Python에서의 블록체인 시뮬레이션 구현

201825206 부은성

데이터처리언어 기말 프로젝트

1. 프로젝트의 목표

   - 블록체인을 간략화하여 python상에서 시뮬레이션을 구현
   - 블록을 생성하는 '채굴자' 노드들을 다수 생성하여 시뮬레이션 실행
   - 채굴자들이 블록을 생성함에 따라 받게 되는 '블록생성보상'을 그래프상으로 표현하여 어떠한 채굴자들이 더 많은 채굴을 하는지 확인

2. 블록체인이란?

   블록체인은 하나의 중앙화된 노드나 서버가 아닌 다수의 노드가 네트워크에 참여하여 '블록'을 생성하고, 이러한 블록들이 사슬처럼 연결된 하나의 장부로 볼 수 있다. 모든 거래들은 블록에 담김으로써 블록이 새로 생성될 때마다 블록의 최대 사이즈만큼의 거래들을 처리할 수 있다. 가령 A가 B에게 디지털상의 화폐를 전송한다고 했을 때 이는 하나의 트랜잭션으로 간주된다. 이러한 트랜잭션은 어떤 은행같이 신뢰할 수 있는 제 3자의 개입 없이도, 다수의 노드가 Proof-of-work란 작업을 통해 블록을 생성하고 모든 노드가 생성되는 블록을 기록함으로써 트랜잭션들이 위변조를 어렵게 하고 이중지불문제를 어렵게 한다. 따라서 신뢰할 수 있는 제 3자 없이도, 디지털 화폐의 peer-to-peer 거래가 가능하며, 데이터의 위변조가 매우 힘든 구조를 가진다. 블록체인의 개념은 사토시 나가모토의 bitcoin 논문에서 최초로 나타났으며 최근 들어 많은 조명을 받고 있다. 본 리포트에서는 peer-to-peer 구조로 데이터를 처리하고, 데이터의 위변조가 힘든 블록체인의 주요 핵심기능을 Python 상에서 구현한다.

3. 블록의 구조

블록의 구조는 아래 Figure 1과 같다. 블록의 컴포넌트는 크게 3가지로 나눌 수 있다. 개별 트랜잭션, 블록헤더, 그리고 블록해쉬로 구성된다. 블록 해쉬는 블록헤더를 hashing 하여 구하게 되는데 hashing 한 값이 특정 Threshold보다 낮은 값을 찾아야 한다. 블록 헤더의 채굴 난이도가 이 Threshold를 결정하는 역할을 한다.

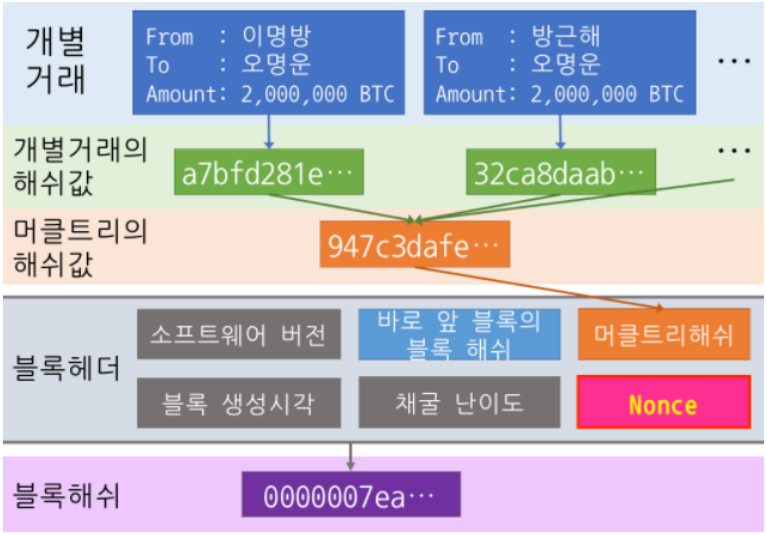


Figure 1. 블록의 구조

본 리포트에서는 Figure 1의 블록 구조를 간략화하여 주요 컴포넌트인 블록해쉬, 블록헤더, 트랜잭션, nonce값으로 블록을 구성하도록 하였다.

```
def new_block(self,index, miner_name = None):
      #새로운 block 생성
      #block 구조: block hash + block header + transactions + nonce
      tx_list = self.generate_coinbase_transaction(miner_name) + self.current_transactions
      tx_num = len(tx_list)

      blockheader = {
        'index': index,
        'timestamp': time(),
        'previous_hash': self.get_previous_hash(),
        'transactions_hash': self.transactions_hash(tx_list)
      }

      #find nonce
      nonce = self.proof_of_work(blockheader);

      block = {
        'block_hash': self.calculate_block_hash(nonce,blockheader),
        'header': blockheader,
        'transactions': tx_list,
        'nonce': nonce
      }

      return block,tx_num
```

Figure 2. 블록의 생성

위는 블록의 구조를 간략화하여 python 코드상에 표현한 것으로, block에는 block_hash, header, transactions, nonce가 들어가 있다. Blockheader는 다시 index, timestamp, previous_hash, transactions_hash로 세분화된다. Index는 블록의 순서를 의미하며, timestamp는 블록 생성 시간, previous_hash는 이전 블록의 hash값,

transactions_hash 값은 블록에 담기는 트랜잭션들의 hash값을 계산한 값이다.

4. Proof-of-Work

Proof-of-Work란 블록체인의 가장 기본적인 합의 알고리즘(consensus algorithm)으로, 다수의 노드가 있을 때 Proof-of-work를 통해서 하나의 노드가 블록생성권한을 얻는다. 즉, Proof-of-Work는 블록의 Hash값을 찾는 행위로 이 Hash 값이 특정 Threshold 보다 낮은 값을 찾을 때까지 블록의 nonce값을 증가시키면서 Hashing 작업을 반복한다. 이런 반복적인 작업 도중에 특정 Threshold값보다 낮은 Hash값을 어떠한 노드가 발견한다면, 이 노드가 블록을 생성하게 되고 다른 노드들에게 생성한 블록을 전달한다. 따라서 Proof-of-Work는 다수의 노드가 존재할 때 어떠한 노드가 블록을 생성할 권한을 갖는가에 대한 합의로 볼 수 있으며, 블록체인의 가장 기본적인 토대가 되는 알고리즘이다. 아래 Figure 3은 본 알고리즘을 Python 상에서 구현한 코드이다.

```
def proof_of_work(self,bh):
    # 앞자리수 5개가 0이 되는 hash 값을 찾을 때까지 nonce값을 증가시키면서 해쉬값 계산
    nonce = 0
    while (1):
        hash_value = self.calculate_block_hash(nonce,bh);
        if hash_value[0:5] == '00000':
          print ("find hash value " + hash_value)
          break
        nonce += 1
    return nonce
```

Figure 3. Proof-of-Work (PoW)

Proof_of_work 함수는 block header를 인자로 받는다. 그리고 nonce 값과 block header를 인자로 삼아 Hashing 값을 계산한다. 이 Hashing value의 앞자리 5개의 수가 0이 되는 Hash값을 찾게 되는데, nonce를 1씩 증가시키면서 Hashing 값 계산을 반복한다. 또한 이러한 hashing 값을 찾는 노드들을 흔히 '채굴자 노드'라고 한다.

5. Python Code 구현 설명

Python 코드 상에선 Miner 클래스와 Blockchain 클래스 두개가 존재한다.

Blockchain 클래스의 함수들의 설명은 다음과 같다.

def __init__(self, miner_name): Blockchain 클래스의 생성자로 변수들을 초기화하고 첫번째 블록(genesis block)을 초기화한다. Miner_name 인자는 첫번째 블록을 생성하는 채굴자의 이름이 된다.

def new_block(self,index, miner_name = None): 새로운 블록을 생성하며 적절한

nonce값을 찾기 위해서 proof_of_work 함수를 호출한다.

def generate_coinbase_transaction(self, miner_name): 채굴자들은 블록 생성에 대한 보상을 얻게 되는데, 이러한 보상은 블록의 첫번째 트랜잭션으로 등록된다. 즉 매 블록 생성마다 채굴자들은 자신들의 보상을 위한 트랜잭션을 생성하고 블록의 첫번째 트랜잭션에 넣는다. 이를 '코인베이스 트랜잭션'이라고 한다. 이 함수는 '코인베이스 트랜잭션'을 생성하기 위한 함수이다. 블록 생성 보상량은 250으로 고정시켜 놓았다.

def new_transaction(self, sender, receiver, amount, data): 가령 A가 B에게 특정 양의 디지털화폐를 전송 시킬 때 새로운 트랜잭션이 생성되고 이는 다음블록 생성에 포함되어야 한다. 이 함수는 새로운 트랜잭션을 생성하고 트랜잭션 리스트에 포함시킨다.

def transactions_hash(self,tx_list): 트랜잭션 리스트에 담겨있는 트랜잭션들의 hash 값을 계산하여 반환한다.

def calculate_block_hash(self,nonce,bh): block header와 nonce를 인자로 하여 Hashing 값을 계산한다.

def proof_of_work(self,bh): calculate_block_hash를 통해 hashing value를 구하고 특정한 Threshold보다 낮은 값을 찾을 때까지 nonce값을 증가시키며 반복한다.

def insert_block(self,block,tx_num): validate_block을 호출해 생성한 block이 유효한다고 확인되면 새로운 블록을 블록체인 리스트에 추가한다**.**

def validate_block(self,block): 블록이 유효한지를 검사한다. 같은 index를 가진 블록이 이미 다른 노드를 통해 생성되 있는지는 아닌지, 적절한 Hashing value를 갖고있는 지를 검사한다.

def balance_update(self,block): 블록이 새로 리스트에 추가될 경우, 블록의 내부 트랜잭션들을 반영하여 전체 잔고를 업데이트한다.

채굴자(Miner) 클래스의 함수는 다음과 같다.

def __init__(self,miner_name, Blockchain = None) : 채굴자 클래스의 생성자로 miner_name과 Blockchain 객체를 받아 변수를 초기화한다.

def mining(self): blockchain class의 new_block 함수를 호출하여 블록을 생성한다.

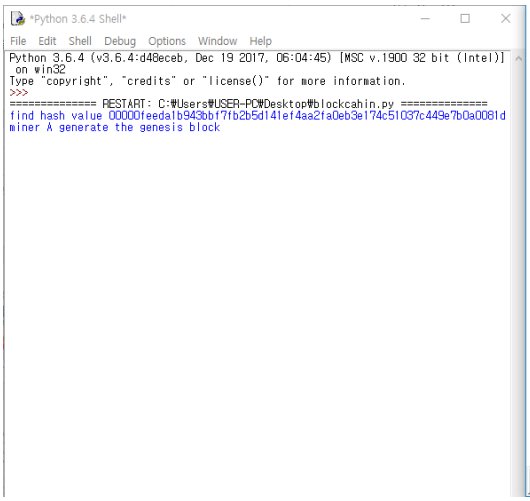
6. 시뮬레이션 결과

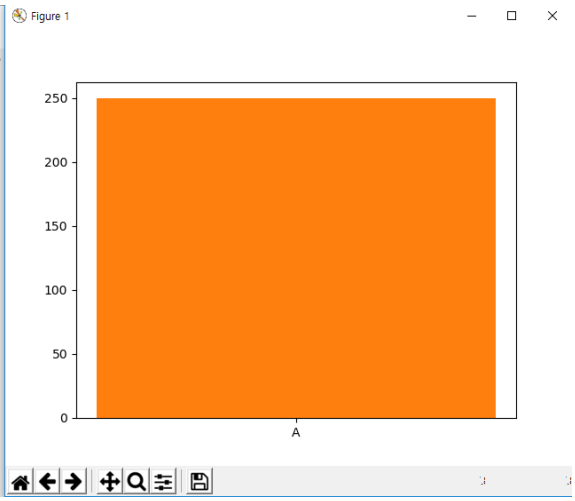
시뮬레이션은 다음과 같이 진행되었다.

1. Main 함수에서 A,B,C,D,E 의 채굴자 리스트를 생성한다.
2. A채굴자를 Genesis block의 생성자로 하여 blockchain class를 생성한다.

   Code: blockchain = Blockchain(miner_name = 'A')
3. 채굴자 리스트에 있는 채굴자 이름을 갖고와 miner_thread를 생성하여 채굴자 객체를 생성한다.
4. 채굴자 객체들은 miner 클래스 내부 함수인 mining을 수행하여 블록 생성을 수행한다.
5. 채굴자 노드들은 경쟁적으로 블록을 생성하게 되고 채굴 보상을 받게 된다.
6. 5초마다 채굴자 노드들이 얼마만큼의 채굴 보상을 받았는지를 보여준다.

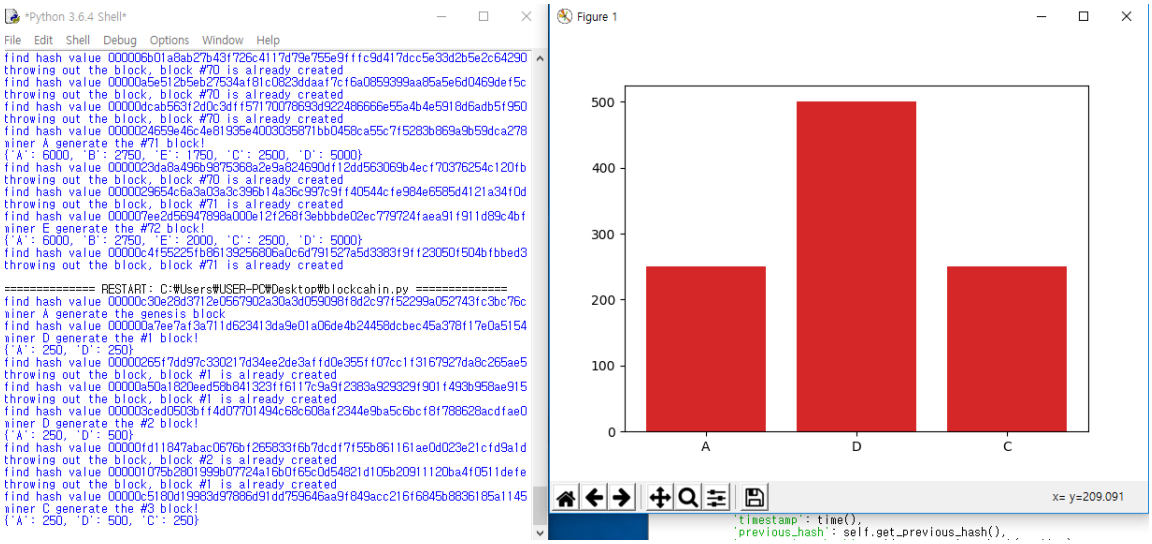
2번을 실행하였을 때 다음과 같은 메시지와 그래프가 출력되는 것을 확인할 수 있다





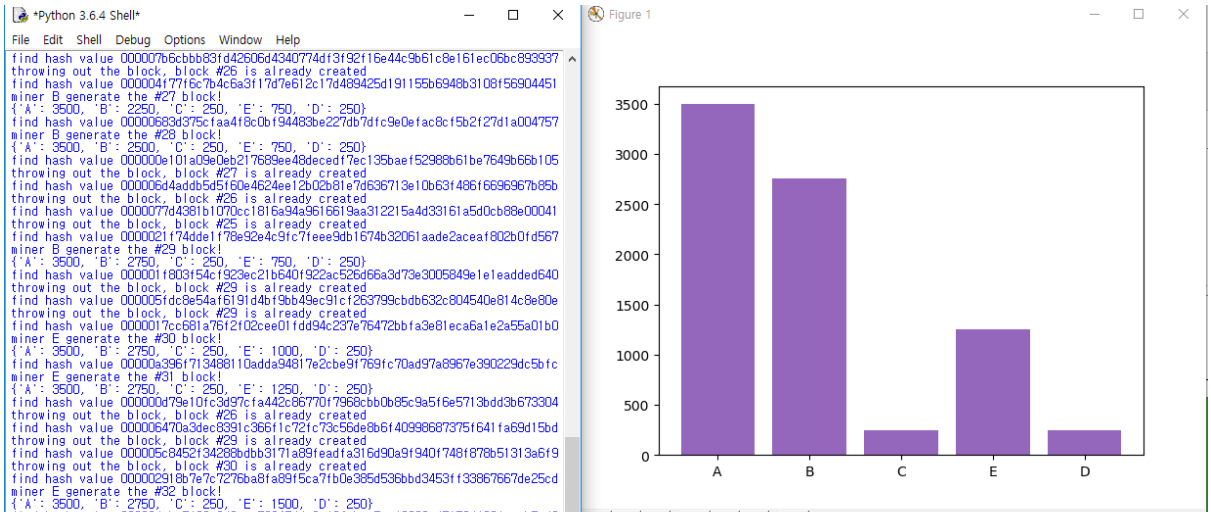
Hash value는 앞자리 다섯자리 수가 0인 것을 확인할 수 있고, Miner A가 첫번째 블록을 생성하였다. 블록 생성에 대한 보상으로 A는 250만큼의 가상화폐를 얻었고 이는 위의 그림의 오른쪽 그래프에서 확인할 수 있다.

모든 채굴자 스레드가 생성되고 채굴함에 따라 다음과 같은 변화를 보인다.



위의 그림은 총 4개의 block이 생성되었을 때에 모습이다. A가 genesis block을, D가 #1 block, #2 block을 C가 #3 block 을 생성하였기 때문에 왼쪽 시리얼창의 마지막 print 문에서 balance가 {A: 250, D: 500, C: 250} 인 것을 확인할 수 있다. 오른쪽 그래프는 이 Balance를 반영한 그래프이다.

#33 블록이 생성 됐을 때의 시리얼 창과 그래프는 다음과 같다.



오른쪽 그래프를 보면 알 수 있듯이 채굴 보상은 A>B>E>C=D 순으로 많다.

이는 A>B>E>C=D 순으로 블록을 더 많이 생성한 것을 의미한다.

지금은 Thread 상으로 각각의 채굴 객체를 생성하여 블록을 생성하도록 하였기 때문에 어떠한 일관성을 찾아볼 수 없다. 그때 그때 실행할 때마다 전부 다른 그래프를 가지며, 어떠한 채굴자들이 더 많은 블록을 생성하는지에 대한 일관성이 없다. 하지만 각각 계산 능력이 다른 컴퓨터 마다 각자 하나의 채굴객체를 생성한다면, 컴퓨팅 파워가 큰 컴퓨터가 더 많은 블

록을 생성하고, 채굴 보상을 더 많이 얻을 것이다.