Dauphine | PSL
UNIVERSITÉ PARIS

Année universitaire 2019/2020

Projet programmation par composant

Dossier de spécifications techniques du composant N°6 :

Signature

Membres de l'équipe :

AÏT-MOULOUD Lisa

ARZUR Samuel

HAMI Hocine

LEBIB Mohamed

Liste des modifications du document

Version	Date	Description des modifications
1.0	07/04/2020	Création du document
1.1.1	16/04/2020	Génération des paires de clés publique/privée
1.1.2	18/04/2020	Modification de la signature de la fonction « generate »
1.2	19/04/2020	Rendu version 1
2.0	26/04/2020	Mise à jour suite à l'implémentation du code
2.1	10/05/2020	Révision des types de retour
3.0	21/05/2020	Manuel d'utilisation de la librairie signature.so

Documents de référence

Nom du document	Description
Programmatuonpar_composants_2.ppt	Pseudo dossier de spécifications générales concernant la structure de la Blockchain
Schema_bloc.ppt	Document précisant les relations d'utilisation et d'inclusion des composants de la Blockchain
Les fondamentaux de la blockchain	https://systematic-paris-region.org/fr/les-fondamentaux-de-la-blockchain/
Librairie micro-ecc	https://github.com/joseluu/micro-ecc

1. Contexte et objectifs

Dans le cadre du cours « Programmation par composants », il nous a été demandé d'implémenter une Blockchain fonctionnelle. Et ceci, en utilisant une approche modulaire qui respecte les principes de la programmation orientée composant.

L'idée est d'implémenter l'architecture d'une Blockchain sous forme de plusieurs petits modules. Chacun de ces modules, qu'on appellera « composant » dans la suite du document, sera implémenté par un groupe différent. Afin d'assurer une lisibilité et réutilisabilité du code, ainsi qu'une facilité d'implémentation.

Le document portera donc sur l'implémentation d'une Blockchain simplifiée qui simulera des transactions ***Bitcoin*** simples entre plusieurs portefeuilles.

2. Blockchain et composants

La *blockchain* est une technologie de stockage et de transmission d'informations à faible coût, sécurisée, transparente et fonctionnant sans organe central de contrôle. Par extension, la *blockchain* désigne une base de données sécurisée et distribuée (car partagée par ses différents utilisateurs), contenant un ensemble de transactions dont chacun peut vérifier la validité. La *blockchain* peut donc être comparée à un grand livre comptable public, anonyme et infalsifiable.

Dans notre implémentation, la Blockchain sera simulée par un **fichier (composant 1)** qui contient un verrou. Celui-ci empêche l'ouverture, si le fichier en question et donc la blockchain contient une incohérence.

Le fichier qui simulera la Blockchain contiendra des données structurées qui représentent les **blocs (composant 4-a)**. Ces blocs en question résument les transactions (envoi et réception de Bitcoin) effectuées entre les utilisateurs. Ces derniers sont représentés par des **portefeuilles/wallet (composant 2)** et **signés** avec une clé privée **(composant 6)**.

Les blocs sont identifiés par le **hash (composant 4-b)** qui se base sur tous les éléments qui y sont contenus pour sortir un code d'identification unique.

Toute transaction effectuée est d'abord mise en attente, puis elle est par la suite validée par un module appelé **« mineur » (composant 3).**

Les incohérences dans le contenu des blocs et dans les transactions sont détectées par un module appelé le **vérificateur de bloc (composant 5).** Celui-ci se base sur la **signature** du bloc **(composant 6)** en utilisant des clés publiques pour vérifier si les données sont toujours intègres.

Les liens entre les composants sont résumés et schématisés dans le fichier cité dans les références *[Schema_bloc.ppt].*

3. Spécifications techniques du composant 6 : signature

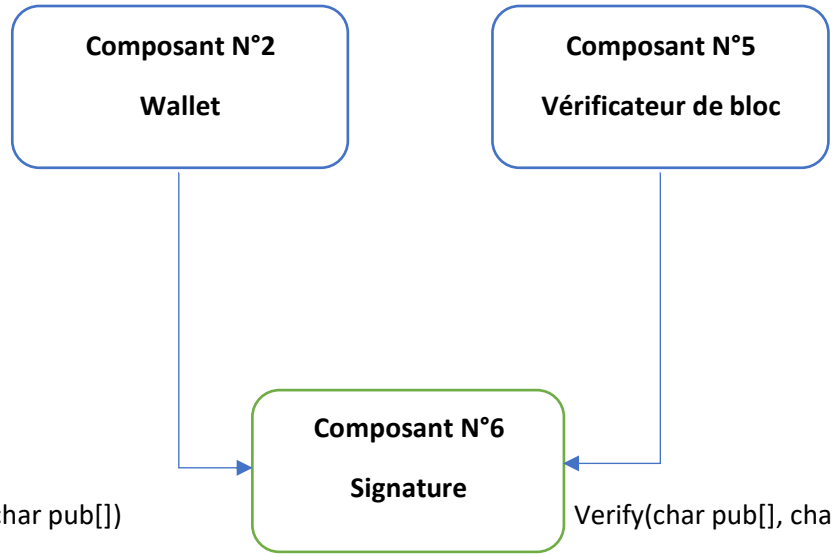
Lorsqu'un utilisateur enregistré souhaite ajouter une transaction à la blockchain, il la soumet au mineur pour que celui-ci puisse la recevoir, la vérifier et l'ajouter au bon bloc. Mais pour être vérifiées, les transactions (qui sont des données) doivent être signées afin que le mineur puisse identifier incontestablement l'expéditeur.

La signature est générée par une fonction cryptographique qui prend en entrée les données d'origine et la clé privée de l'expéditeur, et fournit en sortie une signature. Cette signature est vérifiable par toute personne utilisant une autre fonction qui vérifie la cohérence entre les données, la signature et la clé publique. Par conséquent, lors de l'envoi de données l'expéditeur ajoute également la signature correspondante et sa clé publique.

Notre composant est chargé de :

- Générer les paires de clés : publique/privée
- Signer les données
- Valider une signature

Ci-dessous, une représentation schématique de l'utilisation de notre composant par les autres composants du projet.



1. Générateur de paires de clés :

Chaque transaction a recours à la cryptographie asymétrique, la génération de cette paire de clés présente le double intérêt de chiffrer ou de signer un message. Ainsi, dans ce système un utilisateur créer une suite aléatoire de chiffres, appelée clé privée. Puis à partir de celle-ci un algorithme permet de produire une seconde clé, appelée clé publique.

Dans le schéma ci-dessus on prend l'exemple du Bitcoin où il s'agit d'un algorithme de signature numérique à clé publique dit à courbes elliptiques appelé ECDSA (pour *Elliptic Curve Digital Signature Algorithm*) qui effectue les étapes suivantes pour la création des clés.

- **Préparation des clés :**
  - **a)** Choisir un entier *s* entre 1 *et* $n-1$ *qui sera la clé privée.*
  - ***b)*** Calculer $Q = s\,G$ *en utilisant l'élément de la courbe elliptique.(G un point de la courbe élliptique)*
  - ***c)*** *La clé publique est Q et la clé privée est s.*

En-tête de la fonction de génération de paires de clés :

generatePairKey(char priv[], char pub[])

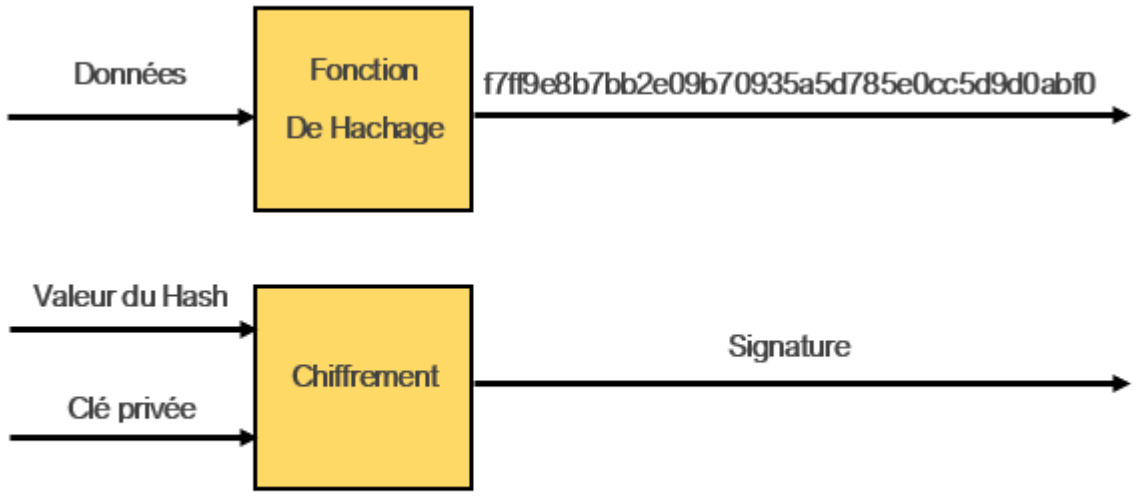
- Entrées:
  - *priv* : tableau pour accueillir la clé publique qui sera gardée dans le portefeuilles/wallet pour être utilisé lors de la vérification du bloc.
  - *pub* : tableau pour accueillir la clé privée qui sert à signer le bloc

2. Signature des données :

Signer un bloc d'une Blockchain revient à effectuer les étapes suivantes :

- Récupérer le hash du bloc généré par le ***composant 4 : hacheur SHA-256***. Ce dernier génère un code SHA-256 en se basant sur les données contenues dans le bloc (ID, date de création, émetteur, récepteur, montant et le hash du bloc précédent).
- Utiliser une clé privée générée par le composant lui-même pour chiffrer le hash récupéré.
- Envoyer le bloc, sa signature et une clé publique au vérificateur de bloc pour que celui-ci puisse effectuer sa vérification.

La procédure de signature d'un bloc est résumée dans le schéma ci-dessous :



Création d'une signature

En-tête de la fonction de signature d'un bloc :

La fonction qui se chargera de signer les blocs de la Blockchain sera donc à priori comme suit :

sign(char priv[], char hash[], char signature[])

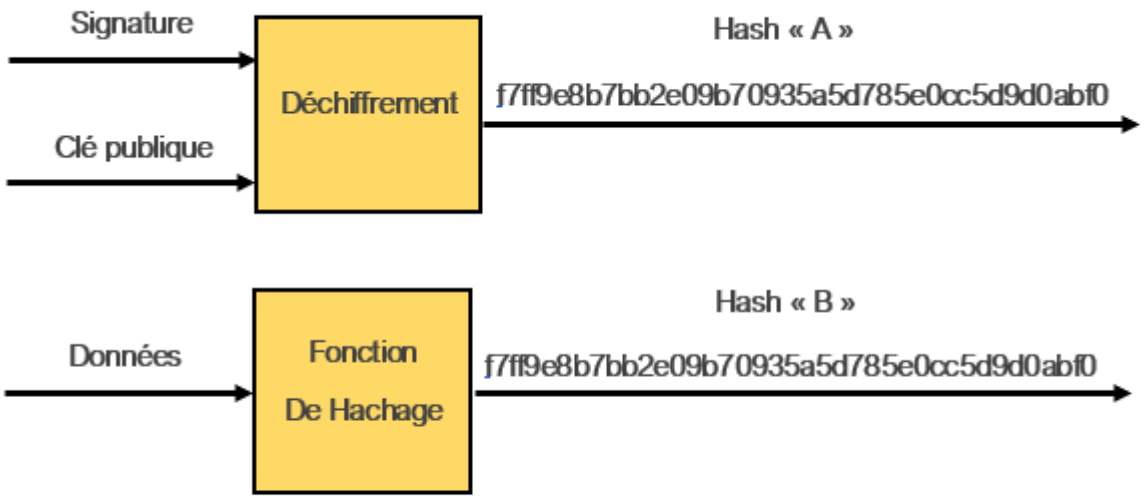
- Entrées:
  - *hash(Bloc)* : tableau du hash SHA-256 (composant 4).
  - *priv* : une clé privée générée en paire clé publique/clé privée qui est stockée dans le portefeuilles/wallet.
  - *Signature* : tableau pour accueillir la signature digitale d'un bloc qui sera sous forme de hash également.

3. Vérification de la signature :

Vérifier un bloc se résume en 4 étapes. Le vérificateur de bloc doit recevoir un bloc et une clé publique.

- La clé publique est utilisée pour déchiffrer la signature du bloc. Cette opération donne en sortie un hash qu'on appellera ***« hash à vérifier ».***
- D'un autre côté, il faudra appliquer l'algorithme de hash du composant 4 au bloc reçu pour re-générer le bon hash du bloc. On appellera le code obtenu ***« hash réel ».***
- La dernière étape serait donc de comparer le ***« hash à vérifier »*** et le ***« hash réel ».***
  Si les deux sont identiques, cela prouve que le bloc n'a pas été altéré durant le transit et que les données qu'il contient sont toujours intègres. Sinon, le bloc est considéré comme non valide.

La procédure de vérification d'une signature d'un bloc est résumée dans le schéma ci-dessous :



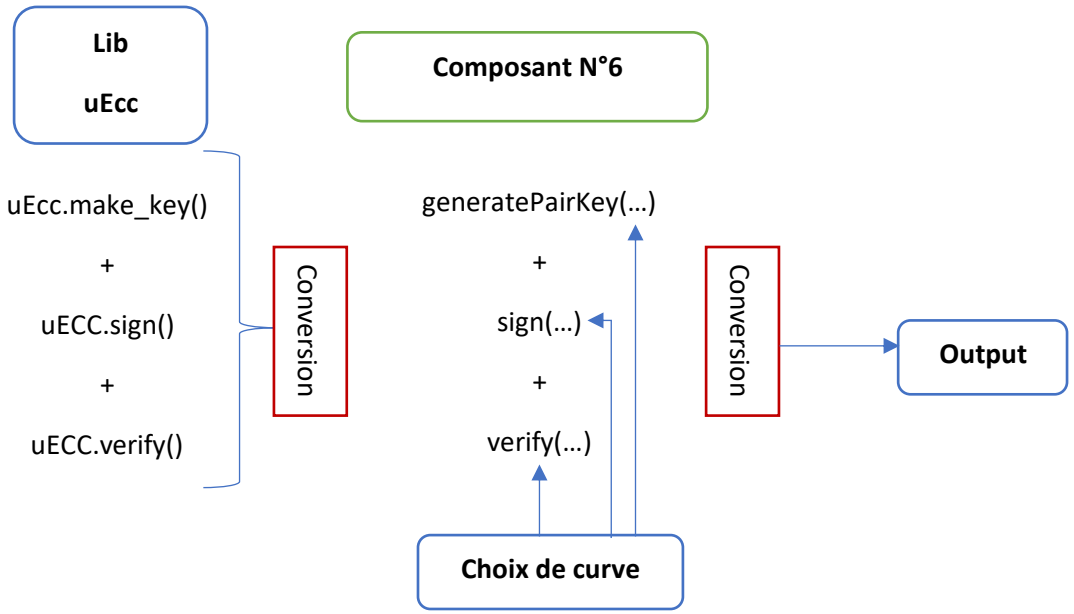
Vérification d'une signature

En-tête de la fonction de vérification de signature d'un bloc :

La fonction qui se chargera de vérifier les signatures des blocs de la Blockchain sera donc, à priori, comme suit :

Verify (char pub[], char hash[], char signature[])

- Entrées :
  - *pub* : une clé publique générée en paire clé publique/clé privée et qui correspond à la clé privée avec laquelle a été signé le bloc
  - *signature* : le résultat du chiffrement du bloc avec la clé privée
  - *hash* :
- Sortie :
  - *boolean* : true si la vérification est correcte, false sinon.


Lib
uEcc
Composant N°6
uEcc.make_key()
+
uECC.sign()
+
uECC.verify()
Conversion
generatePairKey(…)
+
sign(…)
+
verify(…)
Conversion
Output
Choix de curve

4. Négociations

Après concertation avec les autres groupes nous en sommes venus à la conclusion que c'est le groupe du composant n°2 qui va générer les paires de clés (**), puis ils vont appeler la fonction qui permet de signer avec la clé privée de la paire générée (**) et joindre la clé publique à la transaction avant de l'envoyer aux autres composants.

Aujourd'hui on a un problème de types dans nos fonctions. Nous manipulons des tableaux de unint8_t alors que les autres composants nous transmettent et ont besoins de char[].

Après concertation avec les autres équipes, et vu la structure des autres composants, il s'est avéré plus pratique de manipuler des string pour les clés (privée et publiques), la signature et aussi le hash code.

5. Librairie signature.so

Pour utiliser le composant #6 de signature, il suffit d'importer la librairie signature.so se trouvant sur le lien : https://github.com/lisa1am/Blockchain/blob/master/signature.so

Contenu de la librairie :

La librairie signature contient trois fonctions principales pour gérer la signature de hash chose :

- ***String GeneratePairKey()*** : cette fonction ne prend aucun paramètre et renvoie une chaine de caractère.
    - Il est important de noter que le couple (clé publique, clé privée) est renvoyé dans la même chaine de caractère et séparés par un "-". Il suffit alors de faire appel à la fonction split (voir la suite) pour récupérer chacune des clés dans une chaine de caractères séparée.
    - ***String split(string paireDeClés, int arg)*** : cette fonction prend en paramètre la chaine de caractère ou les deux clés publique et privée sont concaténées et un entier **[arg=0 pour récupérer la clé publique et arg=1 pour récupérer la clé privée]**. Et retourne à la fin la clé souhaitée selon le paramètre sous forme de string hexadécimal.
- ***String sign(string clé_privée, string hash_code)*** : cette fonction sert à signer un hash code de type string qui sera passé en paramètre avec une clé privée générée et retourne la signature sous forme de string en format hexadécimal.
- ***Bool verify(string clé_publique, string hash_code, string signature) :*** cette fonction prend en paramètre la clé publique générée (qui vient du couple (clé publique, clé privée) et dont la clé privée a été utilisée pour signer le hash code), le hash code signé auparavant, et la signature récupérée de la fonction sign. Elle renvoie un booléen :
    - Vrai : la signature est valide
    - Faux : la signature n'est pas valide

6. Tests unitaires et fonctionnels

Avant de générer la librairie, les fonction C++ ont été testées unitairement et dans un contexte de scénario de signature d'un hash code exemple. La librairie a été générée ensuite tout en vérifiant que l'appel des fonctions C++ à partir d'un programme python fonctionnait correctement.

Tests du scénario de signature en C++ :

```
1809  int main(){
1810      // hash code exemple
1811      std::string hash_code = "5a822196bf1c45b1e74c6d04e4127d0296d64695932f40909bdab3ba8a9b0528";
1812
1813
1814      std::string pairKey = generatePairKey();
1815      std::string pub = split(pairKey, 0);
1816      cout << "GENERATED PAIR KEY => PUBLIC AND PRIVATE -> \n";
1817      cout << "PUBLIC KEY [A]= " << pub << "\n";
1818
1819
1820      std::string priv = split(pairKey, 1);
1821      cout << "PRIVITE KEY [A]= " << priv << "\n";
1822
1823
1824      std::string sig = sign(priv, hash_code);
1825      cout << "\nSIGNING THE HASH \"" << hash_code << "\" WITH PRIVATE KEY [A]...\n";
1826      cout << "SIGNATURE [A]= " << sig << "\n";
1827
1828      cout << "\nVERIFYING SIGNATURE WITH PUBLIC KEY [A].. \n";
1829      if(verify(pub, hash_code, sig)){
1830          printf("VALID SIGNATURE !\n");
1831      }
1832      else{
1833          printf("NON VALID SIGNATURE !\n");
1834      }
1835
1836      // trying to verify with public key B, a signature signed with private key A
1837      std::string pairKey2 = generatePairKey();
1838      std::string pub2 = split(pairKey2,0);
1839      cout << "\n\nGENERATED PAIR KEY => PUBLIC AND PRIVATE -> \n";
1840      cout << "PUBLIC KEY [B]= " << pub << "\n";
1841      cout << "\nVERIFYING SIGNATURE WITH PUBLIC KEY [B].. \n";
1842      if(verify(pub2, hash_code, sig)){
1843          printf("VALID SIGNATURE !\n");
1844      }
1845      else{
1846          printf("NON VALID SIGNATURE !\n");
1847      }
```

Output :

```
liisa@liisa:~/Documents/PC/Blockchain/signature$ ./sig
GENERATED PAIR KEY => PUBLIC AND PRIVATE ->
PUBLIC KEY [A]= 4b9fe178ad3da88ea8f3aa4666b3cad4b9ba41caca9c1d3671a9e70ad11f229104c8786a4aeeeb564cea8e87996bc9091e09002f149e1c74f19752df38f93ffe
PRIVITE KEY [A]= 1b7d8a98cc8fcc0bfab75493cb80dac919285260049c21af571faa46d13a4418

SIGNING THE HASH "5a822196bf1c45b1e74c6d04e4127d0296d64695932f40909bdab3ba8a9b0528" WITH PRIVATE KEY [A]...
SIGNATURE [A]= 2ae04fd23607a46304813fef70eb6d19eb038ff5a5e3ab99b265913afeb110a4f83cb847fb692149c95a89571a4c72c351b94d0a0cc56e840f46db055d79dacf

VERIFYING SIGNATURE WITH PUBLIC KEY [A]..
VALID SIGNATURE !


GENERATED PAIR KEY => PUBLIC AND PRIVATE ->
PUBLIC KEY [B]= 4b9fe178ad3da88ea8f3aa4666b3cad4b9ba41caca9c1d3671a9e70ad11f229104c8786a4aeeeb564cea8e87996bc9091e09002f149e1c74f19752df38f93ffe

VERIFYING SIGNATURE WITH PUBLIC KEY [B]..
NON VALID SIGNATURE !
```

Test du scénario de signature en utilisant la librairie **signature.so** dans un programme python :

```
import signature

keys=signature.generatePairKey();
priv=split(keys,1);
pub = signature.split(keys,0);
signature.sign(priv,"5a822196bf1c45b1e74c6d04e4127d0296d64695932f40909bdab3ba8a9b0528");
bool = signature.verify(pub, "5a822196bf1c45b1e74c6d04e4127d0296d64695932f40909bdab3ba8a9b0528", sig);
print(bool);
```