DAUPHINE

Spécification du projet composant

Composant 2 : « Portefeuille / Wallet »

Réalisé par :

REGRAGUI Imane

SAMOU Timothée

TAMAZI Hanane

Encadré par :

LUU José

Année universitaire : 2015/2016

Introduction

Ce document a pour but de rassembler les spécifications concernant le composant 2 « portefeuille/wallet » du projet de classe

Version du document	Date	Auteur(s)	Modifications
1.0	27/01/2016	Jose Luu	Version initiale
1.1	27/01/2016	Jose Luu	Modification pour exemple
1.2	25/02/2015	Groupe 3	Création des spécifications du composant 2 « Portefeuille »

I- Fonctionnement du Composant 2

Un portefeuille est un porte-monnaie de BitCoin qui permet à son propriétaire d'échanger les données avec les autres utilisateurs. En fait, il rend son possesseur titulaire d'un solde en BitCoin afin qu'il puisse recevoir et envoyer des BitCoin.

Un portefeuille permet d'insérer de nouvelles transactions signées dans le registre Bitcoin. La clé publique est une adresse Bitcoin. N'importe qui disposant d'un porte-monnaie crédité peut envoyer des bitcoins à cette adresse. La clé privée est un mot de passe qui permet de signer des transactions pour dépenser des bitcoins.

Le composant portefeuille doit pouvoir connaître à n'importe quel instant le montant qui lui est attribué, pour cela il doit connaître toute les sorties (UTXO) dont il est le destinataire. Rappelons qu'un compte est rattaché à une clé privée et une clé publique. Le portefeuille doit retrouver toute les transactions qui lui sont destinées.

Deuxièmement le composant doit pouvoir composer des transactions sachant que le portefeuille peut avoir plusieurs compte (plusieurs clés privées et plusieurs clés publiques). Une transaction est définie par une clé privée de l'émetteur, une clé publique du récepteur et un montant associé.

Le processus du composant se présente comme suit :

1- Connaître la valeur du portefeuille

- Lecture des blocs du **composant 1** avec la fonction *Bloc getBlocs()*
- On récupère les UTXO de chaque bloc lu avec *UTXO getUTXO(bloc)*
- On récupère les montants disponibles avec *Double getMontant(publicKey)*

2- Composer une transaction

Une transaction se définit par :

- Une clé privée de l'émetteur de la somme à envoyer
- Une clé publique qui désigne le récepteur des BitCoin
- La somme des Bitcoin à envoyer.

Pour réaliser une transaction il faut :

- Recevoir la clé privée de l'émetteur avec String *getprivateKey(emetteur)*
- Recevoir la clé publique du récepteur avec String *getPublicKey(recepteur)*
- Définir un montant que l'on souhaite associer à la transaction *void setAmount(amount)*

Remarque :

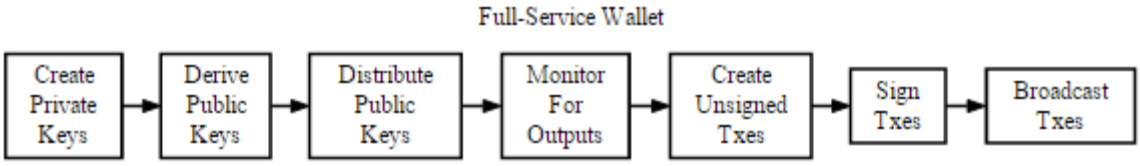
La somme des Bitcoin qu'un utilisateur doit envoyer ne doit pas dépasser la somme des Bitcoin dont il dispose, pour vérifier cela, on utilise la notion des UTXO qui désignent les OutPut, autrement dit les BitCoin non dépensés. On note que les Output qu'un utilisateur utilise lors d'une transaction, deviendront des InPut pour le destinataire des BitCoin.

- Identifier les UTXO dont la clé est celle dont dispose le destinataire *Vector<UTXO> getUTXO(cléDestinataire)* en vérifiant que les UTXO n'ont pas été dépensé auparavant
- Vérifier si la somme des montants UTXO identifiés dans l'étape précédente est supérieure ou égale au montant à envoyer. *checkSum(amount, Vector<UTXO>)*
- Générer une UTXO comprenant le montant indiqué avant, avec le destinataire. *UTXO generateUTXO1(amount, receiver, sender)*
- Une deuxième UTXO sera générée et qui aura comme montant la différence entre la somme des UTXO valides et le montant de la transaction, ainsi que la clé de l'émetteur. *UTXO generateUTXO2(amount, Vector<UTXO>, senderKey)*

- Faire appel au composant 6 pour signer la transaction avec notre clé privée
- Envoyer la transaction au réseau, avec les informations suivantes : Emetteur, destinataire, montant disponible... au composant 5 pour la vérification et la validation de la transaction.
- Dès que la transaction est validée, elle est enregistrée dans un bloc sur le registre global de transactions (Blockchain).

Dans la suite du document, chacune des étapes du schéma sont explicitées afin de mieux comprendre le fonctionnement du composant et son mode de réalisation.

II- Les étapes



Un portefeuille effectue les trois fonctions :

- Il génère les clés privées
- Il en déduit les clés publiques correspondantes
- Aide à distribuer les clés publiques si nécessaire
- créer et signer les transactions qui ont dépensés les Output
- Émettre en broadcast les transactions signées

1- Réception du composant 1

Notre module se base sur le composant 1 pour:

- Ouvrir le fichier.
- Parcourir et vérifier tous les blocs.

Vector<Bloc> getBlocs(File fich) ; c'est une fonction qui retourne une liste de blocs à partir d'un fichier "

On peut dire que le type "Bloc" est une classe ayant un certain nombre d'attributs caractérisant un bloc".

Retour du composant 1 : un bloc avec un numéro donné

2- Récupération des UTXO

On lit les UTXO de chaque bloc retourné par le composant 1 grâce à la méthode UTXO (Bloc block) de *Type UTXO.*

3- Composer des transactions

Pour composer une transaction, on a besoin de :

- L'adresse bitcoin de destination pour la transaction, qui est une adresse publique
- L'adresse bitcoin privé de l'émetteur de la transaction
- Le montant de bitcoin à envoyer

Pour cela, il faut récupérer l'adresse publique du destinataire des bitcoin *String getpublicKey(recepteur)* et récupérer l'adresse privé de l'émetteur des bitcoins *String getprivateKey(émetteur)*, finalement il faut saisir le montant de bitcoin à transmettre avec la fonction *setAmount(amount)* comme récapitulatif, le portefeuille de l'émetteur construit une transaction qui attribue une somme à l'adresse bitcoin fournie par le destinataire , déplaçant les fonds sélectionnés et contenus dans le portefeuille de l'émetteur et signant la transaction à l'aide de la clé privée de ce dernier, Cela signale au réseau bitcoin que l'émetteur a autorisé le transfert de valeur depuis une de ses adresses vers la nouvelle adresse publique du destinataire. Pendant la transmission de la transaction via le protocole peer-to-peer, elle se propage rapidement sur le réseau bitcoin. et tous les nœuds connectés au réseau reçoivent la transaction.

- **Les UTXO**

Les transactions transfèrent des fonds depuis leurs entrées UTXI vers leurs sorties. Une entrée représente la provenance des fonds, en général la sortie d'une transaction précédente.
Une sortie matérialise le transfert des fonds en les associant à une clef, qui représente une

Contrainte permettant de verrouiller les fonds en spécifiant le type de signature qui sera nécessaire pour les dépenser. Les sorties d'une transaction peuvent être utilisés en entrée de nouvelles transactions

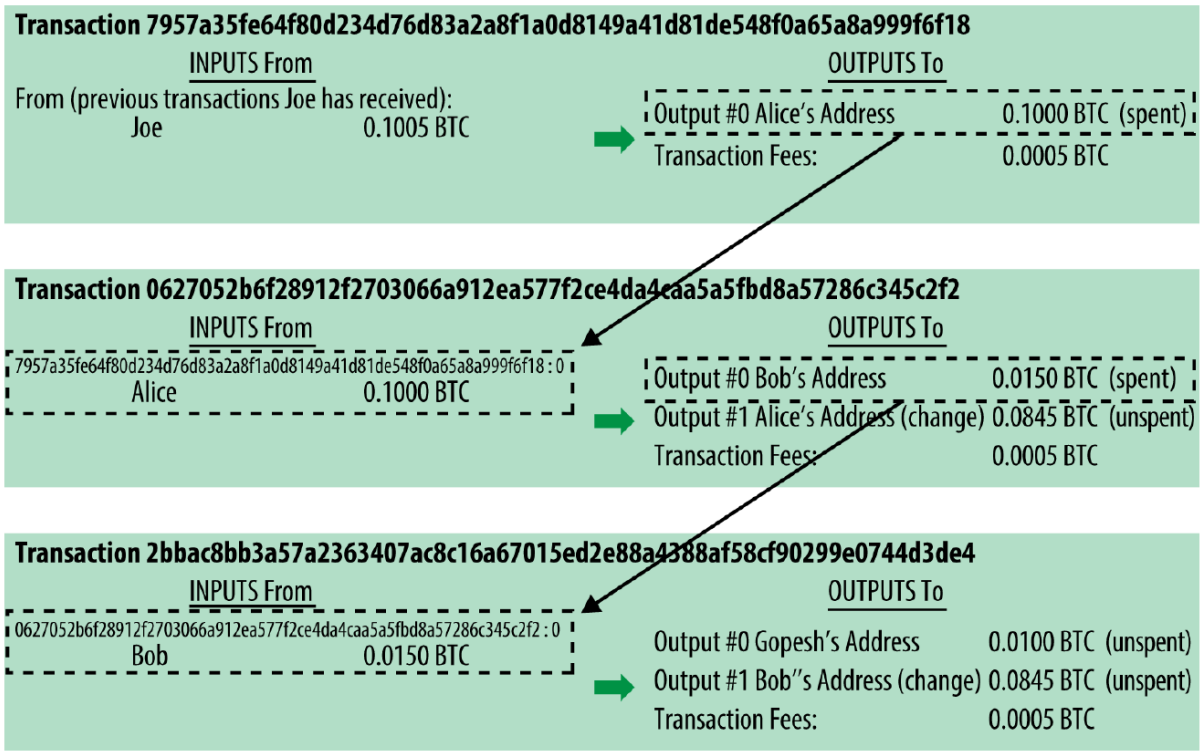


Figure : Une chaîne de transactions, où la sortie d'une transaction est l'entrée de la transaction suivante

Les transactions forment une chaîne, les entrées des dernières transactions correspondent aux sorties des transactions précédentes. La clef d'Alice permet de créer une signature qui déverrouille les sorties précédentes, prouvant ainsi au réseau bitcoin que c'est elle qui détient ces fonds. Elle lie ce paiement à l'adresse de Bob, ce qui verrouille les fonds qui ne peuvent être utilisés que si Bob fournit une signature valable. Cela représente un transfert de valeur de Alice vers Bob. Cette chaîne de transaction, de Joe vers Alice puis Bob, est illustrée dans Une chaîne de transactions, où la sortie d'une transaction est l'entrée de la transaction suivante.

III- Description des erreurs

Pour le composant, chacune des erreurs est gérée par les exceptions (throws) suivantes :

Etape 2.1

1. Nom Fonction : *VerifyAmount(publicKey X)*	
Erreur : somme à envoyer invalide	*the value you want to send is higher than what you have in your wallet*
Erreur : données négatives	*Negative value*
Erreur : données manquantes	*Missed data*

Etape 2.2

Nom Fonction : *AmountSet(double UTXO_emet, double UTXO_dest, double Amount, double somme_portfeuille)*	
Erreur : données négatives	*Negative value*
Erreur : données manquantes	*Missed data*
Erreur : valeur UTXO_dest est différente du montant envoyé	*UTXO_DESTINATION must be updated by the amount sent by the transmitter after every transaction*
Erreur : valeur UTXO_emet est différente de la différence du solde disponible avant l'ouverture de la transaction et le montant envoyé	*UTXO_TRANSMITTER must be updated and debited after every transaction*

Etape 2.3

Nom Fonction : *VerifyKeys(publicKey X)*	
Erreur : texte crypté invalide	*The text is not well crypted: combination of he key and the the text to be crypted is not well done.*
Erreur: données erronées	*wrong type of Data*

IV- Plan de tests

4.1- Interaction vis-à-vis des autres composants

Il s'agit d'une application généralement constituée de multiples composants, et les interactions entre ceux-ci pouvant être complexes il va falloir choisir dans que ordre réaliser ces tests d'intégrations. Si l'on considère l'exemple suivant, le composant utilise les composants 1, 5, 6 et est utilisé par le composant 5 et 6.

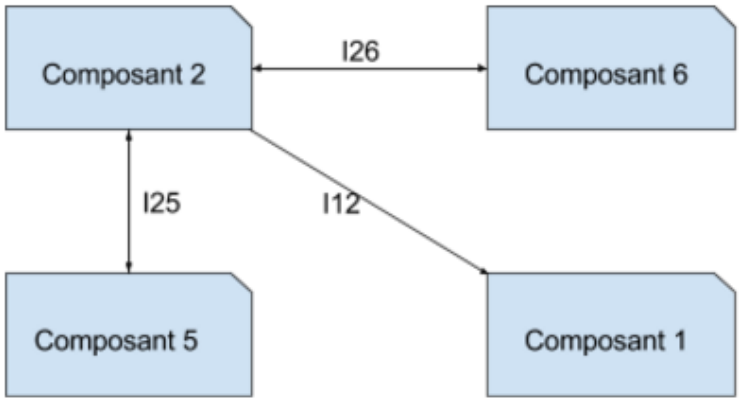


Figure: *Schéma d'architecture logicielle*

Dans un premier temps, pour pouvoir simuler le fonctionnement de notre composant de manière indépendante nous allons simuler les outputs des composants dont notre wallet dépend, en l'occurrence, les composants 1, 5 et 6

Composant 1 : Nous retournons une série de bloc fictive (une blockchain) avec hypothèse que les blocs sont valides

Composant 5 : Nous envoyons un bloc non-valide, puis un bloc valide, une transaction valide et une transaction non valide et nous simulons un retour de validation du composant

Composant 6 : Nous transmettons notre transaction et notre clé privé au composant qui nous retourne une signature

4.2- Test du composant

Un main fait appel aux fonctions des différents composants et compare le comportement du composant avec celui attendu. Si le composant n'a pas le comportement souhaité pour ce cas de test, il y aura un message nous indiquant l'erreur. Si tout est correct, il n'y aura donc pas de message affiché d'erreur affiché.

NB : Pour les trois fonctions listées ci-dessous, il est possible qu'une exception soit retournée alors qu'elle n'était pas attendue. Dans ce cas, le message d'erreur suivant apparaitra :
"FAILED: 'XXX' EXCEPTION TRIGGERED WHEREAS DATA ARE CORRECT";
XXX : Name of the exception

- ✓ *VerifyAmount*

Le message d'erreur suivant s'affiche dans le cas d'une valeur négative des UTXO.

"NEGATIVE VALUE EXCEPTION UNTRIGGERED ";

Le message d'erreur suivant s'affiche dans le cas d'une donnée manquante dans les UTXO.

"MISSING DATA EXCEPTION UNTRIGGERED ";

Le message d'erreur suivant s'affiche dans le cas d'une donnée invalide: Le montant que l'émetteur désire envoyer est supérieur à ce qu'il a dans son propre portefeuille.

"INVALID: SUM_VALUE < VALUE_INIT EXCEPTION UNTRIGGERERD ";

- ✓ *AmountSent:*

Le message d'erreur suivant s'affiche dans le cas d'une valeur négative des UTXO.

"NEGATIVE VALUE EXCEPTION UNTRIGGERED ";

Le message d'erreur suivant s'affiche dans le cas d'une donnée manquante dans les UTXO.

"MISSING DATA EXCEPTION UNTRIGGERED ";

En notant UTXO1 la valeur des UTXO du destinataire UTXO2 celle de l'émetteur ce test doit permettre de vérifier que la valeur UTXO1 du destinataire a bien reçu la somme M envoyé et de vérifier que la valeur UTXO2 de l'émetteur est devenue S (valeur du portefeuille avant la transaction) - M (valeur de la transaction)

Le message d'erreur suivant s'affiche dans le cas où la valeur les UTXO de l'émetteur et du destinataire n'ont pas été mise à jour.

"UPDATE FAILED: UTXO_DESTINATION IS DIFFERENT OF AMOUNTSENT EXCEPTION UNTRIGGERERD ";

"UPDATE FAILED: UTXO_TRANSMITTER HAS NOT BEEN DEBITED YET EXCEPTION UNTRIGGERERD ";

- ✓ *VerifyKeys*

Si la clé n'est pas représentée par le bon type par exemple la clé de type binaire, le message d'erreur suivant doit être renvoyé :

"WRONG DATA EXCEPTION UNTRIGGERED ";

Dans le cas où l'algorithme mathématique ne combine pas bien la clé avec le texte à crypter pour le rendre illisible, le message suivant s'affiche:

"INVALID DATA EXCEPTION UNTRIGGERED ";