Blockchains – Feuille de TME #2

25 septembre 2021

Le but de TME est d'implémenter une bibliothèque permettant de créer et de manipuler des arbres de Merkle. Ce TME peut être réalisé dans le langage de programmation de votre choix. La fonction de hachage utilisée sera Blake2b avec comme paramètre de taille : 32, sans clé.

Rappels :

— 1 octet = 2 caractères hexadécimaux

— Au besoin, on peut utiliser la commande `xxd -b` pour passer de la représentation hexadécimale à la représentation binaire.

1 Arbres de Merkle

Un arbre de Merkle est une structure de donnée permettant de fournir un résumé de données (généralement imposantes) et permettant, entre autres, de vérifier de manière efficace l'intégrité de ces données sans nécessairement les avoir. L'arbre de Merkle se construit a partir d'un ensemble de **hash** de valeurs qui vont composer les feuilles de l'arbre. Comme illustré par la figure 1, les nœuds au-dessus des feuilles sont construits en **hashant de la concaténation de leur fils gauche et de leur fils droit** (qui sont également des hash). En procédant récursivement, l'arbre de Merkle est construit et la racine de Merkle (Merkle root) est obtenue. En utilisant une fonction de hachage cryptographique (i.e. non-inversible), il est (pratiquement) impossible de mentir sur l'intégrité des données.

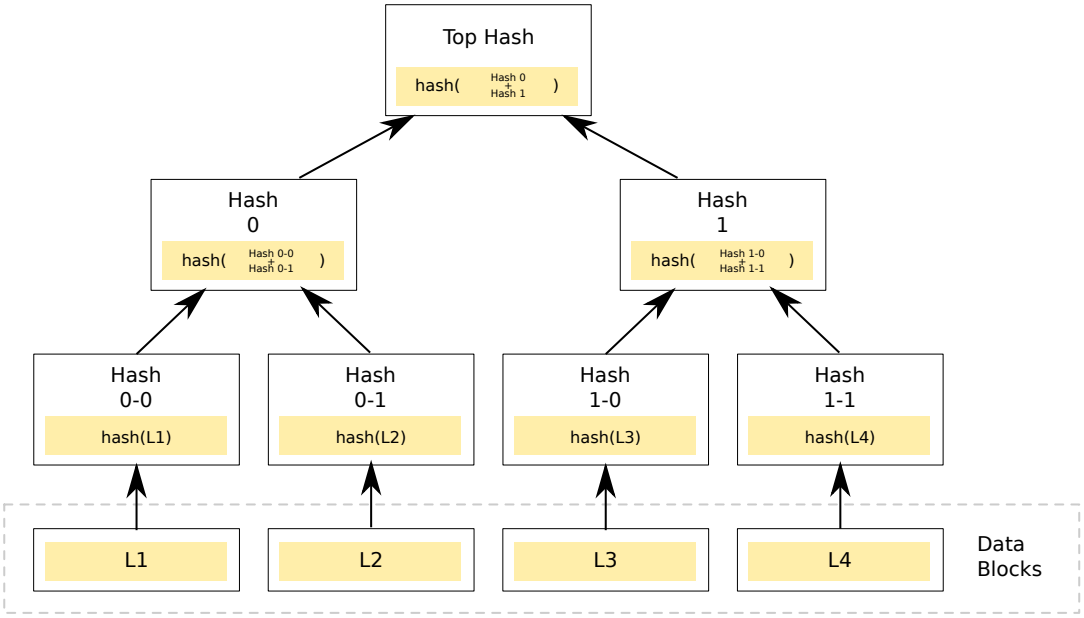

FIGURE 1 – Un arbre de Merkle [1]

Pour illustrer, lorsque l'on souhaite télécharger un ensemble de fichiers sur internet via un réseau pair-à-pair et que l'on cherche à s'assurer de la correction des données reçues, il est possible d'utiliser un arbre de Merkle : tout d'abord, il faut obtenir la racine de Merkle d'une source sûre (ami, organisme de confiance, etc.). Une fois obtenue, les pairs vous proposant cet ensemble de fichier vont devoir vous communiquer l'intégralité de l'arbre de Merkle avant de commencer à vous envoyer les fichiers. La première chose à faire est alors de s'assurer que l'arbre

1. `https://en.wikipedia.org/wiki/Merkle_tree`

reçue est effectivement correct en recalculant la racine : si elle diffère, les données sont incorrectes ou corrompues, si elle correspond alors les fichiers sont cohérents et vous pouvez autoriser le pair à vous transmettre ses données. Toutefois, il reste nécessaire de s'assurer que les données reçues correspondent aux hashs annoncés (en hashant localement les données reçue). Un des avantages de cette structure réside également dans le fait qu'il est possible de partitionner l'arbre. Par exemple, dans la figure 1, si le pair ne possède que les fichiers L1 et L2, plutôt que de fournir l'arbre de Merkle complet, il lui est possible de ne communiquer qu'un sous-arbre. Ce sous-arbre peut alors être le hash de `L1`, `L2` et `Hash 1` : le hash résultat de L3 et L4 et, bien sûr, la racine de Merkle. Ainsi, il reste possible pour le récepteur de vérifier l'intégrité du sous-ensemble de fichiers qu'il possède sans qu'il ait à stocker l'intégralité de l'arbre de Merkle.

Dans le contexte de la blockchain, les arbres de Merkle sont utilisés, entre autres, pour permettre de vérifier qu'une transaction a correctement été incluse dans un bloc sans avoir à télécharger toutes les données. Pour ce faire, l'émetteur d'un bloc inclut dans son bloc la racine de Merkle de l'ensemble des opérations appliquées. Si une personne cherche à vérifier que sa transaction a été incluse dans ce bloc, il est possible de lui fournir un témoin (ou preuve de Merkle) consistant en un sous-ensemble minimal de hash permettant de vérifier son inclusion sans avoir à lui transmettre tout l'arbre de Merkle.

Exercice 1. Implantez une fonction `concat_hash` qui prend deux hash h_1 et h_2 en entrée et qui retourne le hash de ces hashs concaténés : $\mathcal{H}(h_1 \cdot h_2)$.

Exercice 2. Implantez une fonction `create_merkle_tree` qui prend en entrée un ensemble d'éléments et qui en construit son arbre de Merkle.

Note : pour plus de simplicité, nous supposerons que la cardinalité n de l'ensemble sera de l'ordre de $n = 2^k$ et que l'arbre de Merkle construit est binaire et parfaitement équilibré.

Exercice 3. Pour un arbre contenant 4 données, combien de hashs sont nécessaires pour vérifier l'intégrité d'une donnée ? Même question pour un arbre contenant 8 données. Généralisez et donnez le nombre de hashs nécessaire pour vérifier un arbre de taille n. $2^n - 1$

Exercice 4. Implantez une fonction `witness` qui a partir d'un arbre de Merkle et d'une feuille de cette arbre retourne son témoin, c'est-à-dire le **sous-arbre de Merkle minimal** nécessaire pour garantir l'intégrité de cette donnée.

Exercice 5. Pour finir, donnez la fonction `verify` prenant un témoin et une racine de Merkle et qui vérifie que ce témoin correspond effectivement à la racine fournie.

FIGURE 2 – Exemple d'arbre de Merkle