Nexium

Un micro-système économique décentralisé pour étudiants
basé sur la blockchain et la cryptographie

Jean HERAIL
jean.herail@epita.fr

Milo DELBOS
milo.delbos@epita.fr

Antonin BESSIÈRES
antonin.bessieres@epita.fr

William VALENDUC
william.valenduc@epita.fr

1 Principe fondamental de Nexium

Nexium serait une monnaie virtuelle décentralisée conçue pour les étudiants d'Épita. Chaque participant disposera d'une paire de clés cryptographiques (privée et publique) de type RSA associée à son adresse `@epita.fr`, permettant d'authentifier et de sécuriser les transactions de crédits entre utilisateurs. Grâce à une architecture distribuée, Nexium garantira la transparence et l'intégrité des virements, tout en offrant une expérience simple et accessible via une interface graphique. La monnaie du réseau Nexium, le `NXM`, sera utilisée pour effectuer des transactions entre les utilisateurs, qui pourront consulter leur solde et leur historique de transactions de manière publique.

Le réseau Nexium s'articule donc comme une blockchain privée intrinsèquement liée à Epita, où chaque utilisateur est un noeud du réseau.

Notre projet s'articulera donc en ces trois axes inter-dépendants :

- **La blockchain Nexium (le réseau) :** les fonctionnalités de communication, vérifications et synchronisation des blocs.
- **L'interface utilisateur :** les fonctionnalités graphiques, de gestion de compte, de virements, et de consultation de blocs.
- **Les implémentations cryptographiques :** les fonctionnalités "backend" de chiffrement, déchiffrement, signature et vérification de données. Nous utiliserons le chiffrement RSA appliqué au standard GPG pour la cryptographie asymétrique, et le standard SHA-256 pour les fonctions de hachage. Afin de correspondre à la consigne du projet, nous implémenterons toutes les fonctions liées à cette partie nous-même, sans utiliser de bibliothèques externes (y compris pour les fonctions arithmétiques, le RSA étant basé sur la primeur de grands nombres).

2 Organisation théorique du réseau

Chaque bloc, contenant un type d'information unique, sera horodaté, signé par l'émetteur du message et validé par les autres utilisateurs du réseau. Afin de faciliter le calcul des soldes disponibles de chaque utilisateurs, ils seront calculés de manière régulière et intégrés en tant que blocs de résumés (dits `Checkpoints`). Si un membre du réseau détecte un message dont la signature est invalide, il peut le signaler aux autres membres du réseau, qui pourront alors le rejeter et marquer le bloc comme invalide.

Nous avons pour l'instant défini les types de blocs suivants, représentés en Rust:

```
enum BlockType {
    AccountCreation,
    Transaction,
    Checkpoint,
    Validation,
}
```

Définissons également les types suivants :

- `Login`: une chaîne de caractères représentant le login Epita de l'utilisateur (prenom.nom). La vérification du format de la chaîne sera faite à partir d'une expression régulière.
- `Hash`: une chaîne de caractères représentant un hash.

Tout bloc contiendra toujours au moins ces quatre champs :

- `type` indiquant le type de bloc (type: `BlockType`).
- `emitter` indiquant le login de l'émetteur du bloc. (type: `Login`)
- `timestamp` indiquant la date et l'heure de création du bloc. (type: `DateTime`)
- `id_hash` un hash du bloc, permettant de l'identifier de manière unique. Cet identifiant sera incrémental. (type: `Hash`)
- `control` une trame permettant de vérifier l'intégrité et l'authenticité du bloc. (type: `Hash`)

La trame de controle `control` sera la signature du id_hash du bloc, avec la clé privée de l'émetteur.

2.1 AccountCreation

Ce type de bloc ne sera théoriquement envoyé qu'une fois par login Epita. Il ne contiendra pas de champs supplémentaires à ceux par défaut.
Lors de la création d'un compte, un certain solde fixe de `NXM` sera crédité sur le compte de l'utilisateur. Nous définirons ultérieurement le montant de ce solde initial dans les spécifications détaillées du réseau.

2.2 Transaction

Ce type de bloc sera utilisé pour effectuer des transactions entre utilisateurs. Il contiendra les champs additionnels suivants :

- `amount` indiquant le montant de la transaction en `NXM`. (type: `u64`)
- `receiver` indiquant le login du destinataire de la transaction. (type: `Login`)
- `comment` indiquant un commentaire facultatif, chiffré avec la clé publique du destinataire. (type: `String`)

2.3 Checkpoint

Ce type de bloc sera utilisé pour stocker les soldes de chaque utilisateur. Il contiendra le champ additionnel `balances` (type: `HashMap<Login, u64>`) indiquant le solde de chaque utilisateur sous forme d'un tableau associatif.
Étant donné l'importance de ce bloc, il devra obligatoirement être validé par une certaine proportion majoritaire des utilisateurs du réseau (que nous définirons ultérieurement) avant de pouvoir être pris en compte.
L'utilisation d'un tel bloc permet de limiter le nombre de transactions à vérifier pour calculer le solde d'un utilisateur, et donc de réduire la complexité de l'algorithme de vérification des transactions.

2.4 Validation

Ce type de bloc sera utilisé pour valider ou invalider un bloc. Il contiendra les champs additionnels suivants :

- `verif_id` indiquant l'id du bloc validé ou invalidé (type: `Hash`)
- `validation` indiquant si le bloc est validé ou invalide. (type: `bool`)
- `reason` indiquant la raison de l'invalidation du bloc. (type: `String`)

Afin de ne pas voir le réseau saturé par des blocs invalidés, une suppression périodique de tous les blocs marqués comme invalides sera effectuée à chaque checkpoint.

3 Fonctionnalités du programme

3.1 Initialisation du compte

Lors du lancement de l'interface, le programme cherche, à un emplacement prédéfini, une paire de clés RSA associée à l'utilisateur. Si une paire de clé est trouvée, le programme demande à l'utilisateur de saisir son mot de passe pour déchiffrer sa clé privée.
Le programme vérifie ensuite l'existence d'un compte associé à l'utilisateur sur le réseau Nexium. Si le compte n'existe pas, un bloc de type `AccountCreation` est créé et ajouté à la blockchain, avec confirmation de l'utilisateur.

3.2 Opérations

L'utilisateur peut effectuer des transactions sortantes vers d'autres utilisateurs du réseau Nexium. Il doit saisir le montant de la transaction, le destinataire (son adresse `@epita.fr`) et un commentaire facultatif. Le commentaire est chiffré avec la clé publique du destinataire et ajouté à la transaction, de telle manière que seul le destinataire puisse le lire. Le programme demande ensuite à l'utilisateur de saisir son mot de passe pour déchiffrer sa clé privée et signer la transaction. Une fois la transaction signée, elle est ajoutée à la blockchain et diffusée à l'ensemble du réseau.

3.3 Historique des transactions

L'utilisateur peut consulter l'historique de ses transactions entrantes et sortantes associées à son compte. L'historique est récupéré depuis la blockchain et affiché sous forme d'un tableau, avec les colonnes `Date`, `Montant`, Commentaire et `Entrant/Sortant`.

3.4 Interface graphique

L'interface graphique du programme sera sûrement basée sur le framework Slint s'il nous est permis de l'utiliser. Elle devra être simple et intuitive, permettant à l'utilisateur de naviguer facilement entre les différentes fonctionnalités du programme.

L'utilisation d'un tel framework nous permettra de gagner du temps sur le développement de l'interface graphique, et de nous concentrer sur les fonctionnalités fondamentales du programme.

4 Contraintes techniques et solutions envisagées

4.1 Gestion du RSA

Le traitement mathématique des clés RSA sera effectué en Rust, sans l'utilisation de bibliothèques externes. La page Wikipédia de l'algorithme RSA nous donne une idée des opérations à effectuer pour générer une paire de clés, chiffrer et déchiffrer des données, et signer et vérifier des messages.

Or, nos recherches sur la gestion des types abstraits liés à ce type de cryptographie nous ont montré une limitation intrinsèque à Rust. En effet, le RSA est basé sur la primeur de grands nombres, et le standard de sécurité que nous avons observé lors de nos recherches nécessite de calculer deux nombres premiers (p et q) codés sur 512 bits minimum. Mais les types primaires de Rust ne permettent pas de gérer des nombres de cette taille, le maximum étant `u128` (128 bits).

Il existe bien entendu le type `BigInt` de la bibliothèque `num-bigint`, mais nous ne savons actuellement pas si nous serons autorisés à l'utiliser. Il serait très complexe d'implémenter manuellement un type U512, car sa compatibilité avec les opérations mathmatiques classiques de Rust serait à notre entière charge!

Cela impliquerait la re-défition de toutes les opérations mathématiques de base (addition, soustraction, multiplication, division, modulo, etc.) pour ce type.

Il est toutefois important de garder à l'esprit que cet aspect de Nexium ne nous est absolument pas étranger, car c'est exactement cette partie de l'arithmétique que nous avions traitée lors de notre première année à l'EPITA au travers de l'`AFIT`.

4.2 Liaison avec Gitlab

La récupération et l'envoi des clés publiques se fera grâce à l'API du Gitlab d'Épita. Ce paradigme de récupération de clés nous permet donc de placer Épita comme autorité de certification, sans altérer ni l'infrastructure déjà en place, ni la sécurité du réseau Nexium.

L'API de Gitlab nous permettra de récupérer les clés publiques des utilisateurs tel que défini sur la documentation officielle de l'API Gitlab. Voici donc les deux requêtes que nous utiliserons :

- `GET /user/keys` pour récupérer les clés publiques d'un utilisateur.
- `POST /user/keys` pour ajouter une clé publique à un utilisateur.

Deux problèmes se posent alors à nous :

- **Les clés publiques du Gitlab sont des clés GPG.** Nous devrons donc implémenter, sûrement sous forme de méthodes implémentées aux types abstraits des clés, la conversion bi-directionnelle entre les clés GPG et les clés RSA.

- **L'API Gitlab nécessite une authentification.** Il nous sera donc nécessaire de disposer d'un moyen d'authentification lié au compte Gitlab Epita de l'étudiant. Pour des raisons de sécurité, nous préférons éviter l'éventualité de récupérer et stocker les informations brutes de connexion de l'utilisateur dans le projet binaire final. De plus, ce fonctionnement de prendrait pas en compte la présence d'une double authentification sur le compte Gitlab concerné. Nous envisageons donc d'utiliser l'authentification OAuth de Gitlab, qui permet à une application tierce de se connecter à un compte Gitlab sans avoir accès aux informations de connexion de l'utilisateur. Le token obtenu par cette intégration sera chiffré par le mot de passe de l'utilisateur, et stocké localement. Nous nous pencherons ultérieurement sur la mise en place de cette authentification, et implémenterons pour l'instant une authentification basique pour le prototype.

4.3 Stockage et sécurisation de clés

Lors de la génération de la paire de clés RSA, un mot de passe sera demandé à l'utilisateur. Le programme générera ensuite un **hash** de ce mot de passe, qui sera utilisé pour chiffrer la clé privée de l'utilisateur. La clé privée chiffrée sera ensuite stockée localement, et la clé publique sera envoyée sur le Gitlab d'Épita.

Afin de garantir la meilleure étanchéïté possible, nous ne stockerons jamais ni le **mot de passe**, ni son **hash**, ni la **clé privée** sur le disque. Grâce à la *memory safety* de Rust, nous n'avons pas à nous soucier de la suppression de ces données, ou de leur utilisation par un autre programme : le *borrow checker* nous assure que ces données ne sont jamais utilisées en dehors de leur portée.