Contrôle Continu

Implémentation d'une Blockchain en Python

Guinga Aly
Filière : Sciences des Données, Big Data & IA
Module : Fondamentaux de la Blockchain
Prof. Imad Sassi

Table des matières

1 Introduction

Ce rapport présente l'implémentation d'une blockchain fonctionnelle en Python réalisée dans le cadre du Contrôle Continu. Le travail est divisé en deux parties : la création d'une blockchain mononœud (Partie I) et son extension vers un réseau décentralisé avec consensus (Partie II).

2 Partie I : Blockchain Mononœud

2.1 Architecture et Concepts Clés

Notre implémentation repose sur la classe `Blockchain` qui gère :
— La chaîne de blocs (`self.chain`)
— Les transactions en attente (`self.current_transactions`)
— Le bloc genesis créé à l'initialisation
Chaque bloc contient : index, timestamp, transactions, preuve de travail (PoW) et hash du bloc précédent.

2.2 Hachage SHA-256

Le hachage utilise `sort_keys=True` pour garantir un résultat déterministe :

Listing 1 – Fonction de hachage

```python
@staticmethod
def hash(block):
    block_string = json.dumps(block, sort_keys=True).encode()
    return hashlib.sha256(block_string).hexdigest()
```

2.3 Preuve de Travail (PoW)

Difficulté fixée à '0000' (4 zéros). L'algorithme incrémente un nonce jusqu'à trouver un hash valide :

Listing 2 – Proof of Work

```python
def proof_of_work(self, last_proof):
    proof = 0
    while self.valid_proof(last_proof, proof) is False:
        proof += 1
    return proof

@staticmethod
def valid_proof(last_proof, proof):
    guess = f'{last_proof}{proof}'.encode()
    guess_hash = hashlib.sha256(guess).hexdigest()
    return guess_hash.startswith("0000")
```

2.4 API REST - Endpoints Partie I

2.5 Tests et Résultats - Partie I

2.5.1 Installation et Démarrage

Listing 3 – Installation

```
pip install flask requests
python blockchain.py
```

Endpoint	Méthode	Description
`/mine`	GET	Mine un nouveau bloc avec récompense
`/transactions/new`	POST	Ajoute une transaction
`/get_chain`	GET	Retourne la blockchain complète
`/is_valid`	GET	Vérifie l'intégrité de la chaîne

TABLE 1 – Endpoints Partie I

FIGURE 1 – Démarrage du serveur

2.5.2 Test 1 : Consultation de la chaîne initiale

```
Invoke-WebRequest -Uri http://localhost:5000/get_chain |
    Select-Object -ExpandProperty Content
```

FIGURE 2 – Blockchain initiale (bloc genesis)

2.5.3 Test 2 : Création de transaction et minage

```
# Creer une transaction
$body = @{sender="Alice"; recipient="Bob"; amount=10} | ConvertTo-Json
Invoke-WebRequest -Uri http://localhost:5000/transactions/new `
    -Method POST -Body $body -ContentType "application/json"

# Miner le bloc
Invoke-WebRequest -Uri http://localhost:5000/mine
```

2.5.4 Test 3 : Validation de la chaîne

```
Invoke-WebRequest -Uri http://localhost:5000/is_valid
```

3 Partie II : Décentralisation et Consensus

3.1 Réseau de Nœuds

Implémentation d'un système d'enregistrement des nœuds voisins utilisant un `set` pour garantir l'idempotence :

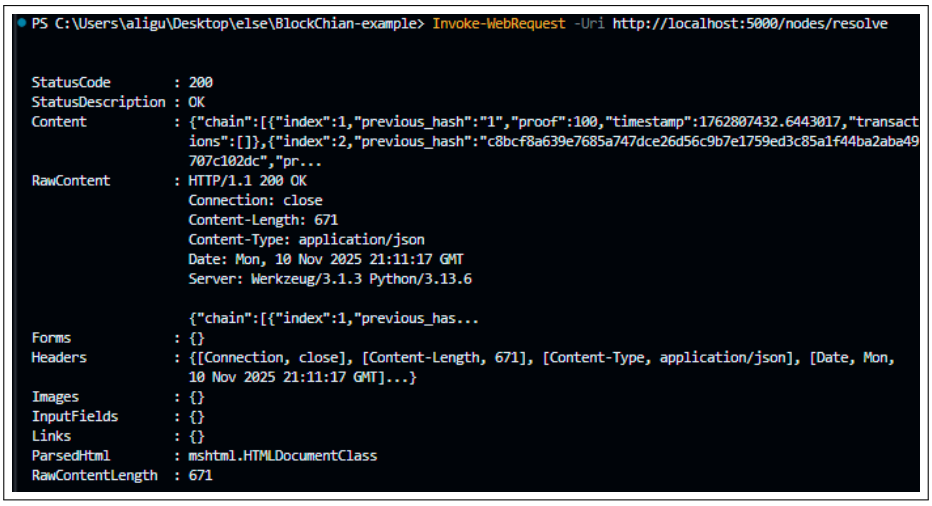

FIGURE 3 – Résultat du minage avec récompense

FIGURE 4 – Validation de l'intégrité

Listing 4 – Enregistrement de nœuds

```python
def register_node(self, address):
    parsed_url = urlparse(address)
    if parsed_url.netloc:
        self.nodes.add(parsed_url.netloc)
    elif parsed_url.path:
        self.nodes.add(parsed_url.path)
```

3.2 Algorithme de Consensus

Implémentation de la règle de "la chaîne la plus longue" :

Listing 5 – Résolution des conflits

```python
def resolve_conflicts(self):
    neighbours = self.nodes
    new_chain = None
    max_length = len(self.chain)

    for node in neighbours:
        response = requests.get(f'http://{node}/get_chain')
        if response.status_code == 200:
            length = response.json()['length']
            chain = response.json()['chain']

            if length > max_length and self.is_chain_valid(chain):
                max_length = length
                new_chain = chain

    if new_chain:
        self.chain = new_chain
        return True
    return False
```

3.3 API REST - Endpoints Partie II

Endpoint	Méthode	Description
`/nodes/register`	POST	Enregistre des nœuds voisins
`/nodes/resolve`	GET	Lance l'algorithme de consensus

TABLE 2 – Endpoints Partie II

3.4 Tests et Résultats - Partie II

3.4.1 Configuration Multi-Nœuds

Lancement de deux instances sur les ports 5000 et 5001 :

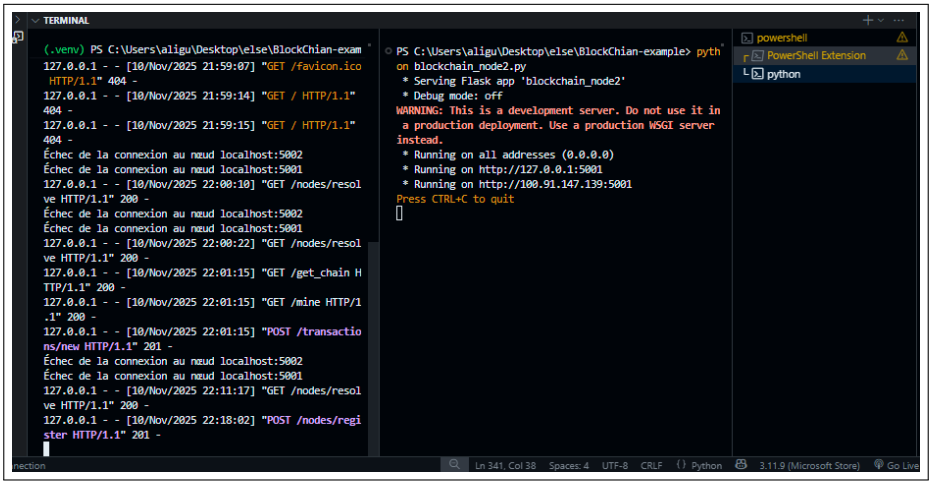

FIGURE 5 – Deux nœuds actifs

3.4.2 Test 4 : Enregistrement de nœuds

```
$body = @{nodes=@("http://localhost:5001")} | ConvertTo-Json
Invoke-WebRequest -Uri http://localhost:5000/nodes/register `
    -Method POST -Body $body -ContentType "application/json"
```

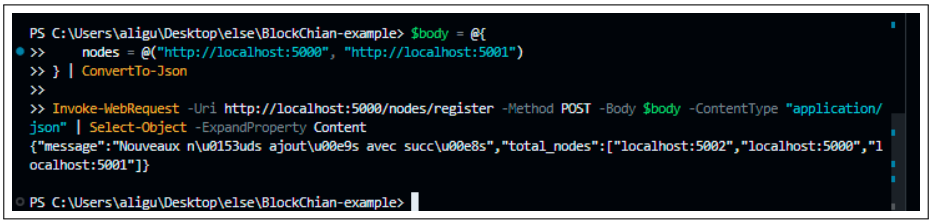

FIGURE 6 – Enregistrement réussi

3.4.3 Test 5 : Création d'un conflit

Création de chaînes de longueurs différentes :
— Nœud 1 (port 5000) : 3 blocs
— Nœud 2 (port 5001) : 4 blocs

FIGURE 7 – Conflit : chaînes de longueurs différentes

3.4.4 Test 6 : Résolution du consensus

```
Invoke-WebRequest -Uri http://localhost:5000/nodes/resolve
```

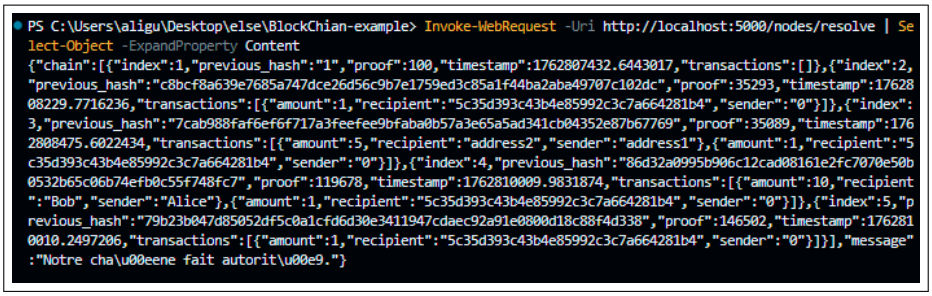

FIGURE 8 – Consensus résolu : chaîne remplacée

Résultat : Le nœud 1 adopte la chaîne du nœud 2 (4 blocs). Les deux nœuds sont maintenant synchronisés.

4 Conclusion

Cette implémentation a permis d'implémenter une blockchain fonctionnelle comprenant :

Partie I :

— Structure de blocs avec hachage SHA-256
— Preuve de Travail (difficulté '0000')
— Système de transactions avec récompense de minage
— Validation de l'intégrité de la chaîne
— API REST avec 4 endpoints

Partie II :

— Système de réseau décentralisé
— Enregistrement de nœuds voisins
— Algorithme de consensus (chaîne la plus longue)
— Résolution automatique des conflits

Tous les tests ont été effectués avec succès, démontrant le bon fonctionnement de l'implémentation conforme aux exigences du sujet.