Créez votre propre dystopie avec une monnaie numérique

Projet avancé - Licence Informatique UPS

Vincent Dugat

9 septembre 2022

Table des matières

Vous êtes invités, pour bien profiter de ce document, à :

1. Vous détendre,
2. Lire le document jusqu'au bout (éventuellement en plusieurs fois)
3. Répéter le mantra : « c'est moins compliqué que ça en a l'air »

1 Présentation

La BCE va mettre en place un euro numérique à partir de 2024. Il y a encore peu d'informations techniques qui circulent à son sujet. On trouve par contre beaucoup d'analyses, de prédictions, de craintes, etc. Au début on parlait d'une crypto-monnaie basée sur une blockchain. Il semble que cela sera plutôt une monnaie numérique comme l'est déjà l'euro, mais centralisé à la BCE (ou partiellement décentralisée) et qui à terme, pourrait supprimer le cash.

Cette annonce semble susciter des inquiétudes concernant la vie privée, le gel des comptes ou des contrôles des dépenses des usagers, malgré les démentis de la BCE. Peu importe les démentis, lançons nous sans retenue dans la dystopie.

Donc, pour que cela soit un peu drôle, nous allons donc implémenter une monnaie numérique et créer un monde imaginaire pour mettre en oeuvre notre monnaie. Elle sera basée sur une blockchain [1] car cette technologie est intéressante à connaître. Ce sera une blockchain simplifiée de type Bitcoin, mais : centralisée (pas de peer2peer), pas d'anonymat, le seul mineur est la banque centrale, pas de vote ni de consensus. En clair tout est sous contrôle, l'antithèse du Bitcoin.

2 Préambule : principe général d'une blockchain, simplifié pour le projet

La blockchain ou chaîne de blocs est une structure de données informatique qui permet de sécuriser des données, supposées être publiques, donc lisibles sans restriction, contre toute tentative de modification ou altération. Pour cela on utilise un algorithme de hash coding qui permet d'associer à une donnée une valeur numérique unique. Si H est la fonction qui implémente un de ces algorithmes, et x une donnée quelconque, on a :

— $H(x)$ donne toujours la même valeur pour le même x
— $H(x) \neq H(x')$ même si la différence entre x et x' est minime.
— connaissant $H(x)$ est très difficile de retrouver x.

L'algorithme de hash que nous utiliserons est le SHA256 (où $H(x)$ est un nombre binaire de 256 bits). Nous allons manipuler son résultat sous forme d'une chaîne de caractères hexadécimaux. La longueur de la chaîne est fixe (32 octets, donc 64 caractères puisque un nombre hexadécimal code quatre bits). Cet algorithme n'est pas à programmer, nous allons utiliser des programmes de bibliothèques.

1. Ce que ne sera vraissemblablement pas l'euro numérique.

La blockchain est constituée de blocs d'information chaînés entre eux sous forme de liste. Jusque là ça reste une SD très classique en informatique. Pour éviter qu'un bloc disparaisse sans laisser de trace, on va ajouter (signer) chaque bloc avec le hash code de son contenu. Et pour qu'il laisse une trace en cas de suppression dans la chaîne, on va inclure dans le contenu du bloc, le hash code de son prédécesseur. Et donc, le hash code du bloc va intégrer le hash code du bloc précédent dans la chaîne.

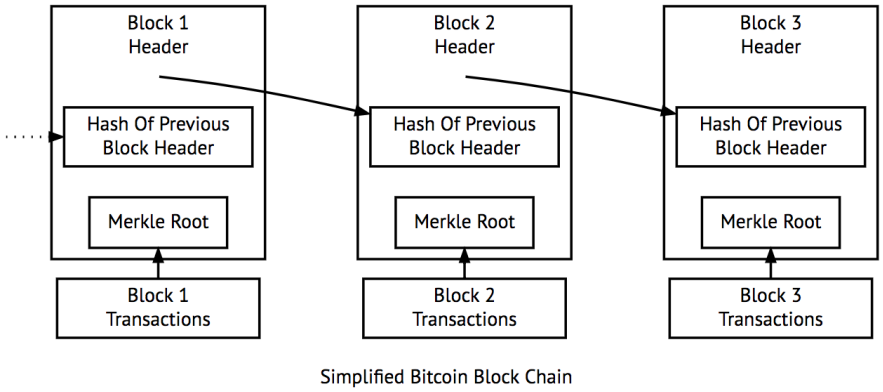

FIGURE 1 – La blockchain

Que se passe-t-il si on veut supprimer un bloc quelque part au milieu de la chaîne ? De toute évidence on perturbe la continuité des hash codes. Il faut donc recalculer le hash de tous les blocs suivant celui qui disparaît et refaire le chaînage. La sécurité de la chaîne tient ici à la quantité de travail à faire pour altérer la chaîne.

Même sur un grand nombre de blocs, on peut imaginer pouvoir le faire avec un gros PC. Ça reste jouable me direz-vous. On va donc pimenter le tout avec la "difficulté". Pour faire simple, disons que la difficulté est un critère que doit remplir le hash code et qui assure qu'il n'est pas trop facile à calculer même avec une machine performante. Dans la pratique (Bitcoin) la difficulté est adaptée à l'évolution de la puissance des machines. Elle est choisie pour qu'il faille toujours 10mn pour la calculer. A itérer autant de fois qu'il y a de blocs à recalculer pour altérer la chaîne sans que cela ne se voit. Comme un nouveau bloc arrive toutes les 10mn, il faudrait recalculer l'ensemble de la chaîne en 10mn pour que cela passe inaperçu (ou presque car il y a aussi d'autres mécanismes dans un univers peer2peer où il y a autant de copies de la chaîne que de noeuds dans le réseau).

Nous allons simplifier ce mécanisme en choisissant une difficulté fixe représentée par un entier d, et en demandant que le hash du bloc commence par d zéros dans sa représentation en ascii hexadécimal. Et si cela n'est pas le cas ?

2.1 Le minage

On introduit dans le contenu de chaque bloc un entier initialisé à zéro et appelé "nonce". Si le hash du bloc (incluant la nonce) ne satisfait pas le critère, on incrémente la nonce et on recommence le calcul du hash. On répète ce processus jusqu'à ce que le hash satisfasse le critère (la difficulté). Ce calcul est appelé «minage» [mining].

2.2 Contenu des blocs

Que met-on dans les blocs ? Ce que l'on veut en fait. Le Bitcoin gère des transactions (tx for short) d'argent (crypto-monnaie [crypto-currency]). Nous allons imiter ce principe. Donc il y aura des transactions du genre «Astérix envoie 10 unités d'argent à Obélix». Il y a un nombre variable de transactions dans un bloc

qui est normalement (Bitcoin) dépendant de la taille maximale du bloc et de celle des transactions. Nous allons fixer arbitrairement le nombre maximal de transactions (constante du programme). De plus, le bloc contient la date de sa création (timestamp). La structure des transactions pour ce projet sera développée plus loin dans le texte.

Le premier bloc de la chaîne est particulier. Il ne peut pas intégrer le hash de son prédécesseur car il n'a pas de prédécesseur. On l'appelle bloc «génésis». Le hash de son prédécesseur est zéro, sa nonce reste à zéro, il a une unique transaction réduite à une création de monnaie [2]. Par contre il a bien un hash. Il n'est pas nécessaire de le miner.

Que se passe-t-il si on veut altérer une transaction d'un bloc sans toucher aux hash de ce bloc ?

2.3 Arbre de Merkle [Merkle tree]

Si on altère une transaction d'un bloc sans toucher aux hash de ce bloc, le hash du bloc va quand même être modifié puisque qu'il prend en compte le contenu de tout le bloc, et la liste des transactions fait partie du contenu (pour notre sujet car le Bitcoin est légèrement différent).

Pour corser le tout on va calculer l'arbre de Merkle des hash des transactions :

1. supposons que nous ayons quatre transactions : $t1, t2, t3, t4$,
2. on calcule les hash de chaque transaction. On obtient $h1, h2, h3, h4 = H(t1), H(t2), H(t3), H(t4)$
3. puis on calcule le hash de $h1$ concaténé avec $h2$ et le hash de $h3$ concaténé avec $h4$. Soit $h1.2$ et $h3.4$ les résultats,
4. puis on calcule $H(h1.2$ concaténé $h3.4)$, soit $h1.2.3.4$ le résultat.

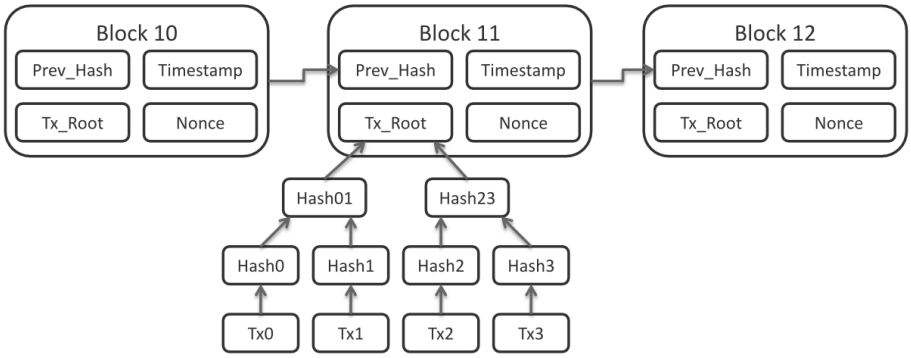

FIGURE 2 – Merkle tree

Le résultat final $h1.2.3.4$ est le racine de l'arbre [Merkle tree root]. C'est un hash code. Ce hash sera inclus dans le contenu du bloc avant de calculer le hash du bloc.

Si le nombre de transactions (ou de hash intermédiaires) est impair, on fait remonter le hash de cette transaction solitaire (ou le hash intermédiaire) au niveau supérieur. Une autre solution est de dédoubler la transaction solitaire comme montré dans la figure 3.

Connaissant les transactions qui sont écrites en clair on peut recalculer leur hash et donc la racine de l'arbre, et vérifier que c'est bien le même hash root que celui qui apparaît dans le bloc, et que donc, aucune transaction n'a été altérée.

2. Pour reprendre le nom donné à ce block par Satoshi Nakamoto (聡中本) l'inventeur du Bitcoin.

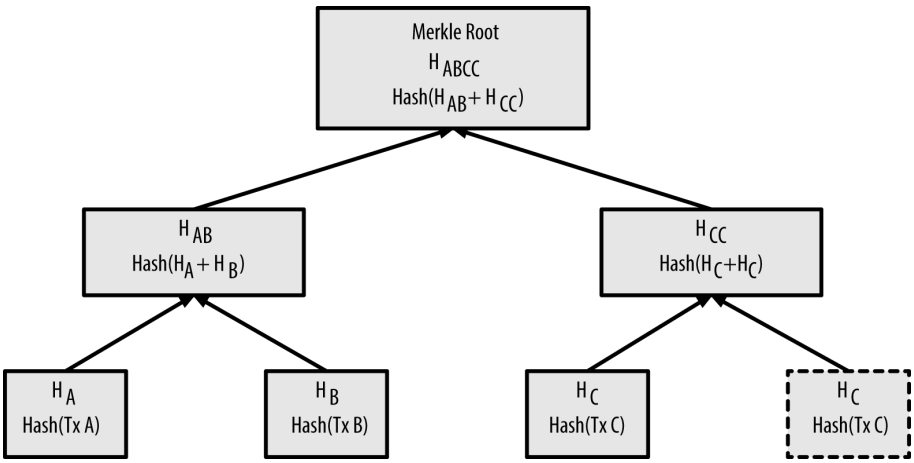

FIGURE 3 – MerkleTree impair

2.4 Sécurité de la blockchain

Quelques attaques (dans la cas de notre blockchain simplifiée) :

— Supprimer ou modifier une transaction : Cela modifie le merkleroot et donc le sha du block et provoque une inconsistance de la chaîne.

— Rejouer une transaction déjà présente. Normalement le timestamp et le numéro de tx doit empêcher ça car ils sont dans le hash de la signature. Il faudrait modifier ces données. De plus cela modifie le merkle tree. Le test de hash root le détecte.

— Agir sur les blocks. Supprimer un bloc pour faire disparaître une transaction. Cela supprime aussi les autres transactions. Et le chaînage de hash est perturbé. Il faut tout recalculer.

2.5 Les adresses et les transactions

Pour créer une transaction, c'est à dire pour envoyer de l'argent, il faut avoir une adresse. Les adresses sont liées à deux clefs de cryptographie : une clef publique et une clef secrète. Comme son nom l'indique, la clef publique est faîte pour être diffusée. Elle sert à vérifier les signatures. La clef secrète ne doit être connue que de son propriétaire. Elle sert à signer les transactions. Cette partie sera simplifiée car ce serait un sujet dans le sujet. Chaque transaction intègre, de plus, une date (timestamp), et un hash.

2.6 Vérification

L'intégrité de la blockchain peut être vérifiée à tout moment.

1. On vérifie que la chaîne commence bien par le bloc génésis et que le chaînage des hash est valide, et que le hash du bloc est bien celui annoncé (vérification 1).
2. On vérifie pour chaque bloc que le hash Merkle root correspond bien aux transactions de ce bloc (vérification 2).
3. On vérifie quand on ajoute une nouvelle transaction que la signature est valide (voir Niveau 2), et que la structure des transactions et du nouveau bloc est valide.

2.7 Qui veut la peau de la blockchain ?

Bien sûr notre blockchain n'est pas inaltérable, ni celle du Bitcoin d'ailleurs. Puisque que la nôtre est centralisée on peut faire ce que l'on veut avec ce qui n'est pas le cas de celle du Bitcoin qui est répartie et soumise au principe du consensus (mais c'est une autre histoire).

Pour faire un *cheater*, il faut faire un programme qui altère un bloc et recalcule les hash de tous les blocs impactés le plus vite possible.

2.8 Détail des transactions

Dans cette partie on décrit un système de transaction proche de celui du bitcoin, mais tout de même simplifié. Chaque transaction de la liste des transactions contenue dans le bloc, contient une liste d'entrées (inputs) et de sorties (outputs). Les entrées contiennent les références aux transactions non dépensées qui vont être utilisées pour payer le destinataire. Toute transaction fait apparaître des "avoirs" non encore dépensés au profit du destinataire appelés UTXO [3] (voir plus bas). Les transactions sont alimentés par les avoirs (UTXO) non dépensés de l'émetteur, qui du coup sont dépensés et disparaissent en tant qu'avoir.

En résumé :

— Une transaction est une consommation de utxo en input (via la liste inputs) et une production d'utxo en sortie (via la liste outputs).
— Il y a autant d'utxo d'entrée que nécessaire pour payer l'output correspondant plus les frais
— Une transaction a un timestamp et éventuellement une date de péremption, et une catégorie
— les utxo de sortie héritent de la date de péremption et de la catégorie de la tx

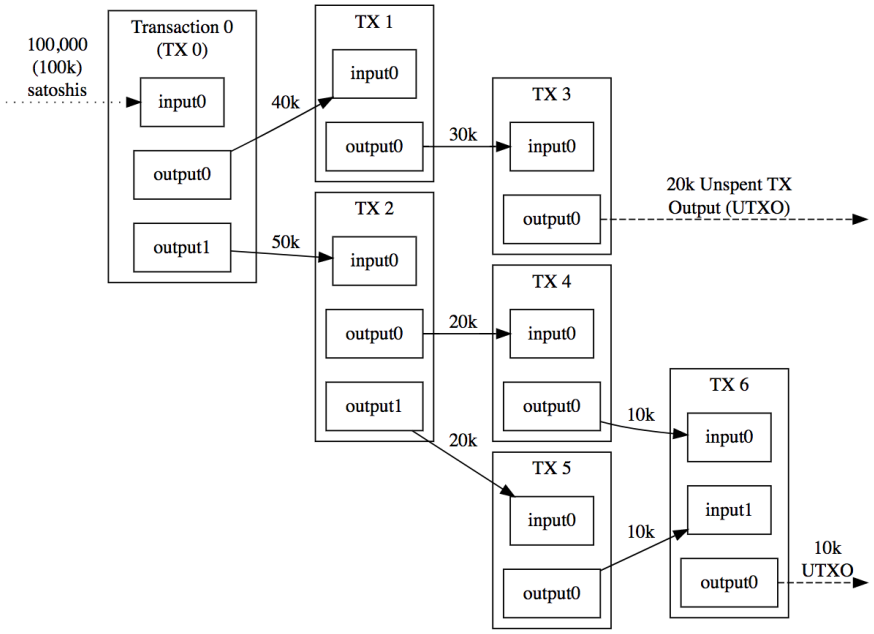

FIGURE 4 – Relations entre les inputs et outputs dans les chaînes de dépenses

Normalement c'est le portefeuille (wallet) qui tient à jour la liste des UTXO de l'utilisateur à partir d'une liste globale des UTXO de tous les utilisateurs.

Une UTXO utilisée comme source de paiement pour une transaction doit être entièrement dépensée (comme un billet). Si le montant de la transaction est inférieur à celui de l'UTXO utilisée, alors il faut générer une transaction de change de l'émetteur vers lui-même.

Toute tx de sortie (qui est une UTXO tant qu'elle n'est pas dépensée) est verrouillée par un script contenant la signature du propriétaire. Pour être utilisée, il faut que ce dernier, qui est l'émetteur de la

3. Unspent TransaCTion Output. CT transcrit par X

transaction, fournisse un script de déblocage. Les deux scripts sont une suite de commandes et de données écrites en notation polonaise inverse et interprétées au moyen d'une pile.

2.9 Choix et simplifications pour le projet

— les transactions coinbase [4] (BCE pour le projet) ne contiennent que ça. Il n'y a pas d'input (liste vide), un seul output. C'est la première transaction de la liste du bloc si elle provient du minage d'un échange [5]. Cela peut aussi être la seule transaction du bloc (genesis, helicopter money [6]).
— les tx ordinaires contiennent des inputs (UTXO) et en output, le paiement proprement dit, plus le change éventuel et les frais.
— genesis et helicopter money : une seule transaction vers "state" ou un utilisateur, pas d'inputs, un seul élément dans outputs, c'est la seule transaction du bloc.
— phase de marché : les outputs de chaque tx sont : l'output de la tx proprement dite, plus un output de frais, plus l'éventuel output de change (si besoin).

2.9.1 Langage des scripts de verrouillage/déverrouillage

Les UTXO sont verrouillée par un script de commandes où figure la signature du bénéficiaire. Il faut construire un script de déverrouillage qui va vérifier que l'utilisateur est bien le bénéficiaire (comparaison de signatures).

Par soucis de simplification on ne va pas utiliser de vraie adresse Bitcoin, ni de système de signature de type RSA ou de courbes elliptiques, mais simuler le principe des signatures numériques avec des chaînes de caractères.

Le langage ici ne comprend que quatre instructions :
— DUP : duplique le sommet de pile et l'empile
— EQ : dépile deux fois en récupérant les valeurs et renvoie true si elle elle égales, false sinon.
— HASH : hash le sommet de pile, dépile, et empile le résultat
— VER : vérifie la signature. Cette étape est simplifiée : <tx sign X> et <pubKey Y> VER renvoie true si X = Y, faux sinon

On a en plus des constantes :
— <H(pubKey X)> : le hash de la clef publique de X. Comme on ne gère pas les clefs dans ce projet on gardera la constante sous cette forme.
— <tx sign X> : le hash de la transaction signé avec la clef privée de X. Même remarque que ci-dessus.
— <pubKey X> : la clef publique de X. Même remarque.

où X représente un utilisateur quelconque.

Un script de verrouillage (lock) est de la forme : <Tx sign A><pubKey A> DUP HASH.

Un script de déverrouillage (unlock) est de la forme : <H(pubKey A)> EQ VER

Pour déverrouiller une UTXO, on concatène les scripts : lock+unlock et on l'exécute de la gauche vers la droite avec une pile.

<Tx sign A><pubKey A> DUP HASH<H(pubKey A)> EQ VER

Exemple :

Les figures se lisent chacune du haut vers le bas avec :
— <sig> = <Tx sign X>

4. Nom donné au moteur de gestion du Bitcoin.
5. Cela suppose que pour miner, le mineur commence par créer une nouvelle transaction à son bénéfice (ici au bénéfice de l'état, qu'il ajoute en tête de la liste des transactions du bloc
6. Nom donné aux USA aux aides de l'état pour relancer la consommation.

— <PubK> = <pubKey X>
— HASH160 = HASH
— EQUALVERIFY = EQ
— CHECKSIG = VER

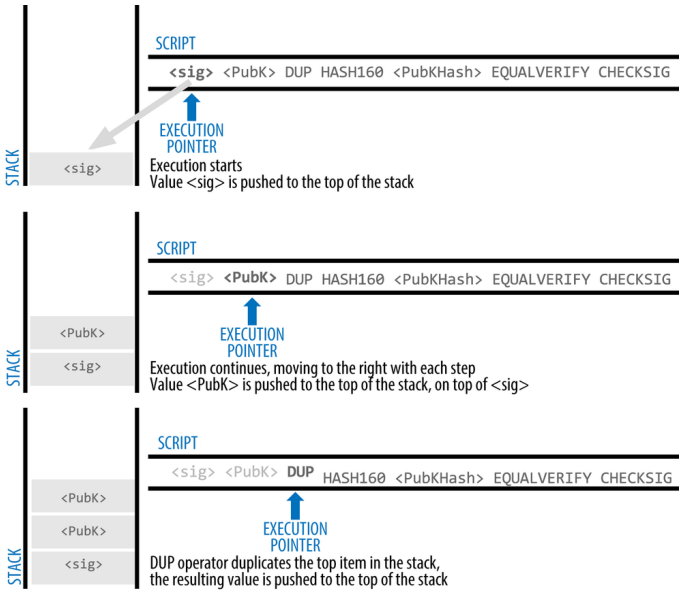

FIGURE 5 – Pile, source : Mastering Bitcoin, Andreas Antonopoulos, O'Reilly

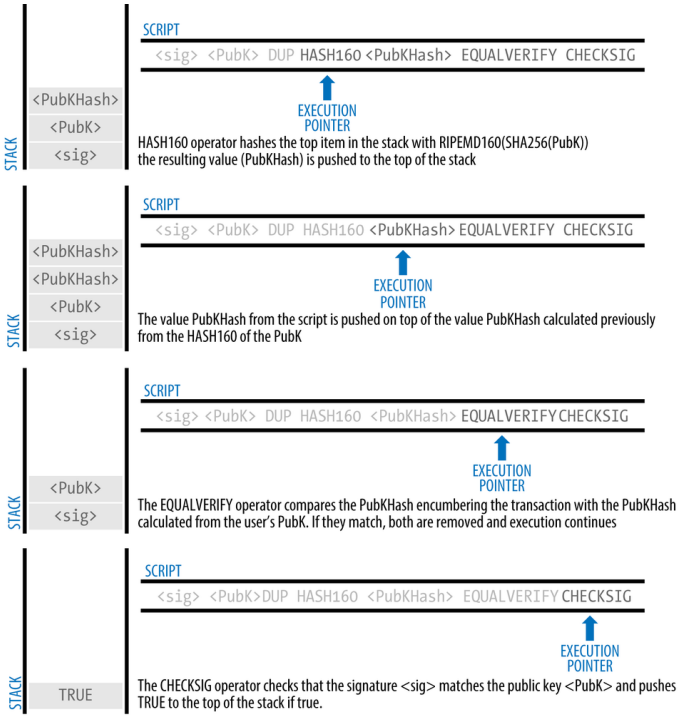

FIGURE 6 – Pile (suite, même source)

Pour comprendre comment ça marche il faut savoir que : seul X a pu signer la tx avec sa clef privée car il est (normalement) le seul à la connaître. En appliquant la clef publique de X à un message qu'il a signé avec sa clef privée, on peut vérifier qu'il en est bien le signataire.

Structure des inputs
— hash de la ou les UTXO qui financent la dépense. Chacune contient un lock script <Tx sign A><pubKey A> DUP HASH
— unlock script : <H(pubKey A)> EQ VER

Structure des outputs
— générer le lock script : <tx sign B><pubKey B> DUP HASH
— action :
  — exec pile lock.unlock de A (le point "." représente l'opérateur de concaténation)
  — si ok supprimer input de utxoList et ajout output dans utxoList
— Pour les récompences : pas d'input, output avec lock script de "state"

2.10 Un point d'économie : la création monétaire

En économie une création monétaire à partir de rien s'appelle *l'inflation*. Il y a augmentation de la masse monétaire, c'est à dire de la quantité d'argent en circulation. Si la création de richesse (production de biens) augmente aussi cela n'est pas un problème. Cela le devient si la richesse stagne car alors la monnaie se déprécie (elle se *débase* et on risque la stagflation = inflation monétaire et récession économique en même temps).

Dans le bitcoin, il y a création monétaire lors de chaque minage car c'est la récompense du mineur pour son travail. C'est le système (coinbase) qui crée cette monnaie en envoyant 50 BTC (au départ mais maintenant c'est 12,5 BTC) au mineur. Ils sont créés ex-nihilo. Tous les 210000 blocs la récompense est divisée par 2 de sorte que l'on peut prévoir quand ce processus de récompense prendra fin et combien de bitcoins auront été créés.

Chaque bloc prend 10 minutes pour être calculé. Donc, 210000 blocs x 10mn = 21x10^5mn = 4 ans, c'est à dire que l'on divise la récompense par deux tout les quatre ans. Quand la récompense atteindra zéro, on aura créé 21 millions de bitcoin et ce sera en 2140. Ce sera la fin de la phase d'inflation. Après cela, plus de création monétaire, et les mineurs ne seront plus récompensés que par les frais de transaction.

Nous passons maintenant plus précisément au sujet du projet.

3 Ze big scénario du monde imaginaire pour se prendre pour un créateur de mondes et utiliser notre monnaie dans un monde parfait

La Sérénissime république fédérale populaire du progrès éclairé (réfépope) est un état qui a pop spontanément [7] dans la cinquième dimension nommée Disto-5.1, qui, elle même, a pop à la suite d'une expérimentation très secrète et très borderline de la base 51 au Nouveau Mexique.

Cet état se veut parfait, et veut mettre en place une société parfaite, et une économie parfaite. Ses habitants sont qualifiés d'« identité » par l'administration étatique, « l'Etat » ou « state », lequel s'est doté de deux institutions :
— une banque centrale : BCE = Banque Centrale Eclairée qui gère les comptes bancaires des usagers, la vérification des autorisations par consultation de BGF (voir ci-dessous), le minage, la masse monétaire.
— une autorité d'autorisation : BGF = Big GodFather qui gère la base de données des droits et non-droits des usagers.

Il y a une monnaie : le Pass (P), et millipass (mP) qui vaut 1/1000 de Pass

La BCE (celle du projet) gère une liste des "identités" et de leur comptes. Comme Etat = BCE = BGF, il n'y a qu'une seule structure de donnée qui gère tout.

7. Et à l'insus du plein gré des expérimentateurs qui se seraient bien passés de cette "bavure".

La BCE met en place sa monnaie à l'aide d'une blockchain : la BCE vérifie les transactions (tx) d'une file d'attente, les complète et mine les blocs après vérification du solde, des dates de péremption des avoirs, et des autorisations auprès de l'Autorité centrale (Big Godfather). La liste des tx validées est incluse dans un nouveau bloc avec celle de la récompence et est minée.

Le minage crée de la monnaie et augmente ainsi la masse monétaire selon un facteur réglable (en variant la rétribution ou récompence). Les jetons (unité de monnaie = Pass) créés sont attribués à l'état. L'état le redistribue sous forme de revenu universel, d'actions diverses que nous ne gèrerons pas ici. Il y a des frais sur les transactions qui sont un impôt qui va à l'état comme la TVA de notre vrai monde.

Les institutions devraient être aussi des usagers aussi mais pour simplifier on ignore ce fait. Les comptes bancaires sont nominatifs, Le BCE connaît l'identité des usagers, leur numéro de compte, leur avoirs, leurs dépenses. Leur numéro de compte est en clair dans la blockchain car l'anonymat et la vie privée ne sont que de la « poussière de fée » pour plagier un tweet de Joe Biden au sujet de la liberté d'expression (22/2/2022) [8]. Les usagers s'échange de l'argent via des transactions qui sont mises en attente dans une file, puis vérifiées et traitées.

Remarque : l'usager « état » n'a pas de limitation et on peut toujours payer l'état même si le compte est bloqué.

limitations génériques : On va en gérer deux :

1. vérification du maximum d'épargne autorisé,
2. vérification de la date de péremption des avoirs.

Algorithme de vérification de BGF :
Une transaction entre X et Y est valide si :
les comptes de X et Y ne sont pas bloqués
&& X ne fait pas partie des émetteurs interdits pour Y,
&& Y ne fait pas partie des destinataires interdits pour X
&& la date de péremption n'est pas dépassée
&& catégorie autorisée en débit pour X et en crédit pour Y

De plus la BCE qui gère les comptes bancaires, vérifie à chaque transaction que le maximum d'épargne n'est pas dépassé pour le bénéficiaire.

Nota Bene :
- Puisqu'une transaction génère potentiellement trois sorties donc trois UTXO, les UTXO doivent hériter de la catégorie de la transaction d'origine, sauf pour les frais qui vont à l'état.
- Il faut prévoir une catégorie «None» signifiant «aucune catégorie», donc permettant l'utilisation pour toute transaction
- Une UTXO est utilisable pour une transaction du point de vue des catégories, si les deux catégories sont les mêmes, ou si la catégorie de l'UTXO est "None", ou «Helicopter» [9]
- Les tx de l'"helicopter money" ont la catégorie "Helicopter"

4 Travail à faire : cahier des charges de programmation

En C, il faut tout écrire. Le développement des structures de données pour la blockchain sera basé sur des *struct* et des pointeurs. Les "identités" seront désignés par des chaînes de caractères "user1" à "userN" où N

8. Free speach in only fairy dust
9. Vu que pour lancer notre économie, nous allons mettre en place une phase d'«helicopter money», il faut bien que cet argent soit utilisable.

est le nombre d'utilisateurs défini comme constante du programme. On y ajoute un utilisateur particulier : "state", l'état.

Dans le Bitcoin, Satoshi a créé une unité spéciale : le satoshi [10] correspondant à $10^{-}8$ bitcoins. Donc un bitcoin correspond à 100,000,000 satoshi. Tous les montants sont convertis en satoshi et ainsi on peut manipuler les sommes comme des *long int* ce qui est vraiment plus pratique.

Nous allons faire pareil avec le Pass, l'unité sera le milliPass : 1 Pass $= 10^3$ milliPass., ainsi nous pouvons gérer les montants comme des entiers, ce qui simplifie beaucoup de choses.

Définir une monnaie n'est qu'une étape, il faut aussi mettre en place une économie. Cela va se passer en plusieurs phases correspondant à des étapes de mise au point du programme :

— *Phase 0 - structures de données* : création des structures de données, test du calcul de sha256, listes pour les transactions, les blocs, les utxo, les inputs et outputs. Ecrire le code d'initialisation, de gestion, etc. Il faut régler ici les allocations dynamiques et tester tout ce code avec soin.

Note : on peut avoir des listes génériques avec des pointeurs void *, par exemple :

```
typedef struct Slist {
    void * info; // info est un pointeur vers une struct blocks ou transactions,
    ce qu'on veut en fait
    struct Slist *next;
} Slist;

```

Ce n'est qu'un exemple et vous êtes libres de vos structures de données.

Liste, pas forcément exhaustive, des SD et tests

1. variables, éventuellement globales, à vous de voir, pour gérer : les comptes des utilisateurs, les utxo, la blockchain.
2. Structure de données pour la blockchain centralisée
3. Structure de données pour les blocs
4. Structure de données pour les listes de transactions avec listes d'entrée, liste de sortie
5. Calcul du hash256 d'un bloc, d'une transaction, d'une liste (de blocs, de transactions, etc.)
6. Minage des blocs
7. Test de validité 1 et 2 (présence du Génésis, les ash des blocks sont cohérents, le chaînage aussi, les racines de Merkle aussi)
8. Calcul de l'arbre de Merkle d'une liste de transactions simplifiées
9. Générateur aléatoire de transactions (choix random d'un émetteur, d'un bénéficiaire, d'une somme)
10. Création de blocs
11. Un *main* qui gère l'ensemble. Pas d'interface demandé, mais il faudra prévoir de générer une blockchain avec choix de difficulté, nombre de blocs et de transactions. Ces paramètres pourront être passés en ligne de commande, ou stockés dans un fichier qui sera lu à l'exécution, ou encore définis avec des #define.
12. Vérification de la cohérence de la blockchain (des hash)
13. Vérification du hash Merkleroot en le recalculant et en le comparant à celui du bloc

    Remarque : Les timestamp sont un autre élément de sécurité permettant de détecter des anomalies dans la blockchain ou dans la liste des transactions. On ne les utilisera que pour gérer les dates de péremption.

10. Nacissic Syndrom.

— *Phase 1 - Mise en place de la monnaie* : écrire le code de la blockchain et un main de test : initialisation de la blockchain, création de blocs et minage, mise à jour de la masse monétaire, comptes et solde des users, genesis, hélicopter money avec intx et outtx tx mais ni récompense, ni frais. A la fin les soldes de tous les utilisateurs sont de 50 Pass si la constante correspondante est de 50 Pass. "State" a également 50 Pass à la suite de la création du bloc "genesis". Il faut également mettre à jour les UTXO de la liste globale et des comptes.

  Donc deux actions principales dans cette phase :

  1. Création du genesis par BCE avec envoi de 50 Pass à State par la BCE qui a le privilège de la création monétaire.
  2. Phase "Helicopter money" où BCE envoie 50 Pass à tous les utilisateurs de user1 à userN, (pas pour State, il a déjà été servi).

— *Phase 2 - Mise en place de l'économie de marché* :
  — mise en place d'une FIFO,
  — générateur aléatoire de transaction entre deux utilisateurs pris au hasard avec un montant aléatoire (les transactions sont incomplètes et on ne remplit que les champs : émetteur, destinataire, catégorie, montant, liste input des UTXO consommés)
  — mises en file de n telles transactions où n est un nombre aléatoire,
  — défiler k transactions
  — compléter les tx défilées : ajout et suppression, timestamps, vérification des scripts, ajout des frais, du change, des liste outputs, gestion complète des UTXO
  — ajout d'un nouveau bloc avec ces tx et ajout de la tx de récompense
  — minage du bloc et ajout à la blockchain
  — mise à jour de la masse monétaire
  — mise à jour du montant de la récompense si besoin

— *Phase 3 - Mise en place du contrôle étatique* :
  — mise en oeuvre du contrôle par BGF : vérification des catégories, émetteur, destinataire. L'argent périmé et confisqué (dépassant le max d'épargne autorisé) est recyclé et attribué à l'état (évite la réduction de la masse monétaire).
  — création d'outils liés au contrôle : ajouter supprimer un item des listes d'autorisation, geler ou dégeler un compte, ajouter/supprimer une date limite.

Résumé du fonctionnement final du système en phase 3 On suppose qu'on est dans la phase de marché (après donc la phase d'«helicopter money». On utilise une file (FIFO) globale de transactions où BCE (en tant que mineur) vient lire des transactions en attente pour les traiter.

On répète le cycle suivant :

1. créer aléatoirement un nombre aléatoire de n transactions (partielles) entre deux utilisateurs aléatoires, avec un montant et une catégorie aléatoire (on pourra gérer une liste de catégories du type "cat1", "cat2",...)
2. Le mineur (BCE) lit k transactions dans la FIFO (k sera un nombre aléatoire généré pour chaque lecture),
3. pour chaque tx la BCE doit vérifier, les dates, choisir les UTXO en vérifiant les scripts et les catégories, vérifier les autorisations, calculer le change, les frais, la tx récompense où BCE envoie à "state" le montant de la récompence s'il est supérieur à zéro. Sinon, c'est que la phase d'inflation est terminée. On continue alors avec la phase de marché avec éventuellement un minage basé uniquement sur les frais de transaction
4. Le mineur (BCE) mine alors le bloc et l'ajoute à la blockchain

5. mise à jour de la masse monétaire et du montant de la récompense
6. impression de diverses informations : numéro du cycle, information du nouveau bloc, masse monétaire, à vous de voir.

Gestion des récompenses et de l'inflations : Cela peut se faire avec deux variables (c'est juste un exemple) :

```
1 int cycleRounds = 0; // nombre de cycles, tient compte du genesis
2 int limit = 30;// nb de cycles entre deux divisions de la récompence
```

A chaque cycle *cycleRounds* est incrémenté à chaque cycle et chaque fois qu'elle est un multiple de *limit* on divise la récompense par deux. Quand elle est à zéro, la phase d'inflation est terminée.

On arrête le programme soit sur une interruption gérée, soit sur un nombre de blocs ou cycle prédéterminé (au choix).

Liste de quelques points délicats ou à ne pas oublier et à caser quelque part :
- Gestion de la liste globale des UTXO, mettre à jour les UTXO des comptes ajout/suppression
- pour chaque nouvelle transaction (au sens échange d'argent) :
  - rechercher dans la liste des UTXO une ou plusieurs UTXO appartenant à l'émetteur et couvrant la dépense. S'il n'y en a pas, l'échange est refusé et on passe au suivant.
  - Générer la tx pour gérer cet échange
  - Dans cette tx, générer la liste des entrées pointant chacune vers une des UTXO et générer les scripts de déverrouillage.
  - exécuter les scripts au moyen d'une pile et vérifier la légitimité de l'émetteur
  - générer la liste des sorties avec la tx d'origine et le change.
  - créer le nouveau bloc et ajouter cette transaction à sa liste de transactions.
  - Calculer les frais et ajouter une transaction pour les frais dans la liste
  - Gérer la récompense et l'inflation
  - miner le bloc et l'ajouter à la blockchain.
  - Supprimer de la liste des UTXO celles qu'on vient d'utiliser (mais elles restent dans la blockchain), ajouter les nouvelles à qui de droit.

5 Structure de données

Le header "bc_define.h" sur Moodle définit des *struct* avec les champs utiles. Cela tiendra lieu de spécification pour les structures de données.

5.1 Programmes C fournis

5.1.1 sha256

L'algorithme de hash que nous allons utiliser est celui qu'utilise la blockchain du Bitcoin, le sha256. Le calcul du sha256 en C se fera en utilisant le code fourni : sha256.h, sha256.c, sha256_utils.h, sha256_utils.c par Brat Conte et adapté aux besoins du projet.

Vous n'avez besoin d'appeler que la procédure :

```
void sha256ofString(BYTE * str,char hashRes[SHA256_BLOCK_SIZE*2 + 1])
```

où str est la chaîne de caractères dont on veut calculer le hash (entrée), et hashRes est la chaîne de 32 caractères représentant le hash sous sa forme de caractères hexadécimaux (sortie).

Les fichiers sha256.c et sha256.h sont nécessaires mais doivent être considérés comme des boîtes noires.

Les types BYTE et SHA256_BLOCK_SIZE sont déclarés dans sha256.h

5.1.2 Timestamp

Pour le timestamp en C, il y a dans *time.h* des fonctions pour récupérer la date et l'heure qu'on peut utiliser pour créer une fonction *getTimeStamp* :

```
time_t getTimeStamp(){
    time_t ltime;
    time(&ltime);
    return ltime;
}
```

Consultez le manuel de "time.h" pour plus de détails.
Cette fonction renvoie une chaîne de caractères du type "Fri Jan 19 13 :40 :09 2018"

5.2 Constantes des programmes et performances attendues

La difficulté sera de 4 pour les tests. Pour les tests, le nombre de transactions doit pouvoir être de 1 à 10 (nombre aléatoire pour chaque bloc). Le nombre de blocs n'excèdera pas de 10, et le nombre d'utilisateurs 10 aussi. A terme, le programme doit pouvoir supporter la génération de beaucoup plus de blocs, par exemple 1000 blocks avec 20 ou 30 transactions par bloc et 100 utilisateurs.

Il est conseillé de faire des économies de mémoire (n'allouez que ce qui est nécessaire, faîtes des free de ce qui devient inutile. Attention toutefois à ne pas libérer une zone mémoire utilisée par ailleurs par une de vos structures de données, ça fait des seg fault difficiles à trouver).

6 Evaluation

Le barème détaillé est téléchargeable sur Moodle. Les points du programme rendu seront attribués à l'équipe dans son ensemble. Vous êtes libres de vous les partager comme bon vous semble. Par défaut, les points seront divisés par le nombre de membres de l'équipe sauf décision contraire du tuteur.