RAPPORT TECHNIQUE : PROTOTYPE DE BLOCKCHAIN SIMPLIFIEE

Réalisé par :

1. Warisse ABOUDOU
2. Emmanuel ADJALIAN
3. Aïcha TIDJANI

Filière : Master II Finance Digitale

Module : Blockchain et Cryptomonnaie

Formateur : M. Lionnel P. DOOKO

ESMT – Janvier 2025

Liste des captures

Sommaire

Introduction

Ce rapport présente le développement d'un prototype de blockchain simplifiée. Le projet illustre les concepts fondamentaux des blockchains : création de blocs, transactions, méthode de consensus (Proof of Work), gestion de la chaîne, et résolution de conflits. La blockchain se présente comme une chaîne de blocs interconnectés, où chaque bloc renferme des informations : un horodatage précis, et un hash cryptographique du bloc précédent... Ce mécanisme garantit non seulement l'intégrité des données, mais rend également toute tentative de modification des informations antérieures presque impossible. Développé en Python, ce prototype utilise la Programmation Orientée Objet (POO) pour structurer le code de manière modulaire et évolutive. Chaque composant a été soigneusement testé pour assurer son bon fonctionnement, permettant ainsi une exploration approfondie des fonctionnalités essentielles telles que la création de blocs, la gestion des transactions, et la mise en œuvre de la preuve de travail (Proof of Work). Les outils de travail utilisés incluent : VS Code, Google Colab, Github, Chatgpt.

Comprendre le Code Python

Notre implémentation de la blockchain en Python se compose de plusieurs composants principaux :

Classe Block

Elle représente un bloc individuel de la blockchain. Chaque bloc contient un **index**, un horodatage (**timestamp**), une liste de **transaction(s)**, l'hash du bloc précédent (**previous_hash**), un hash de lui-même (**hash**) et un **nonce**.

L'horodatage est géré automatiquement à l'aide du module **datetime** de Python. Ce module fournit des classes pour manipuler les dates, les heures et les intervalles de temps avec des fonctionnalités avancées comme la gestion des fuseaux horaires et le formatage flexible des dates/heures. Contrairement au module **time**, il est plus lisible, riche en fonctionnalités, et traite directement des objets de type date et heure, évitant les manipulations fastidieuses de timestamps.

Le hash est calculé à l'aide de l'algorithme **SHA-256** (Secure Hash Algorithm 256 bits) en combinant les champs du bloc. L'algorithme SHA-256 est une fonction de hachage cryptographique qui prend une donnée en entrée et génère une empreinte unique, fixe de 256 bits (64 caractères hexadécimaux). Il est conçu pour être déterministe, rapide, et résistant aux collisions, préimages, et attaques de second préimage. Par rapport à des algorithmes plus anciens comme MD5 ou SHA-1, il offre une bien meilleure résistance aux collisions et attaques cryptographiques. Le module **hashlib** a été utilisé pour l'implémenter à travers la méthode **generate_hash()**.

Elle contient également une méthode **proof_of_work()** pour la preuve de travail. Dans la preuve de travail, le nonce est une valeur modifiable utilisée pour générer des hachages valides respectant une condition [**Le nonce est incrémenté jusqu'à ce que le hash commence par un certain nombre de zéros** (déterminé par la difficulté)].

```python
class Block:
    """Represents a block in the blockchain."""

    def __init__(self, index, transactions, previous_hash, timestamp=None, nonce=0):
        """
        Initialize a new block with :
        - index : the index of the block in the blockchain
        - transactions : a list of transactions
        - previous_hash : the hash of the previous block
        - timestamp : the timestamp of the block creation
        - nonce : Value of the proof of work
        """
        self.index = index
        self.timestamp = timestamp or datetime.now().strftime("%d/%m/%Y %H:%M:%S %f")
        self.transactions = transactions
        self.previous_hash = previous_hash
        self.nonce = nonce                      # Using nonce for proof of work
        self.hash = self.generate_hash()        # Hash of the block

    def generate_hash(self):
        """
        Calculates the hash of the block based on its contents.
        Combines index, transactions, previous_hash, timestamp and nonce into a JSON string
        and generates a SHA-256 hash.
        """
        block_string = (str(self.index) + str(self.timestamp) + str(self.transactions)
                        + str(self.previous_hash) + str(self.nonce))
        return hashlib.sha256(block_string.encode()).hexdigest()

    def proof_of_work(self, difficulty):
        """
        Performs a proof of work by finding a hash with a certain number of zeros.
        Increments the nonce until the hash starts with 'difficulty' zeros.
        """
        while not self.hash.startswith("0" * difficulty):
            self.nonce += 1
            self.hash = self.generate_hash()
```

Capture 1 *: Implémentation de la classe Block*

Classe Transaction

Elle représente une transaction de valeur entre deux parties. Chaque transaction contient l'identifiant de l'expéditeur (**sender**), l'identifiant du destinataire (**receiver**) et le montant de la transaction (**amount**). La méthode **is_valid()** assure que les transactions sont correctement renseignées (par exemple, un montant positif). La méthode **to_json()** sérialise une transaction en dictionnaire JSON-compatible. Le JSON est un format léger et lisible par les humains pour partager des données, très utilisé dans les applications web et systèmes distribués.

```python
class Transaction:
    """Represents a transaction in the blockchain."""

    def __init__(self, sender, receiver, amount):
        """
        Initialize a new transaction with:
        - sender: The identifier of the sender.
        - recipient: The identifier of the recipient.
        - amount: The amount being transferred.
        """
        self.sender = sender
        self.receiver = receiver
        self.amount = amount

    def to_json(self):
        """
        Serializes the transaction into a JSON-compatible dictionary.
        """
        return {"sender": self.sender,
                "receiver": self.receiver,
                "amount": self.amount
                }

    def is_valid(self):
        """
        Validates the transaction.
        Ensures that all fields are properly filled and the amount is positive.
        """
        if not self.sender or not self.receiver:
            return False                        # Sender and receiver must be filled
        if self.amount <= 0:
            return False                        # Amount must be positive
        return True
```

Capture 2 *: Implémentation de la classe Transaction*

Classe Blockchain

Elle représente la chaîne de blocs elle-même. Elle contient une liste de blocs, des méthodes pour ajouter des blocs [**add_block()**], vérifier la validité de la chaîne [**is_chain_valid()**], de résolution de conflits entre deux chaînes [**resolve_conflicts()**]. La méthode resolve_conflicts() compare la chaîne locale à une chaîne externe et applique la règle de la **chaîne la plus longue**. La méthode **is_valid_chain()** valide la chaîne externe avant remplacement.

```python
class Blockchain:
    """
    Represents a chain of blocks.
    """
    def __init__(self):
        """
        Initializes the blockchain with a genesis block and a list to store blocks.
        """
        self.chain = [self.create_genesis_block()]
        self.difficulty = 4

    def create_genesis_block(self):
        """
        Creates the first block in the blockchain, known as the genesis block.
        """
        return Block(0, [], "0", datetime.now().strftime("%d/%m/%Y %H:%M:%S %f"))

    def add_block(self, new_block, difficulty):
        """
        Adds a new block to the blockchain after solving the proof of work.
        """
        new_block.previous_hash = self.chain[-1].hash
        new_block.proof_of_work(difficulty)
        self.chain.append(new_block)

    def is_chain_valid(self):
        """
        Validates the entire blockchain.
        Ensures that each block's hash and linkage to the previous block is correct.
        """
        for i in range(1, len(self.chain)):
            current_block = self.chain[i]
            previous_block = self.chain[i - 1]

            # Check if the current block's hash is correct
            if current_block.hash != current_block.generate_hash():
                return False

            # Check if the current block's previous hash matches the previous block's hash
            if current_block.previous_hash != previous_block.hash:
                return False
        return True
```

Capture 3 : *Implémentation de la classe Blockchain (Début)*

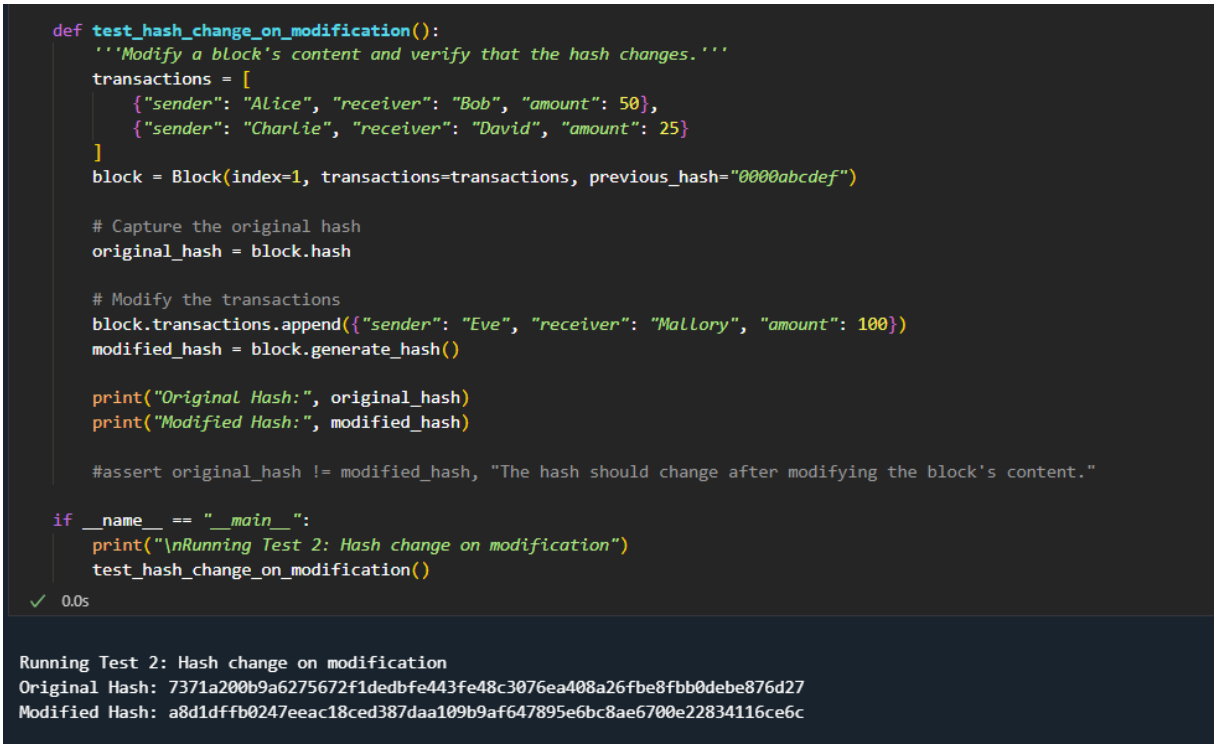
```python
def display_chain(self):
    """
    Displays the blockchain in a readable format.
    """
    chain_data = []
    for block in self.chain:
        temp = json.dumps({"index": block.index,
                           "timestamp": block.timestamp,
                           "transactions": block.transactions,
                           "hash": block.hash,
                           "previous_hash": block.previous_hash,
                           "nonce": block.nonce
                           }, indent=4)
        chain_data.append(temp)
    return "\n".join(chain_data)

def resolve_conflicts(self, other_chain):
    """
    Resolves conflicts by replacing the chain with the longest valid chain if necessary.
    """
    if len(other_chain) > len(self.chain) and self.is_valid_chain(other_chain):
        self.chain = other_chain

def is_valid_chain(self, chain):
    """
    Validates an external chain.
    """
    for i in range(1, len(chain)):
        current_block = chain[i]
        previous_block = chain[i - 1]

        # Check if the current block's hash is correct
        if current_block.hash != current_block.generate_hash():
            return False

        # Check if the current block's previous hash matches the previous block's hash
        if current_block.previous_hash != previous_block.hash:
            return False
    return True
```

Capture 4 : *Implémentation de la classe Blockchain (Fin)*

Comprendre les tests effectués

Chaque test est conçu pour être compatible avec **Pytest**. Pytest est un framework permettant d'écrire des tests unitaires simples et lisibles. Les fonctions de test sont identifiées par leur préfixe **test_**, et les assertions permettent de valider le comportement attendu des différentes fonctionnalités implémentées.

Test 1 : Valider que chaque bloc génère un hash unique basé sur son contenu.



```
def test_hash_change_on_modification():
    '''Modify a block's content and verify that the hash changes.'''
    transactions = [
        {"sender": "Alice", "receiver": "Bob", "amount": 50},
        {"sender": "Charlie", "receiver": "David", "amount": 25}
    ]
    block = Block(index=1, transactions=transactions, previous_hash="0000abcdef")

    # Capture the original hash
    original_hash = block.hash

    # Modify the transactions
    block.transactions.append({"sender": "Eve", "receiver": "Mallory", "amount": 100})
    modified_hash = block.generate_hash()

    print("Original Hash:", original_hash)
    print("Modified Hash:", modified_hash)

    #assert original_hash != modified_hash, "The hash should change after modifying the block's content."

if __name__ == "__main__":
    print("\nRunning Test 2: Hash change on modification")
    test_hash_change_on_modification()
```

✓ 0.0s

```
Running Test 2: Hash change on modification
Original Hash: 7371a200b9a6275672f1dedbfe443fe48c3076ea408a26fbe8fbb0debe876d27
Modified Hash: a8d1dffb0247eeac18ced387daa109b9af647895e6bc8ae6700e22834116ce6c
```

Capture 5 *: Test de génération de hash unique basé sur le contenu*

Test 2 : Créer des transactions valides et invalides, puis valider à l'aide de is_valid().

```python
def test_transaction_valid():
    '''Validate a transaction with a positive amount.'''
    transaction = Transaction("Boubs", "Amina", 500)
    print("Validity of transaction:", transaction.is_valid())
    assert transaction.is_valid() == True, "Valid transaction failed validation."

def test_transaction_invalid_negative_amount():
    '''Validate a transaction with a negative amount.'''
    transaction = Transaction("Sonia", "Felix", -100)
    print("Validity of transaction:", transaction.is_valid())
    assert transaction.is_valid() == False, "Transaction with negative amount passed validation."

if __name__ == "__main__":
    print("\nRunning Test 3: Transaction Validation")
    test_transaction_valid()
    test_transaction_invalid_negative_amount()
```

✓ 0.0s

```
Running Test 3: Transaction Validation
Validity of transaction: True
Validity of transaction: False
```

Capture 6 *: Test de validation d'une transaction*

Test 3 : Vérifier la sérialisation en dictionnaire JSON Compatible d'une transaction

```python
def test_transaction_serialization():
    '''Verify the JSON serialization of a transaction.'''
    transaction = Transaction(sender="Alice", receiver="Bob", amount=50)
    serialized = transaction.to_json()

    print("Serialized Transaction:", serialized)

    expected_serialized = {
        "sender": "Alice",
        "receiver": "Bob",
        "amount": 50
    }

    assert serialized == expected_serialized, "The serialized transaction should match the expected JSON format."

if __name__ == "__main__":
    print("\nRunning Test 4: Transaction Serialization")
    test_transaction_serialization()
```

✓ 0.0s

```
Running Test 4: Transaction Serialization
Serialized Transaction: {'sender': 'Alice', 'receiver': 'Bob', 'amount': 50}
```

Capture 7 *: Test de sérialisation d'une transaction*

Test 4 : Définir une difficulté et vérifier que le hash respecte les critères après le minage.

```python
def test_proof_of_work():
    '''Verify that the proof of work generates a hash with the correct number of leading zeros.'''
    transactions = [
        {"sender": "Alice", "receiver": "Bob", "amount": 50}
    ]


    difficulties = [2, 4, 5]
    for difficulty in difficulties:
        block = Block(index=1, transactions=transactions, previous_hash="0000abcdef")
        block.proof_of_work(difficulty)

        print(f"Difficulty: {difficulty}, Mined Hash: {block.hash}")
        assert block.hash.startswith("0" * difficulty), "The hash should start with the correct number of leading zeros."

if __name__ == "__main__":
    print("\nRunning Test 5: Proof of Work")
    test_proof_of_work()
```

✓ 6.4s

```
Running Test 5: Proof of Work
Difficulty: 2, Mined Hash: 001ad1d70e7c61989c161a23c186b5608287f019f2cf6c12160fcb34def9b000
Difficulty: 4, Mined Hash: 0000b5548afc710c6161527291b8b191f2f40e072848db3ae52b175cc41b8e2e
Difficulty: 5, Mined Hash: 000008d243a9f8f4512e8bd4c90e3698b2289db05dce25993059ad048fcf7a31
```

Capture 8 *: Test de la Preuve de travail*

Dans notre code la difficulté choisie est 4 car elle est plus ou moins compliquée à trouver et ne nécessite pas assez de temps.

Test 5: Ajouter plusieurs blocs avec des transactions et valider la chaîne.

```python
def test_add_block_and_validate():
    '''Add a new block and validate the chain'''
    blockchain = Blockchain()

    print("\nBlockchain before adding a block:\n", blockchain.display_chain())

    transaction = Transaction("Alice", "Bob", 100)
    new_block = Block(index=1,
                      timestamp=datetime.now().strftime("%d/%m/%Y %H:%M:%S %f"),
                      transactions=[transaction.to_json()],
                      previous_hash="")
    blockchain.add_block(new_block, difficulty=4)

    t0 = Transaction("Monica", "Toto", 500)
    t1 = Transaction("Felix", "Cherif", 1000)

    new_block_2 = Block(index=2,
                      timestamp=datetime.now().strftime("%d/%m/%Y %H:%M:%S %f"),
                      transactions=[t0.to_json(), t1.to_json()],
                      previous_hash="")
    blockchain.add_block(new_block_2, difficulty=4)

    print("\nBlockchain after adding a block:\n", blockchain.display_chain())

    assert blockchain.is_chain_valid() == True, "Blockchain should be valid after adding a block."

if __name__ == "__main__":
    print("\nRunning Test 7: Add Block and Validate Chain\n")
    test_add_block_and_validate()
```

✓ 1.1s

Capture 9 *: Test d'ajout de blocs et de validation de la chaine*

```
Running Test 7: Add Block and Validate Chain


Blockchain before adding a block:
 {
    "index": 0,
    "timestamp": "07/01/2025 21:34:28 392777",
    "transactions": [],
    "hash": "0d9a5f65bbe7b3fb5560fd83e4c423e253d6b21174ed3a6b45e5e55e6cf6d96b",
    "previous_hash": "0",
    "nonce": 0
}

Blockchain after adding a block:
 {
    "index": 0,
    "timestamp": "07/01/2025 21:34:28 392777",
    "transactions": [],
    "hash": "0d9a5f65bbe7b3fb5560fd83e4c423e253d6b21174ed3a6b45e5e55e6cf6d96b",
    "previous_hash": "0",
    "nonce": 0
}
{
    "index": 1,
    "timestamp": "07/01/2025 21:34:28 392777",
    "transactions": [
        {
            "sender": "Alice",
            "receiver": "Bob",
            "amount": 100
        }
    ],
    "hash": "00001bb8355b61181bcab3a3e212aa3be1b833b4077994681062c66ed96c1c66",
    "previous_hash": "0d9a5f65bbe7b3fb5560fd83e4c423e253d6b21174ed3a6b45e5e55e6cf6d96b",
    "nonce": 171307
}
{
    "index": 2,
    "timestamp": "07/01/2025 21:34:29 099544",
    "transactions": [
        {
            "sender": "Monica",
            "receiver": "Toto",
            "amount": 500
        },
        {
            "sender": "Felix",
            "receiver": "Cherif",
            "amount": 1000
        }
    ],
    "hash": "0000459e0d83af4c53c1541c708b211dfdfbbd866f9091774ce2ba25d4d46340",
    "previous_hash": "00001bb8355b61181bcab3a3e212aa3be1b833b4077994681062c66ed96c1c66",
    "nonce": 87044
}
```

Capture 10 *: Résultat du test d'ajout de blocs*

Test 6 : Modifier manuellement un bloc et vérifier que la chaîne devient invalide

```python
def test_tampered_block():
    '''Tamper with a block and validate the chain'''
    blockchain = Blockchain()

    transaction = Transaction("Alice", "Bob", 100)
    new_block = Block(index=1,
                      timestamp=datetime.now().strftime("%d/%m/%Y %H:%M:%S %f"),
                      transactions=[transaction.to_json()],
                      previous_hash="")
    blockchain.add_block(new_block, difficulty=2)

    print("\nBlock validity before tampering:", blockchain.is_chain_valid())

    # Tamper with the block

    blockchain.chain[1].transactions.append({"sender": "Eve", "receiver": "Mallory", "amount": 200})

    print("\nBlock validity after tampering:", blockchain.is_chain_valid())

    assert blockchain.is_chain_valid() == False, "Tampered blockchain should be invalid."

if __name__ == "__main__":
    print("\nRunning Test 8: Tampered Block Validation")
    test_tampered_block()
```

✓ 0.0s

```
Running Test 8: Tampered Block Validation

Block validity before tampering: True

Block validity after tampering: False
```

Capture 11 *: Test de la validité d'une chaîne altérée*

Test 7 : Résoudre conflit entre deux chaînes concurrentes avec des longueurs différentes

```python
def test_resolve_conflicts():
    '''Resolve conflicts by replacing the chain with a Longer valid chain'''
    blockchain = Blockchain()
    other_chain = Blockchain()

    # Add a block to the original chain
    transaction = Transaction("Alice", "Bob", 100)
    new_block = Block(index=1,
                      timestamp=datetime.now().strftime("%d/%m/%Y %H:%M:%S %f"),
                      transactions=[transaction.to_json()],
                      previous_hash="")

    blockchain.add_block(new_block, difficulty=4)

    # Add two blocks to the other chain
    other_block = Block(index=1,
                        timestamp=datetime.now().strftime("%d/%m/%Y %H:%M:%S %f"),
                        transactions=[{"sender": "John", "receiver": "Jane", "amount": 50}],
                        previous_hash="")
    other_chain.add_block(other_block, difficulty=4)

    other_block_2 = Block(index=2,
                          timestamp=datetime.now().strftime("%d/%m/%Y %H:%M:%S %f"),
                          transactions=[{"sender": "Marie", "receiver": "Ange", "amount": 90}],
                          previous_hash="")
    other_chain.add_block(other_block_2, difficulty=4)

    # Resolve conflicts
    # Original chain before resolving conflicts
    print("\nOriginal Chain Length:", len(blockchain.chain))
    print("\nOther Chain Length:", len(other_chain.chain))

    blockchain.resolve_conflicts(other_chain.chain)

    # Original chain after resolving conflicts
    print("\nNew Chain Length:", len(blockchain.chain))

    assert len(blockchain.chain) == len(other_chain.chain), "Blockchain should adopt the longer valid chain."

if __name__ == "__main__":
    print("\nRunning Test 9: Resolve Conflicts")
    test_resolve_conflicts()
```

✓ 1.0s

```
Running Test 9: Resolve Conflicts

Original Chain Length: 2

Other Chain Length: 3

New Chain Length: 3
```

Capture 12 : *Test de résolution de conflits*

Conclusion

Le prototype de blockchain simplifiée démontre avec succès les concepts fondamentaux des blockchains, tels que la création de blocs, la validation des transactions, la preuve de travail, et la gestion des conflits. Grâce à une architecture rigoureuse, chaque composant a été conçu pour fonctionner de manière cohérente et efficace. Les tests effectués confirment la robustesse et la fiabilité de l'implémentation, attestant que le système répond aux exigences de sécurité et d'intégrité des données. Les fichiers du code source 'prototype.py' et du code test 'test.py' sont accessibles sur Github via le lien : https://github.com/Warisse88/Simplified_Blockchain_Prototype_ESMT_2024.git

Bibliographie et webographie

Patrick Moret (2024, 15 mai), Développer une Blockchain avec Python : Construire l'Avenir Décentralisé, https://medium.com/developper-une-blockchain-avec-python

Plateforme de ressources informatiques et des défis de codage pour les programmeurs et les passionnés de technologie, https://www.geeksforgeeks.org/create-simple-blockchain-using-python/

Lionnel P. DOOKO (ESMT 2024 – 2025), Cours de Blockchain et Cryptomonnaies, https://www.canva.com/cours-blockchain-cryptomonnaies-esmt-2024-2025

Plateforme d'apprentissage en ligne, https://openclassrooms.com/fr/courses/7155841-testez-votre-projet-python/7414156-ajoutez-des-tests-avec-pytest

Plateforme d'hébergement , https://github.com/dvf/blockchain/blob/master/blockchain.py