

# Implémentation d'une blockchain

Ayoub Anhal

9 novembre 2025

## Table des matières

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Partie I — Blockchain Monoposte</b>	<b>2</b>
2.1	Structure du bloc (Section 1.1)	2
2.2	Proof of Work (Section 1.2)	2
2.3	Construction de la blockchain (Section 1.3)	2
2.4	Système de transactions (Section 1.4)	2
2.5	Validation et sécurité (Section 1.5)	3
2.6	API REST FastAPI (Section 1.6)	3
2.6.1	Architecture	3
2.6.2	Tests automatisés dans le notebook	3
2.7	Statistiques et analytics (Section 1.7)	4
<b>3</b>	<b>Partie II — Décentralisation (Simulation)</b>	<b>4</b>
3.1	Architecture multi-nœuds (Section 2.1)	4
3.2	Communication inter-nœuds (Section 2.2)	4
3.3	Synchronisation et résolution de conflits (Section 2.3)	4
3.4	Scénarios avancés (Section 2.4)	4
3.5	Consensus distribué simplifié (Section 2.5)	4
3.6	Métriques réseau (Section 2.6)	5
<b>4</b>	<b>Section Finale et synthèse</b>	<b>5</b>
4.1	Rapport consolidé	5
4.2	Enseignements clés	5
<b>5</b>	<b>Perspectives</b>	<b>5</b>
<b>6</b>	<b>Annexes</b>	<b>6</b>
6.1	Informations sur l'environnement	6
6.2	Commandes utiles	6

# 1 Introduction

Ce cahier synthétise les résultats obtenus dans le notebook `TP_controle.ipynb` dans le cadre du TP « Implémentation d'une blockchain ». Chaque section reprend les objectifs, les méthodes mises en oeuvre et les principaux résultats observés lors de l'exécution des cellules du notebook. L'ensemble du travail a été réalisé avec Python 3.12 et s'appuie sur des bibliothèques standards pour la cryptographie (SHA-256), la validation de données et l'exposition d'une API REST via FastAPI.

## 2 Partie I — Blockchain Monoposte

### 2.1 Structure du bloc (Section 1.1)

- Le bloc est implémenté sous forme de `@dataclass` avec les attributs essentiels : `index`, `timestamp`, `transactions`, `previous_hash`, `nonce` et `hash`.
- La méthode `calculate_hash()` sérialise le bloc au format JSON canonique avant d'appliquer SHA-256, garantissant l'immuabilité.
- Résultat observé : toute modification d'une transaction (ex. passage de 10 à 15 unités) modifie immédiatement le hash du bloc, vérifiant l'intégrité cryptographique.

### 2.2 Proof of Work (Section 1.2)

- L'expérience varie la difficulté (`difficulty` = 2 puis 3) et mesure le nombre d'itérations nécessaires pour trouver un hash valide.
- Résultats typiques :
  - Difficulté 2 : 232 itérations pour un hash commençant par 00.
  - Difficulté 3 : 448 itérations pour un hash commençant par 000.
- Interprétation : l'augmentation de la difficulté accroît exponentiellement le coût de calcul du minage.

### 2.3 Construction de la blockchain (Section 1.3)

- Instanciation d'une classe `Blockchain` avec un bloc genesis miné et une file de transactions en attente.
- Minage d'un premier bloc (`miner_1`) avec difficulté 3 : le bloc index 1 est accepté et la validité de la chaîne (`True`) est confirmée via `is_chain_valid()`.
- Les soldes résultants montrent l'utilisation de la récompense de minage (`MINING_REWARD` = 50).

### 2.4 Système de transactions (Section 1.4)

- Définition d'une classe `Transaction` pour encapsuler les transferts (validation des champs et montant positif).
- Cas d'usage : trois transactions (Alice, Bob, Charlie) sont ajoutées et minées par `miner_2`.

- Résultats : les soldes finaux confirment la comptabilité simple (voir Table 1).

TABLE 1 – Soldes après minage (Section 1.4)

Adresse	Solde
Alice	-6
Bob	3
Charlie	3
Miner_2	50

## 2.5 Validation et sécurité (Section 1.5)

- Trois scénarios ont été évalués : chaîne intacte, altération d’une transaction, falsification du lien de chaînage.
- Résultats :
  - `Scenario intact`  $\rightarrow$  `True`.
  - `Scenario tampered_data`  $\rightarrow$  `False`.
  - `Scenario tampered_link`  $\rightarrow$  `False`.
- Conclusion : les contrôles d’intégrité et de preuve de travail détectent correctement les manipulations.

## 2.6 API REST FastAPI (Section 1.6)

### 2.6.1 Architecture

- Mise en place d’un service FastAPI (`Educational Blockchain API`) avec un middleware CORS.
- Modèles de payload Pydantic (`TransactionPayload`, `MinePayload`) pour valider les entrées.
- Endpoints exposés :
  - `GET /health` : statut du service.
  - `GET /chain` : export complet de la chaîne.
  - `POST /transactions` : ajout d’une transaction en file d’attente.
  - `POST /mine` : minage des transactions en attente avec adresse de récompense.
  - `GET /balance/{address}` : requête de solde.

### 2.6.2 Tests automatisés dans le notebook

- Utilisation de `TestClient` (FastAPI) pour simuler un flux complet : santé, transaction, minage, lecture du solde et aperçu de la chaîne.

- Exemple de réponse pour `POST /mine` : succès avec hash `00079ef95462...` et deux transactions intégrées (transaction utilisateur + récompense).
- Le hachage du bloc genesis reste accessible via l'appel `GET /chain`.

## 2.7 Statistiques et analytics (Section 1.7)

- Fonction `get_blockchain_stats()` calculant : nombre de blocs (3), nombre total de transactions (7), adresses les plus actives (market, system, alice) et itérations moyennes de minage (1408.5).
- Ces statistiques confirment la progression de la chaîne et l'activité du mineur récompensé.

# 3 Partie II — Décentralisation (Simulation)

## 3.1 Architecture multi-nœuds (Section 2.1)

- Definition d'une classe `Node` encapsulant une blockchain locale et une liste de pairs.
- Trois nœuds (`node_0`, `node_1`, `node_2`) interconnectés.
- Résultat : chaque nœud connaît ses deux pairs, préparant la communication inter-nœuds.

## 3.2 Communication inter-nœuds (Section 2.2)

- Classe `Network` pour diffuser des blocs et synchroniser des transactions.
- Diffusion d'un bloc miné à difficulté 2 : deux nœuds reçoivent un bloc supplémentaire, montrant l'effet de propagation.

## 3.3 Synchronisation et résolution de conflits (Section 2.3)

- Fonction `resolve_conflicts()` appliquant une règle de chaîne la plus longue.
- Scénario : `node_1` supprime son dernier bloc puis résout les conflits en se réalignant sur ses pairs (`True`, chaîne de longueur 2).

## 3.4 Scénarios avancés (Section 2.4)

- **Partition réseau** : Isolement de `node_2`, minage indépendant, chaîne locale de longueur 3.
- **Nœud malveillant** : Tentative d'injection d'un bloc invalide (hash incorrect) détectée (`False` sur les pairs).
- **Double dépense** : Ajout de transactions concurrentes ; la chaîne mine un bloc contenant des transactions en doublon, illustrant la nécessité d'un consensus plus robuste.

## 3.5 Consensus distribué simplifié (Section 2.5)

- Vote par longueur de chaîne : dictionnaire de votes `{1:1, 4:2}`.
- Consensus obtenu sur la longueur 4, privilégiant les chaînes plus longues.

### 3.6 Métriques réseau (Section 2.6)

- Calcul des indicateurs : nombre de nœuds (3), longueur moyenne des chaînes (3.0), transactions en attente (5).
- Ces métriques permettent de suivre l'état global du réseau simulé.

## 4 Section Finale et synthèse

### 4.1 Rapport consolidé

- La fonction `run_full_test_suite()` agrège :
  - Validité actuelle de la blockchain principale (`False` après insertion de transactions supplémentaires, soulignant l'intérêt d'une vérification continue).
  - Statistiques détaillées (Section 1.7).
  - Métriques réseau (Section 2.6).
  - Votes de consensus (Section 2.5).
- Un « playground interactif » montre l'ajout dynamique d'une transaction aléatoire et le minage associé (`play_miner` reçoit 50 unités).

### 4.2 Enseignements clés

1. **Intégrité** : Le chaînage cryptographique protège efficacement contre les altérations simples.
2. **Preuve de travail** : Le réglage de la difficulté influe directement sur le coût et le temps de minage.
3. **Transactions et récompenses** : Le système de récompense est correctement appliqué, dopant le solde du mineur.
4. **API REST** : L'exposition via FastAPI fournit une interface moderne et testable pour interagir avec la blockchain.
5. **Décentralisation simulée** : Les mécanismes de diffusion, de résolution de conflits et de consensus illustrent les défis d'un réseau distribué.

## 5 Perspectives

- Intégration de signatures numériques (ECDSA) pour authentifier les transactions.
- Mise en place d'une difficulté dynamique répondant au temps cible de minage.
- Implémentation d'un consensus plus robuste (ex. Proof of Stake, Raft ou PBFT) pour réduire les risques de double dépense.
- Déploiement réel du service FastAPI couplé à une base de données ou à un stockage persistant pour les blocs.

## 6 Annexes

### 6.1 Informations sur l'environnement

- Version Python : 3.12.7 (environnement virtuel conseillé).
- Dépendances principales : `fastapi`, `uvicorn[standard]`, `pydantic`.
- Les instructions d'installation détaillées sont disponibles dans `GUIDE.md`.

### 6.2 Commandes utiles

Commande	Description
<code>python3 -m venv .venv</code>	Création de l'environnement virtuel.
<code>source .venv/bin/activate</code>	Activation de l'environnement virtuel.
<code>pip install -r requirements.txt</code>	Installation des dépendances FastAPI et utilitaires.
<code>uvicorn main_api:app -reload</code>	Lancement de l'API REST en local.
<code>jupyter notebook</code>	Exécution interactive du notebook pour reproduire les résultats.

Ce document a été généré à partir des résultats du notebook `TP_controle.ipynb`.  
Dernière mise à jour : 9 novembre 2025.