

- Reciben el bloque del forjador.
- Revisan que la firma del bloque sea correcta.
- Si no hay nada raro, y mas del 75% de los validadores estan de acuerdo, el bloque puede pasar a ser parte de la red.

Dentro de nuestro directorio de trabajo vamos a crear un nuevo archivo python llamado attestor.py.

En el vamos a instanciar una clase llamada Attestor.

In [ ]:
```
class Attestor():
    pass
```

Para crear a un testigo, se tiene que inicializar con el validador que ejercerá ese rol.

In [ ]:
```
class Attestor():
    def __init__(self, validator):
        pass
```

Como métodos solo vamos a necesitar una función que revise el bloque.

In [ ]:
```
    def check_block(self, block):
        """Funcion que recibe un bloque, lo verifica y retorna un valor booleano si
        # Asi como se hacian las verificaciones de transacciones, se vuelve a obter
        block_header = json.dumps(block.get_block_header_pos()).encode()
        block_hashed = SHA256.new(block_header)
        # recordemos que se usa el verificador quien forjo el bloque.
        verifier = block.forger.validator.account.verifier
        try:
            verifier.verify(block_hashed, block.forger.block_signature)
            print(f"{self.validator.account.nickname}: Confirmo que la firma del bl
            return True
        except:
            print("La firma es invalida.")
            return False
        #
```

En el archivo attestor.py también vamos a desarrollar una clase Attestors (en plural.). Esta clase va a tomar a los validadores que no ganaron el sorteo, y juntarlos a todos como testigos.

In [ ]:
```
class Attestor():
    ...

class Attestors():
    pass
```

Para crear un objeto de tipo Attestor, vamos a necesitar una lista de los validadores que pasarán a ser testigos.

In [ ]:
```
class Attestors():
    def __init__(self, validators: list):
        pass
```

Como atributos solo tendremos una variable list llamada group, encargada de contener a los validadores e instanciarlos como un clase de tipo Attestor.

In [ ]:
```
class Attestors():
    def __init__(self, validators):
        self.group = [Attestor(validator) for validator in validators]
```

Esta clase va a contar con un método llamado Attest, el cual pone a cada testigo a revisar el trabajo del forjador y juntar sus confirmaciones. Si la mayoría de testigos están de acuerdo, el bloque puede ser añadido a la red.

In [4]:
```
    def attest(self, block):
        """Funcion que recibe un bloque, y va pasando a cada attestor a revisar el
        confirmations = [] # Variable que almacena las confirmaciones de los testig
        for attestor in self.group: # Por cada testigos en el grupo de todos los te
            confirmations.append(attestor.check_block(block)) # Anexa su respuesta
        print('-----------------')
        # Se hace una division para cononcer cuantos votos minimos necesitamos, si
        minimun_votes = len(self.group) // 1.25
        if sum(confirmations) > minimun_votes: # Si las confirmaciones son mayores
            print('Confirmaciones suficientes para anadir el bloque.')
            return True
        else:
            print('Confirmaciones insuficientes para anadir el bloque.')
            return False
```

Algoritmo de selección de Forjador y Testigos.

La Blockchain que utilizan el consenso Proof of Stake tiene un algoritmo que escoge al forjador y sus testigos por diversos factores. En este ejemplo, vamos a desarrollar un algoritmo de tómbola que beneficie a los nodos con más tokens en stacking.

In [5]:
```
from bin.account import Account, Validator

validadores_del_bloque = {} # validadores del bloque confirmados
total_stacked = 0 # dinero que se almacena de los validadores
```

```
# Vamos a crear 5 cuentas. Cada una de ellas se va a instanciar como un validador.
# Si bien sabemos que el dinero no sale de la nada, ponerles un balance nos ayuda p
# efectos de práctica...

charles = Account(350, 'charles')
edwin = Account(500, 'edwin')
oliver = Account(200, 'oliver')
erick = Account(90, 'erick')
sonia = Account(275, 'sonia')

# Las juntamos en una variable
lista_cuentas = [charles, edwin, oliver, erick, sonia]

# Digamos que todos quieren ser validadores, solo van a pasar aquellos
# que tengan mas de 100 de balance en su cuenta.

# Lo primero es hacer ese filto
for cuenta in lista_cuentas: # Bucle for que recorre cada una de las cuentas
    if cuenta.balance >= 100: # Si tiene 100 o mas de balance puede ser validador
        # Instanciar un nuevo objeto Validator.
        new_validator = Validator(cuenta)

        # Variable que almacena el dinero que se va a intercambiar por tokens
        account_money = int(new_validator.account.balance)

        # A traves del objeto Validator, podemos acceder al objeto Account, y
        # despues al atributo balance para restarle lo que gasto en tokens.
        new_validator.account.balance -= account_money

        # Se utiliza al funcion set_tokens para instanciar un numero
        # determinado de tokens en la cuenta. Si el usuario tiene 200
        # de balance, se van a intercambiar por 200 tokens.
        new_validator.set_tokens(account_money)

        # Se almacena en un diccionario el validador y la cuenta.
        validadores_del_bloque.update({new_validator: account_money})

        # Se suma al total stackeado de la red el balance que se cambio por tokens
        total_stacked += account_money

# Al final vamos a tener una lista de validadores. (Menos Erick)
validadores_del_bloque, total_stacked
```

Out[5]:
```
({<bin.account.Validator at 0x106446620>: 350,
  <bin.account.Validator at 0x106445420>: 500,
  <bin.account.Validator at 0x10647cca0>: 200,
  <bin.account.Validator at 0x1064ece50>: 275},
 1325)
```

In [6]:
```
# Su balance tambien fue restado.
for keys in validadores_del_bloque.keys():
    print(f"{keys.account.nickname}, {keys.account.balance}")
```

```
charles, 0
edwin, 0
oliver, 0
sonia, 0
```

Una vez que tenemos la lista neta de los validadores que van a formar parte de la forja y validación del bloque, haremos el sorteo.

¿Cómo desarrollar el algoritmo?

Dentro de esta celda, se va a diseñar un algoritmo que escoja al siguiente forger, el siguiente ejemplo fue una implementación propia, pero cada blockchain puede variar. Veámoslo:

- Cada validador tiene una cantidad finita de tokens.
- Se van a ingresar los tokens de todos los validadores que van a participar en el sorteo en una sola lista.
- Cada token tiene un dueño, por lo tanto, se sabe de quién es.
- Cada ticket que ingresaron da una la posibilidad de ser el siguiente forjador del bloque.
- La lista se revuelve y se escoge un elemento al azar.
- El dueño del ticket ganador pasa a ser el forjador del nuevo bloque, y los no ganadores a ser los validadores.

In [7]:
```
from random import sample, choice
from bin.forger import Forger


# 1.- Lista que almacenara todos los tickets de la "rifa".
pool = []
# 2.- Bucle que recorrera a cada validador, y añadara sus tokens a la lista general
for validator in validadores_del_bloque.keys():
    pool += validator.tokens
# 3.- Se revuelve la lista. (Como si fuera un sorteo.)
print('Acumulando los tokens de los validadores en el servidor actual...')
print(len(pool), '- tokens acumulados.')
print('Revolviendo la lista...')
pool = sample(pool, len(pool))
print('Lista revuelta!')

# Bucle que valida que los tokens se hayan incluido bien
contador = 0
for validator in validadores_del_bloque.keys():  # Por cada validador, en la lista
    for token in pool: # Por cada ticken en el pool
        if token.owner.account.nickname == validator.account.nickname: # Si el toke
            contador += 1
    print(validator.account.nickname, contador) # Se imprimen los tickets de cada v
    contador = 0
# Fin de la validacion

ticket_winner = choice(pool)
forger = Forger(ticket_winner.owner)
print(f'El forjador del nuevo bloque sera... {forger.validator.account.nickname}')

# los validadores no ganadores del sorteo pasan a ser testigos.
# se remueve el forjador, asi solo quedan los testigos
validadores_sin_el_forjador = validadores_del_bloque.copy() # Se hace una copia de
validadores_sin_el_forjador.pop(forger.validator) # Se elimina al forjador de esta
"no ganadores",[validator.account.nickname for validator in validadores_sin_el_for
```

```
Acumulando los tokens de los validadores en el servidor actual...
1325 - tokens acumulados.
Revolviendo la lista...
Lista revuelta!
charles 350
edwin 500
oliver 200
sonia 275
El forjador del nuevo bloque sera... charles
```

Out[7]: ('no ganadores', ['edwin', 'oliver', 'sonia'])

Aquellos que no ganaron el sorteo, pasan a ser "testigos".

In [8]:
```
from bin.attestor import Attestor, Attestors

# Para instanciar al grupo de testigos, necesitamos usar la clase Attestors
attestors = Attestors(validadores_sin_el_forjador)
```

Ya tenemos a nuestro forjador y nuestro testigo, ¿Qué seguiria?

Procedimiento del Forjador y del Testigo

Lo primero que sucede es que el Forjador recibe un bloque y verifica sus transacciones.

In [19]:
```
from bin.account import Account
from bin.transaction import Transaction
from bin.block import Block

# Instanciamos un bloque previo al actual para efectos de práctica
cadena_de_bloques = [Block('0', [], 0)]

# Vamos a crear una transaccion entre dos cuentas.
tx = Transaction(Account(100, 'pedro'), 10, Account(100, 'maria'))

# Cada transaccion se firma con la llave privada del que envia la transaccion.
tx.sign_transaction()

# Se anade la tx a una lista de espera
holding_tx = [tx]

# El forjador verifica las transacciones en la lista
verified_tx = forger.verify_tx(holding_tx)

# El forjador crea un bloque
forger.create_a_block(cadena_de_bloques[-1].hash, verified_tx, len(cadena_de_bloque

# El forjador firma el bloque
forger.sign_block()

# El forjador envia el bloque a la red
bloque_firmado = forger.broadcast_block()

if attestors.attest(bloque_firmado):
    print('TODO CORRECTO, LLEGAMOS A UN ACUERDO')
    cadena_de_bloques.append(bloque_firmado)
cadena_de_bloques[1].block_number
```

```
Firmando transaccion...
El forjador esta verificando las tx.
Verificando la firma de la transaccion...
La firma es valida.
{'sender': 'pedro', 'recipient': 'maria', 'value': 10, 'time': '24/09/2022 17:20:2
0'}
### FIRMANDO EL BLOQUE
### BLOQUE FIRMADO Y CON SU HASH ---
edwin: Confirmo que la firma del bloque es correcta.
oliver: Confirmo que la firma del bloque es correcta.
sonia: Confirmo que la firma del bloque es correcta.
-----------------
Confirmaciones suficientes para anadir el bloque.
TODO CORRECTO, LLEGAMOS A UN ACUERDO
```

Out[19]: 1

Ya abarcamos todas las clases que se desarrollan dentro de la clase Blockchain. Llego la hora de ir a nuestra clase principal y ver cómo podemos integrar todo lo anterior.