COMPOSANT 2 : BLOC

Ahmed DIENG

version 1.0

1- Description

a- Contexte

Une blockchain peut être décrite comme conteneur de donnée qui contient l'ensemble des transactions qui ont eu lieu depuis sa création.

Le but de ce projet est de réaliser une blockchain en gérant séparément ses différentes composant : Transaction, Bloc, Blockchain, Hacheur SHA-256 et Signature.

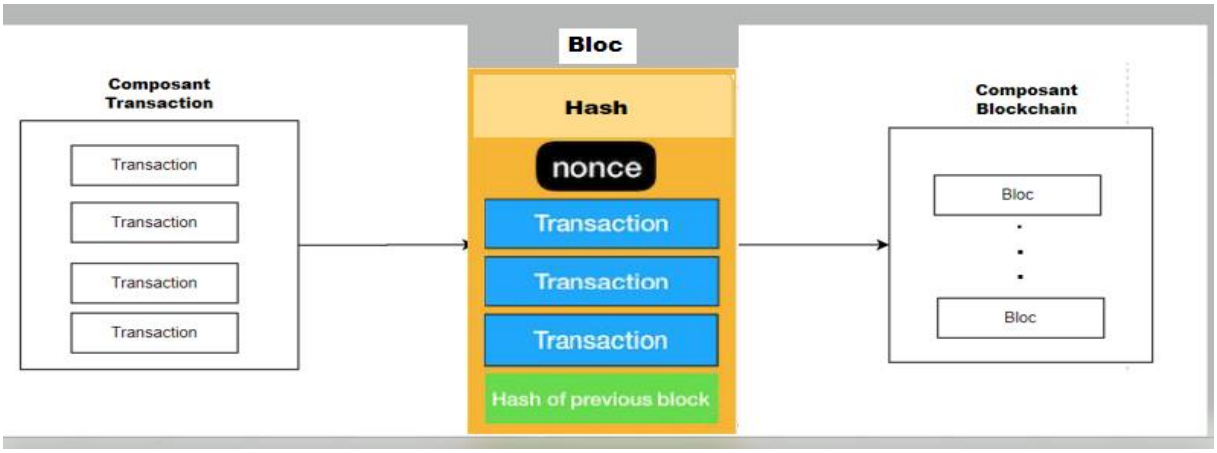
L'objectif de ce document est de rédiger les spécifications du composant **Bloc.**

Un bloc est une structure contenant un numéro de bloc, des transactions, un nombre arbitraire Nonce et son identifiant appelé **hash**.

Un bloc est donc ensemble de transactions. Un ensemble de bloc lié les uns des autres forment une Blockchain.

Un bloc combine ses transactions pour être défini de manière unique par un hachage (calculé par le Hacheur SHA-256) qui lui sert d'identifiant. Le bloc communique sa liste de transactions ainsi que son hachage au bloc suivant sur la chaine. Dans le bloc suivant un hachage sera créé en prenant en compte les nouvelles transactions.

b- Schéma bloc incluant les composants connexes

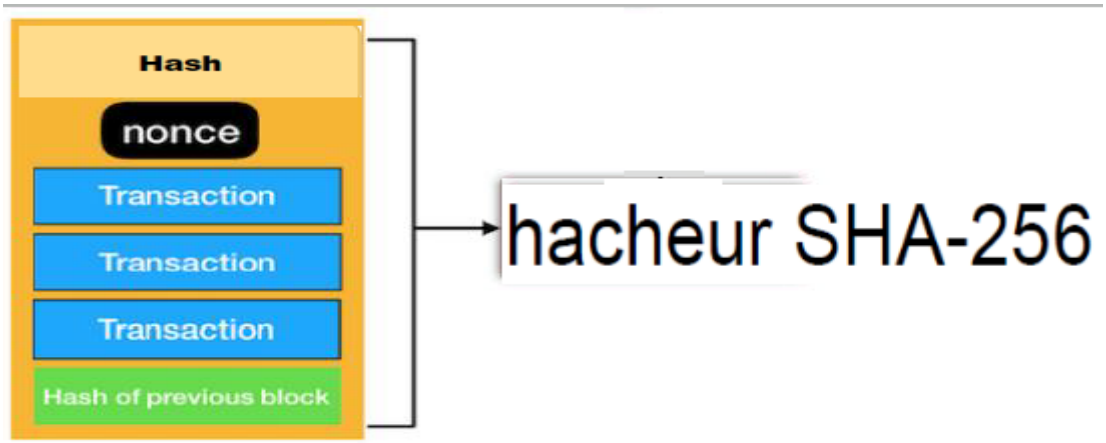


Les transactions sauvegardées sont envoyées dans un bloc pour y constituer un ensemble unique identifié par l'attribut **hash** du bloc.

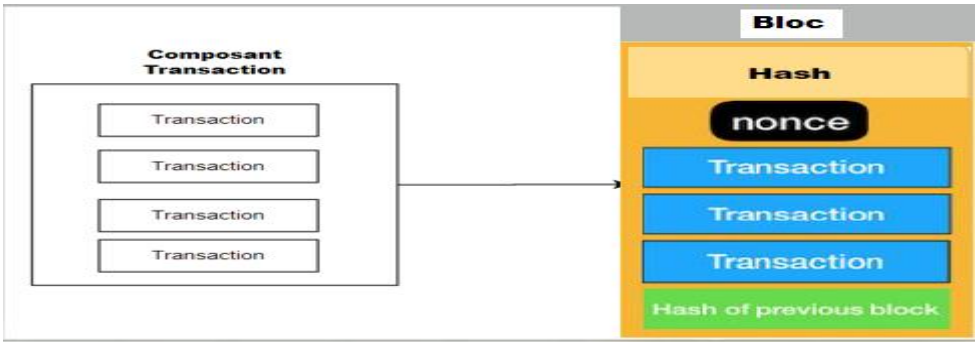
Le composant Blockchain est constitué de plusieurs blocs.

c- Interface et interaction avec chaque autre composant

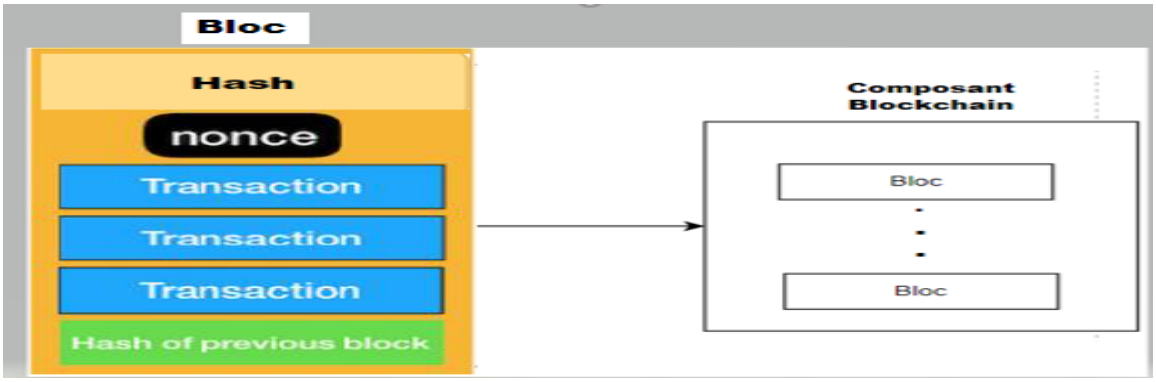
- Le **hacheur SHA-256** calcule le **Hash** du bloc en fonction de ses attributs (la liste de transactions)



- Une liste de transactions est envoyée vers le bloc grâce à la méthode de sauvegarde **to_json()** d'une transaction.



- Un ensemble de blocs est envoyé vers le composant blockchain



d- Résumé : déclarations de fonctions python d'interface et leurs arguments

Bloc(const nlohmann::json &j):

C'est le constructeur de la classe Bloc. En entrée on a un document json qui contient les champs :

- numéro
- liste de transactions
- la transaction du mineur
- un Integer arbitraire Nonce
- deux chaines de caractères correspondant à hash et previous_hash.

On construit un bloc avec les champs du json.

py::object_to_json() :

Permet de sauvegarder un bloc et retourne un json.

void setNonce(int) :

Modifier l'attribut Nonce d'un bloc.

unsigned int getNonce();

Récupère le Nonce d'un bloc.

bool validationDifficultyBloc():

La fonction validationDifficultyBloc permet de valider le hash d'un bloc.

La règle de validation choisie est que l'attribut **hash** du bloc doit être un string de 64 caractères commençant par un nombre minimal de zéros entré en paramètre de la fonction.

La fonction retourne **True** si la condition est vérifiée et **False** sinon.

void computeHash():

Evaluation du hash bu bloc en appelant la méthode du composant Sha256.

bool validationBloc():

Un bloc est validé si :

- son hash est validé et
- son hash ne change pas si on applique à nouveau la méthode computeHash.

e- Cas d'erreurs

Numéro du bloc n'est pas un entier positif : message d'erreur.

Un champ manque dans le json : exception python.

La liste de transaction est vide : message d'erreur.

Le hash ou le previous_hash n'est pas un string de 64 caractères : message d'erreur.

2- Test

Nous allons tester la sauvegarde d'un bloc (méthode to_json()). On utilise comme exemple le bloc_0. On construit un json avec les champs contenant les valeurs des attributs.

Nous allons implémenter une méthode pour tester la fonction de validation du hash d'un bloc (validationDifficultyBloc).

Le test sera fait pour le cas d'un string représentant le hash d'un bloc valide et le test devra retourner **True**.

Puis pour le cas d'un string de taille 64 mais qui ne respecte pas la règle de validation (string qui commence par un nombre de 0 < au nombre spécifié par la règle). Dans ce cas le test devra retourner **False**.

Enfin on testera avec un string de **taille != 64** et retourner un **message d'erreur**.