TABLE DES MATIÈRES

Problématique

La démocratisation de l'utilisation de gestionnaires de mots de passe (MDP) est freinée par la nécessité de recours à un serveur tiers pour le stockage. Le but de ce projet est d'étudier un modèle de gestionnaire de MDP qui limite le coût en capacité de calcul et en stockage d'un serveur centralisé.

Objectifs du projet

Ce projet vise donc à répondre à la problématique précédente en concevant un gestionnaire de MDP (GMP) qui nécessite moins de capacité de calcul et de stockage pour un serveur centralisé. Ceci est dans l'optique d'avoir un GMP qui soit gratuit pour les utilisateurs et à moindre coût pour l'hébergeur afin de démocratiser l'utilisation de ces derniers. Un tel GMP doit donc utiliser des protocoles légers et ne se basant pas entièrement sur une machine centrale qui gère l'ensemble du système. Le GMP doit assurer ces fonctions tout en étant simple d'utilisation et très sécurisé. L'étude théorique a donc permit de définir les fonctionnalités desservant ces contraintes :

- Stockage décentralisé et sécurisé de fichiers
- Transmission sécurisé des fichiers entre plusieurs machines
- Synchronisation et versionning des fichiers
- Gestion simple des appareils utilisateurs

Des contraintes moins “métiers” sont en ambition du projet, à savoir :

- Robustesse de la solution (multi-plateforme)
- Adaptations aux diverses architectures réseaux (diverses méthodes NAT)

Le projet ne s'intéressera qu'aux problématiques autours de ces fonctionnalités et n'ira pas étudier en profondeur les fonctionnalités élémentaires d'un GMP comme le chiffrement local des fichiers. Il s'agit donc avant tout de réaliser une preuve de concept fonctionnelle sur les fonctionnalités et caractéristiques visées.

Les première solutions qui peuvent apparaître sont l'utilisation d'un système décentralisé comme le Peer-to-peer. À cela viennent s'ajouter des questions sur la synchronicité des communications entre les appareils d'un utilisateur. Tout cela, ainsi que les solutions de GMP déjà existantes, sera donc étudié dans la veille technologique.

Veille technologique

Les gestionnaires de mot de passe

Actuellement, diverses solutions de GMP existent déjà. Nous avons donc fait le tour de celles-ci afin de voir si des solutions répondant à nos contraintes existaient déjà. Nous avons pu observer qu'il en existe différents types :

Local *Exemple : Keepass*

Tous les identifiants/MDP entrés dans le gestionnaire sont chiffrés avec un MDP maître dans un fichier. Ce fichier est stocké sur le disque dur. L'accès à ce fichier est donc protégé par un secret connu uniquement par l'utilisateur.
Toutefois il faut noter que mis à part une copie manuelle du fichier d'un PC à un autre, l'utilisation des MDP est restreinte au PC où le fichier chiffré est stocké. Il existe des extensions permettant de synchroniser le fichier sur un drive comme Dropbox, or cela complique beaucoup l'utilisation de l'outil.

Online *Exemple : Dashlane*

Ici, l'ensemble des MDP est centralisé dans une base de données sur un serveur public. Le fonctionnement est similaire à celui décrit précédemment, les MDP sont chiffrés localement sur une machine avec une clé maître et génère un fichier chiffré ensuite stocké dans un serveur.

Sans états *Exemple : Lesspass*

Une autre solution consiste à calculer le MDP à chaque utilisation afin de ne pas le stocker.
Une fonction déterministe prend en paramètre un MDP maître, l'URL du site, l'identifiant, l'adresse mail etc… puis calcule le MDP en local, sur la machine. L'utilisateur doit donc se souvenir de tous ces paramètres. Il est impossible de stocker quoi que ce soit d'autre, MDP ou notes sécurisées.

Blockchain *Exemple : You.*

Certaines solutions proposent d'utiliser des blockchains pour stocker le fichier de MDP. Cette méthode est intéressante car elle utilise un système complètement décentralisé et accessible à tous. Cependant, le fichier de MDP est alors publiquement accessible sur la blockchain et dépend de la Blockchain utilisée. À ce jour, les Blockchain sont trop instables car les Cryptomonnaies sont basées sur cette technologie. Utiliser une Blockchain reviendrait donc à faire fructifier une Cryptomonnaie, ce qui n'est pas notre but.

Bilan

Pour résumer, le modèle Online apporte toutes les fonctionnalités que l'on recherche si l'on omet les contraintes sur la liberté d'utilisation et le coût des infrastructures. Le modèle de Blockchain apporte justement des solutions à ces contraintes mais nous ne pouvons l'utiliser. Nous allons donc nous inspirer de ces modèles qui nous intéressent pour avoir une base avant de trouver les solutions qui nous conviennent.

Les architectures Peer-to-peer

Principe

Le P2P est un modèle de réseau différent du modèle client-serveur. Dans ce modèle toutes les machines du réseau sont au même "niveau". Chaque machine est à la fois client et serveur.

Centralisé

Sur ce modèle, il y a tout de même un serveur central qui héberge un index des ressources du réseau P2P. Il est donc rapide pour un pair de chercher dans cet index quelle ressource est sur quel pair.

Décentralisé et structuré

On peut aussi décentraliser un index des ressources entre les nœuds du réseau. Les pairs ont juste à partager l'information sur la topologie et l'emplacement des ressources. Par contre, un tel modèle devient vite complexe et a une structure assez rigide qui amène des résultats de recherche erronés.

Décentralisé et non-structuré

On peut aussi ne pas avoir d'index du tout sur le réseau P2P. Il n'y a donc aucun contrôle sur la topologie ou l'emplacement des ressources. Dans ce cas les pairs doivent faire des requêtes en Broadcast à tous les pairs dans un certains "rayons". Ce modèle est très robuste mais plus il y a de pairs, moins il est efficace.

Les communication synchrone et asynchrone

Dans la majorité des réseaux pair à pair, lorsqu'un échange de données se fait entre les pairs, on a une connexion synchrone. C'est à dire que les deux pairs qui communiquent sont tous les deux allumés pendant la communication et donc que le transfert d'information est limité aux appareils connectés en même temps au réseau.

Or, vu l'usage qui sera fait de notre GMP, on souhaite transférer les mises à jour d'un fichier sans toutefois que les autres appareils soient allumés. Pour cela nous allons donc mettre en place un système asynchrone où le serveur ne stockera pas indéfiniment les données mais de manière temporaire comme le fonctionnement d'une boîte au lettre.

Nous avons donc étudié le protocole ***Sésame*** de Signal qui répond en bonne partie à nos contraintes. Celui-ci part en effet du principe qu'il y a un serveur qui stock plusieurs utilisateurs ainsi que les appareils de tous ces utilisateurs. Le serveur stocke aussi temporairement les messages que les appareils s'envoient jusqu'à ce qu'ils soient récupérés. Il y a donc un système de boîte au lettre pour chaque appareil où le serveur stock les messages à destination de cet appareil. L'appareil destinataire peut donc à tout moment, de manière asynchrone, récupérer ses messages.

Au délà de ça, ***Sésame*** propose d'établir une session dans laquelle les messages peuvent être chiffrés et déchiffrés avec un secret correspondant à la session en cours et détenu par un appareil. Ainsi, un message chiffré ne peut être déchiffré que par un appareil qui est dans la bonne session. Les secrets des précédentes sessions seraient ainsi supprimées pour assurer le principe de forward secrecy. Cela veut dire que la découverte du secret d'une session ne compromet pas les messages d'une ancienne session.

Nous sommes donc partie des briques proposées par ***Sésame*** qui nous intéressait pour ajouter ce qui nous manquait et ainsi créer notre propre protocole de communication asynchrone.

Le protocole EsiPass

Communication clients - serveur

Contextuellement, nous avons des utilisateurs avec plusieurs appareils. Sur un appareil sont stockés localement des fichiers de MDP qui sont chiffrés à l'aide d'un MDP maître, secret et seulement connu par l'utilisateur. Ces MDP doivent être synchronisés entre les appareils d'un utilisateur et ce de manière sécurisée.

Toutes les communications se feront donc à travers un serveur qui servira uniquement de relai de communication entre les différents appareils. Il faut donc que les appareils soient enregistrés auprès d'un tel serveur. Voici un diagramme d'échange montrant comment un nouvel appareil peut s'enregistrer auprès d'un serveur :

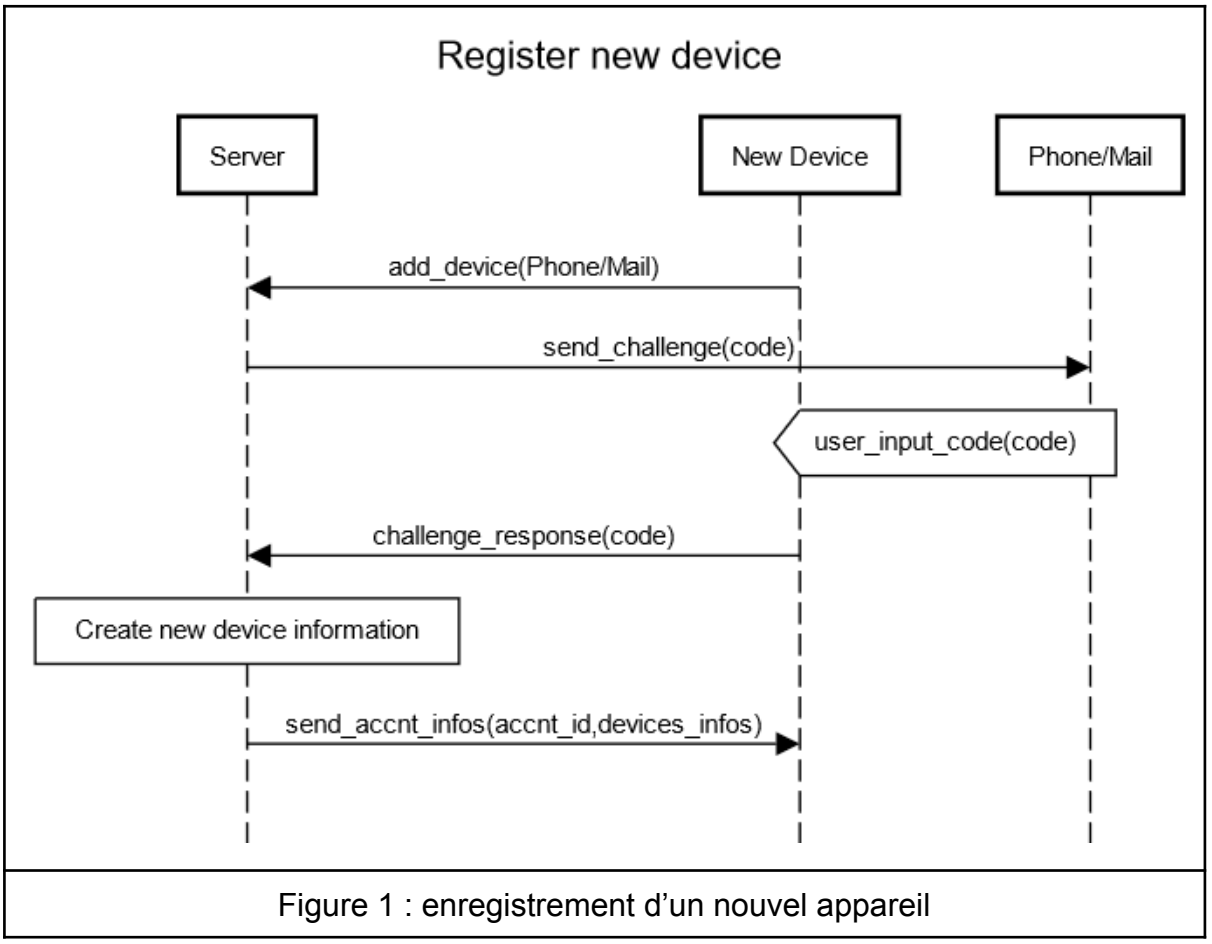

Figure 1 : enregistrement d'un nouvel appareil

L'utilisateur sur son nouvel appareil va s'identifier avec un numéro de téléphone ou une adresse mail auprès du serveur. Cet identifiant sera alors challengé pour que l'utilisateur puisse prouver son identité et l'associer au nouvel appareil. Si le challenge est validé, l'appareil est enregistré par le serveur qui lui répond avec les informations du compte et des autres appareils déjà existants.

Une fois que les appareils sont enregistrés sur le serveur, ils peuvent s'échanger des MDP. Pour que cela se fasse de manière sécurisée, les échanges entre les appareils sont chiffrés avec une clé de transfert commune à tous mais, qui change à chaque message envoyé. Ainsi, à l'instar de Sésame, le forward secrecy est assuré dans les échanges entre clients.
Pour synchroniser cette clé de transfert entre les clients, elle est dérivée d'une clé de session partagée entre tous les appareils. Cette clé de session est de temps en temps renouvelée selon certains critères afin d'assurer le backward secrecy (ou future secrecy) entre les différentes sessions. C'est à dire que si une clé de transfert ou une clé de session est compromise, alors cela ne compromet que les messages futurs issues de la même clé de session mais pas les messages utilisant une clé de session future. En effet, elle est générée aléatoirement et ne peut donc être déterminée à partir des anciennes clés de session.

Voici donc un diagramme d'échange montrant comment un appareil peut générer une nouvelle clé de session et la partager aux autres appareils de manière sécurisé :

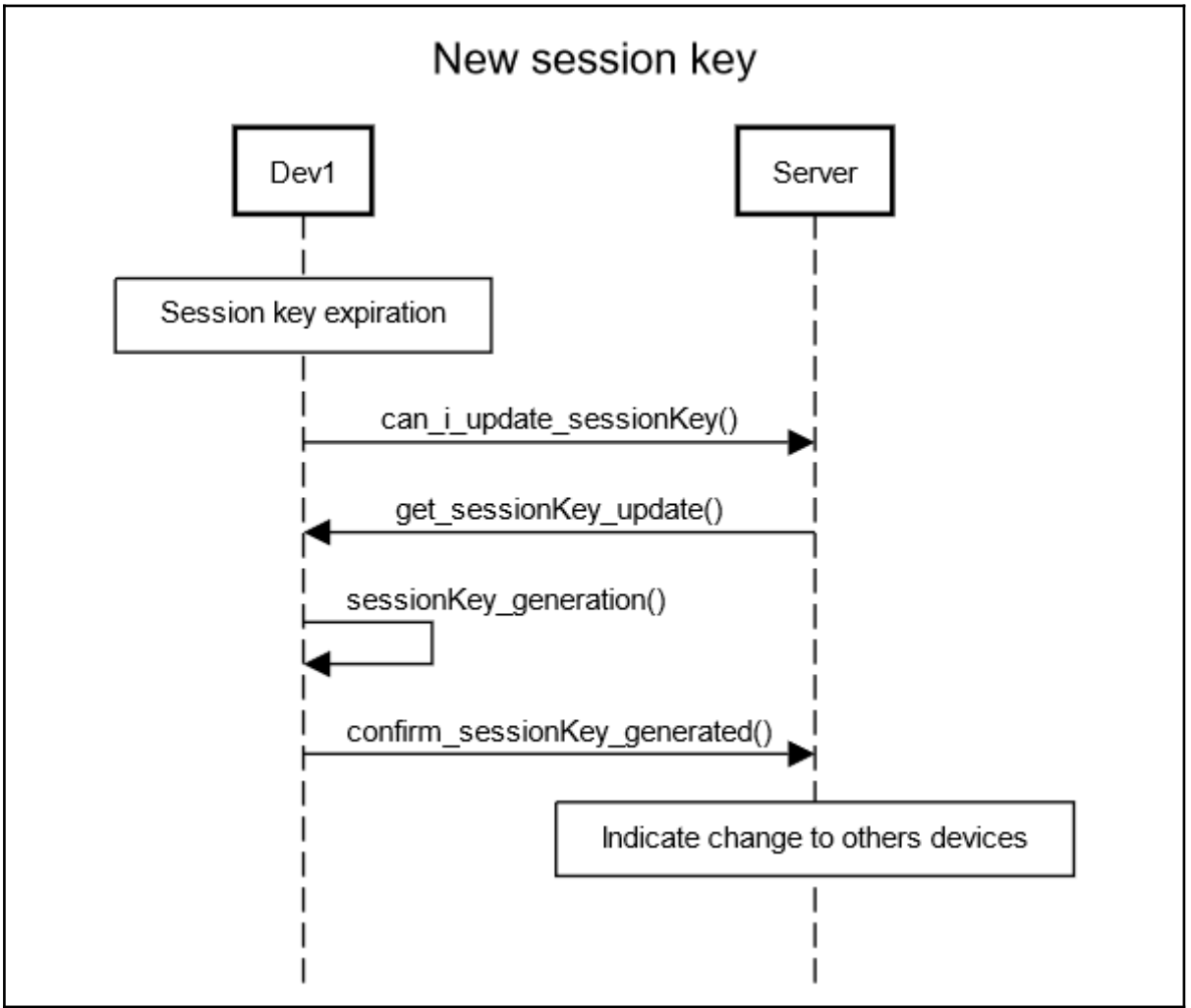

Figure 2 : renouvellement de la clé de session

Ici, la clé de session n'est donc pas envoyée au serveur, seulement une notification de changement pour les autres appareils. Le serveur doit donc noter que les autres appareils n'ont plus leur clé de session à jour et transmettre la notification quand ces derniers viennent récupérer des informations auprès du serveur.
Voici un diagramme d'échange montrant comment un appareil n'ayant plus sa clé de session à jour peut se resynchroniser :

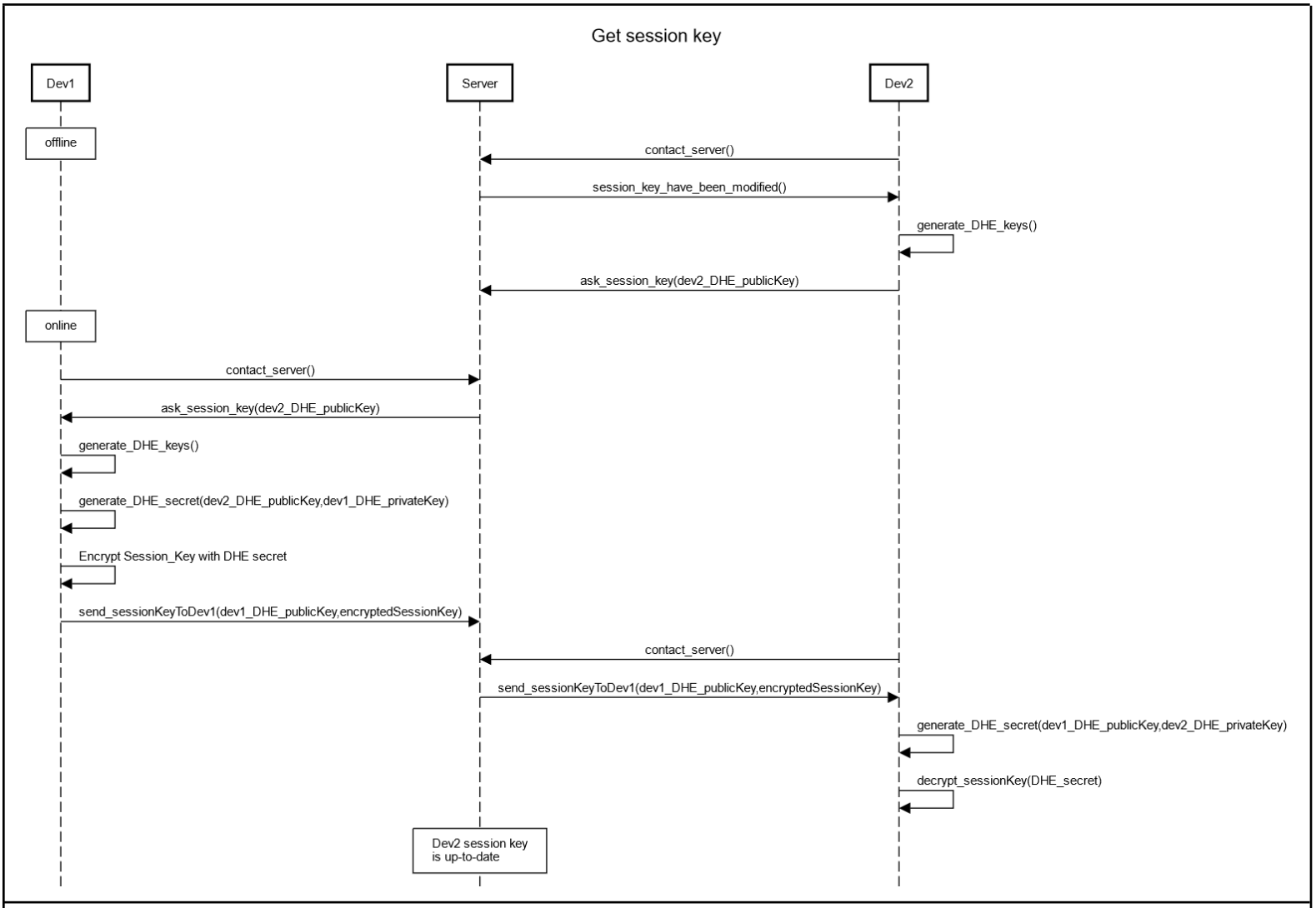

Figure 3 : récupération de la clé de session

On peut donc voir que quand un client souhaite récupérer la dernière clé de session, il fait une demande pour initier un échange Diffie-Hellman (DHE). Le secret partagé issu du DHE est alors utilisé pour chiffrer de manière symétrique la clé de session par l'appareil expéditeur. Une fois la clé de session synchronisée entre les clients, il peuvent donc l'utiliser dans une fonction de dérivation de clé (Key Derivation Function - KDF) pour calculer une clé de transfert. Cette dernière est à usage unique pour le chiffrement et le déchiffrement d'un seul et même message.

Ainsi on arrive presque au fonctionnement du Double Ratchet du protocole de Signal. La différence étant que dans le protocole Signal, la clé de session est initialisée par un triple échange étendu Diffie-Hellman (X3DHE) puis est mise à jour pour chaque message par un simple DHE. Pour EsiPass la fréquence d'envoi de message est bien moindre et donc la clé de session n'est réinitialisée que périodiquement. En effet, comme il y a moins de message, cela donne moins d'information à un potentiel attaquant, rendant plus difficile de déterminer la clé de session. Le protocole Signal génère à l'avance, pour chaque appareil, un certain nombre de paires de clé qui sont stockées sur le serveur afin de pouvoir réaliser des simples DHE de manière asynchrone à chaque message. Étant donné qu'un des objectifs du projet est d'avoir un serveur le plus léger possible (puissance et stockage), l'utilisation d'une clé de session permet de ne pas avoir à garder toutes ces clés pour chaque appareil sur le serveur mais surtout de ne pas à avoir à réaliser un chiffrement et envoi pour chaque autre appareils à chaque mise à jour de MDP, car un échange DHE ne permet d'avoir un secret partagé qu'entre 2 machines. Il faut donc réaliser le DHE avec chaque autre appareil et avoir sur le serveur un fichier en attente pour chaque appareil également. Avec le partage d'une clé de session, tous les appareils ont un secret commun qui peut être utilisé pour chiffrer les mises à jour de MDP. Le serveur n'a donc plus qu'à les stocker qu'une seule fois pour tous les appareils.

La clé de transfert peut donc être utilisé pour l'envoi d'un MDP comme sur cet échange :

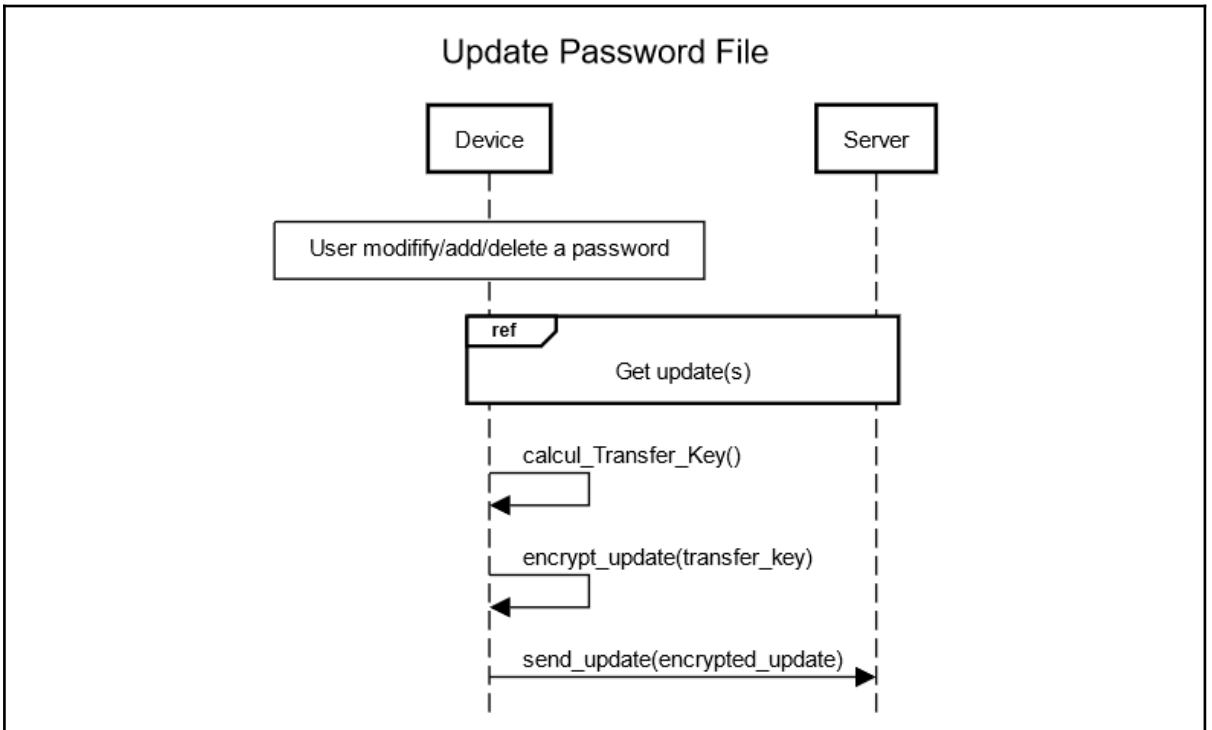

Figure 4 : mise à jour du fichier de MDP

Il est en effet nécessaire que l'appareil récupère avant tout, les potentiels messages qui l'attendent sur le serveur pour notamment avoir sa clé de session et sa clé de transfert à jour.

Si on regarde maintenant la réception de la mise à jour d'un MDP :

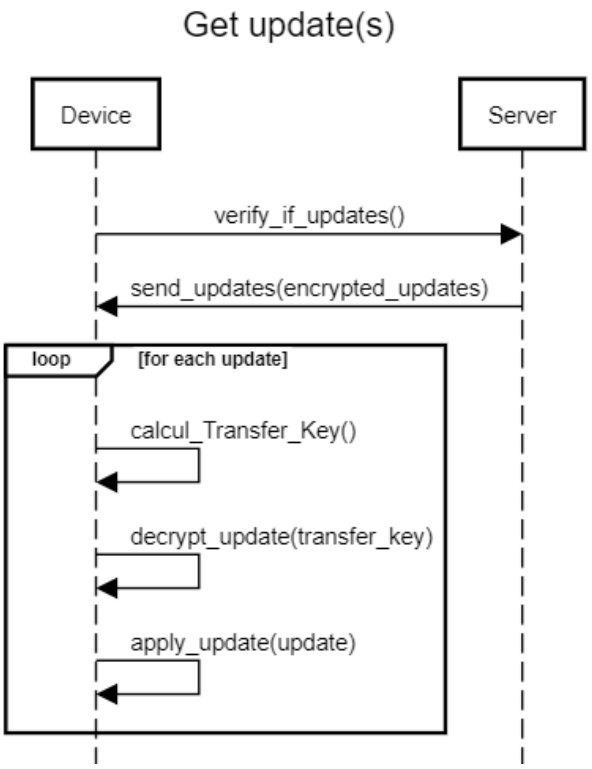

Figure 5 : récupération de mises à jour de MDP

On retrouve donc le même mécanisme à l'envoi et à la réception. Pour éviter de consulter le serveur à chaque fois avant l'envoi d'une mise à jour, on pourrait, comme dans le Signal protocole, utiliser une solution de clé d'envoi et de clé de réception. Cependant, il faudrait donc une clé de réception pour chaque appareil en face, ce qui rend le processus bien plus lourd. En effet, nous avons ici des échanges entre plusieurs appareils et non juste deux interlocuteurs comme sur Signal.

De manière générale, la sécurisation des communications entre plus de deux interlocuteurs est très complexe. D'autant plus lorsque la communication est asynchrone. (Voir "Asynchronous Group Messaging with Strong Security Guarantees"). Heureusement, il n'est pas essentiel comme sur une application de messagerie de récupérer l'ensemble des messages entre nos appareils. Ainsi on évite d'avoir des clés d'expéditeur comme dans le protocole de messagerie de groupe de Signal où elles ne changent pas, ce qui casse le principe du Double Ratchet utilisé sur les communications à deux pairs.

Chiffrement des mots de passe

Le protocole Esipass implique deux étapes de chiffrement/déchiffrement. Un chiffrement doit avoir lieu au niveau des MDP locaux à un appareil avec un secret dérivé du MDP maître et un second avec la clé de transfert lorsqu'une mise à jour de MDP transite sur le serveur.

Le choix du premier algorithme de chiffrement s'est orienté vers l'algorithme ChaCha20. ChaCha20 est un algorithme de chiffrement symétrique à clé secrète, considéré comme plus rapide que AES notamment dû à son mode de fonctionnement (chiffrement par flux). Il est considéré comme sûr et est utilisé dans des protocoles de sécurité tels que TLS, SSH, et IPsec.

ChaCha20 utilise pour son chiffrement une valeur initiale appelée nonce permettant d'initialiser l'algorithme, cette valeur est combinée avec une clé dérivée du MDP maître pour produire une sortie de chiffrement unique pour chaque message et ainsi rendre le chiffrement non déterministe. Le nonce est un paramètre qui peut être publique, toutefois, il doit être suffisamment grand pour être considéré comme sûr de le générer aléatoirement. C'est pourquoi dans notre implémentation nous avons choisi une taille de 24 octets correspondant en réalité à l'algorithme XChaCha20. Le nonce est renouvelé à chaque chiffrement de fichier de MDP et est **communiqué** dans le message de mise à jour de MDP envoyé au serveur.

Pour dériver le MDP maître nous utilisons l'algorithme PBKDF2 (Password-Based Key Derivation Function) résistant aux attaques par dictionnaire et rainbow table.

Le deuxième chiffrement, lorsque la mise à jour de MDP circule sur le serveur, utilise l'algorithme AES-256 bits car il est réputé sûr et est très largement utilisé. La communauté scientifique considère l'AES-256 bits comme résistant aux futures attaques quantiques. Contrairement à l'utilisation de l'algorithme ChaCha20, nous utilisons une valeur initiale constante pour chaque chiffrement car la clé de transfert n'est jamais identique mais calculée à chaque envoie de mise à jour via le serveur.

Implémentation du protocole

Choix techniques

Le prototype a été développé avec le langage Rust. C'est un langage de programmation open-source, multi-paradigme. Sa conception a pour objectif de répondre à un besoin de sûreté notamment grâce à sa sécurité mémoire et son typage fort. Lors du 1er semestre de la 5ème année nous avons été introduit à l'utilisation de ce langage, nous voulions donc approfondir nos compétences sur ce langage. Toutefois, la principale raison de ce choix reste son module cryptographique complet, utilisé par des millions d'utilisateurs.

La réalisation d'opérations de chiffrement avec l'algorithme ChaCha20 est réalisée avec la crate (librairie) chacha20 :

https://crates.io/crates/chacha20

L'utilisation de l'algorithme AES-256 bits se fait par l'intermédiaire de la crate aes-gcm-siv :

https://crates.io/crates/aes-gcm-siv

Enfin, les calculs DHE utilisent la crate k256 :

https://crates.io/crates/k256

Architecture serveur

L'implémentation du projet ***EsiPass*** se découpe en 3 packages :

- **client** : application manipulée par les utilisateurs
- **server** : assure la communication entre les appareils des clients
- **utils** : structures communes aux package client et serveur

Le serveur comprend 6 modules principaux :

account - dhe - notifs - parse - password - session_key

Sa structure suit le diagramme de classe suivant :

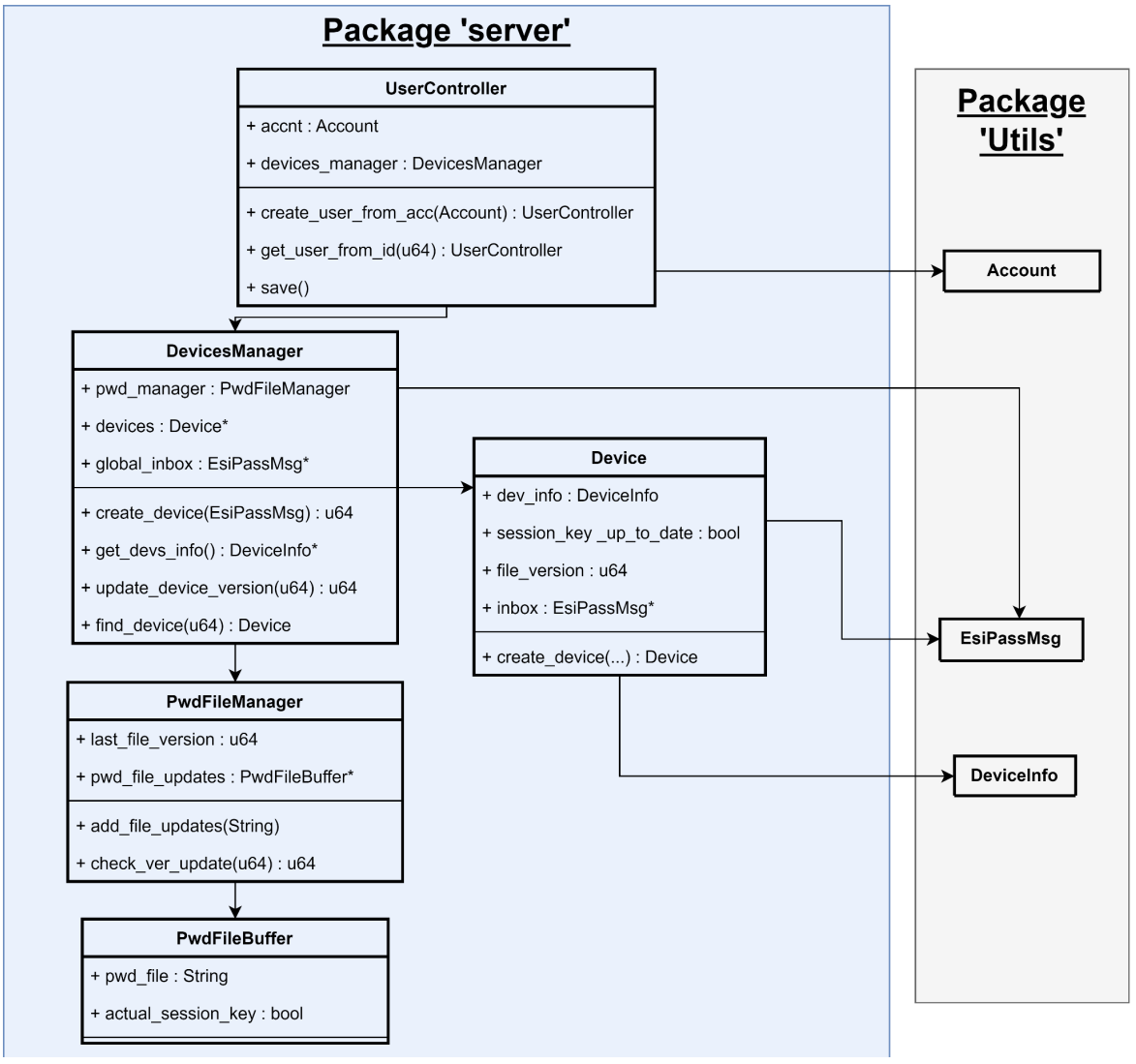

Figure 6 : diagramme de classe du serveur

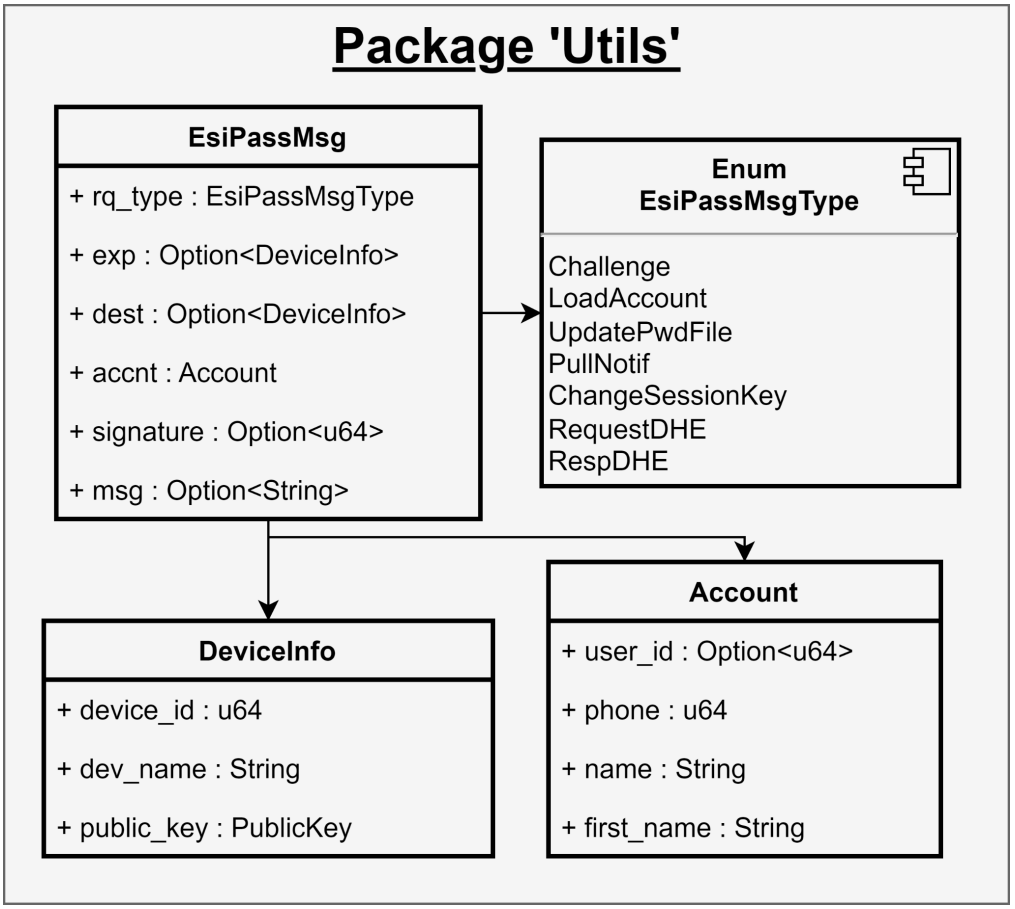

Figure 7 : diagramme de classe du package utils

Chaque client est associé à un ***UserController***, le serveur utilise cette classe pour gérer les clients et leurs appareils.

Une classe ***account*** contient les informations qui permettent d'identifier un client.

La classe ***DevicesManager*** contient une liste de tous les appareils du compte et une "boîte aux lettres" commune à tous les appareils, utilisée pour initier un échange DHE ou pour renouveler la clé de session. L'inbox général sert pour les communications ne nécessitant pas un pair particulier mais un pair répondant à des critères suffisants. Ainsi, la première opportunité est utilisée, évitant d'attendre un appareil qui pourrait ne plus jamais se reconnecter. Enfin, une classe ***PwdFileManager*** est dédiée à la gestion des MDP.

La classe ***Device*** permet au serveur de savoir si un appareil est à jour sur les MDP qu'il contient et sur sa clé de session. Une "boîte aux lettres" personnelle sert à répondre à une requête DHE ou à renvoyer tous les MDP à un appareil précis lorsque ce dernier est désynchronisé (s'il est resté déconnecté trop longtemps par exemple).

La classe ***PwdFileManager*** permet de sauvegarder la dernière version du fichier de MDP sur tous les appareils. Elle contient également une liste de ***PwdFileBuffer***, où les mises à jour apportées aux MDP sont stockées en attendant d'être récupérées par tous les appareils.

La classe ***PwdFileBuffer*** associe une mise à jour apportée au fichier de MDP, sous la forme d'un objet ***EsiPassFile*** sérialisé en chaîne de caractères, avec un flag qui permet de savoir si elle a été chiffrée avec la clé de session actuelle ou une ancienne. Ce flag est utilisé lorsque la clé de session est modifiée avant que tous les appareils aient récupéré les mises à jour en attente. Il n'est pas nécessaire de renvoyer ces mises à jour chiffrées avec la nouvelle clé car elles ne restent sur le serveur qu'un temps limité (non spécifié dans le prototype).

Voici l'arborescence détaillée du serveur :

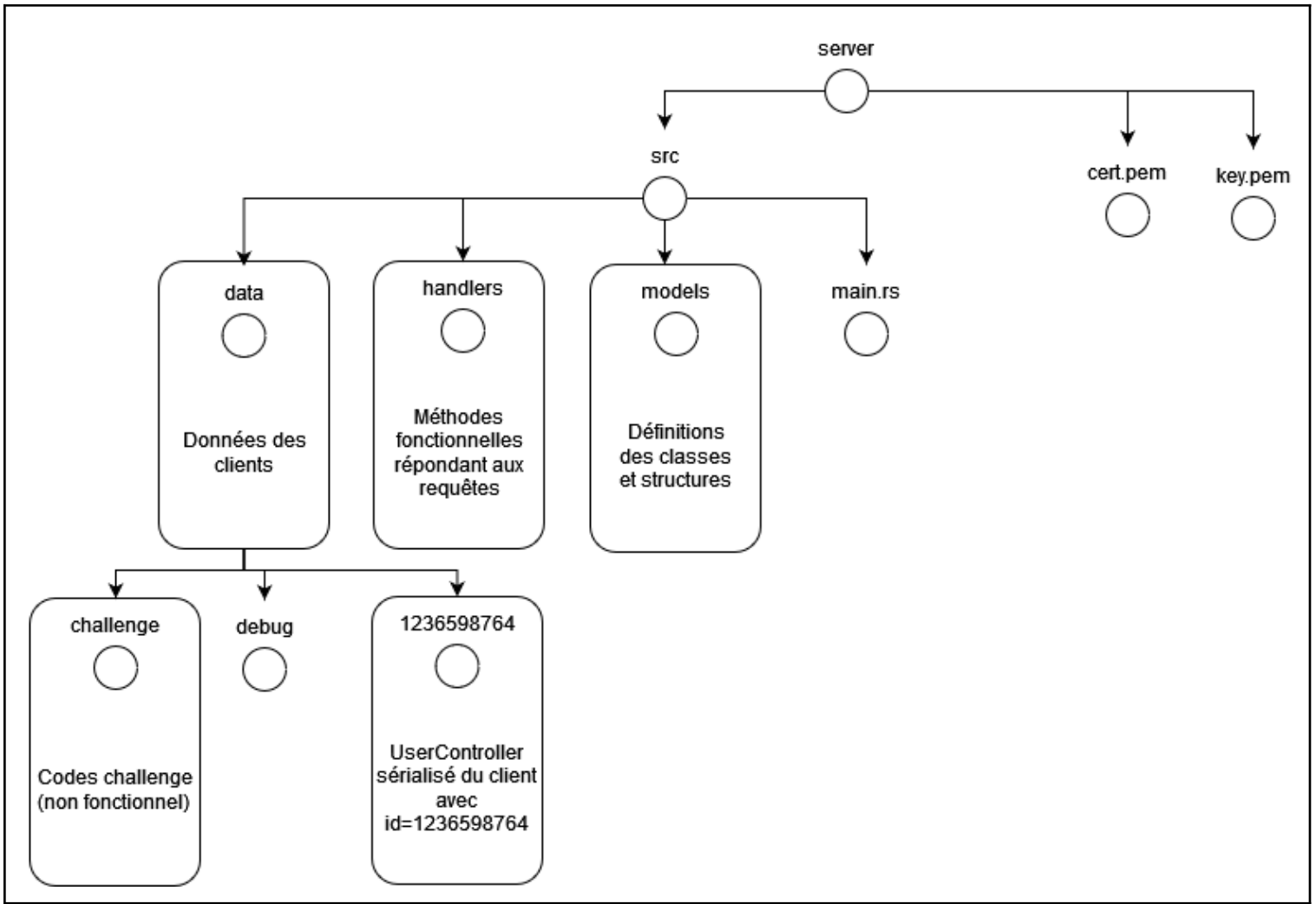

Figure 8 : arborescence du serveur

Architecture client

Le client comprend 5 modules principaux :

account_manager - client_request - crypto - display - pwd_handler

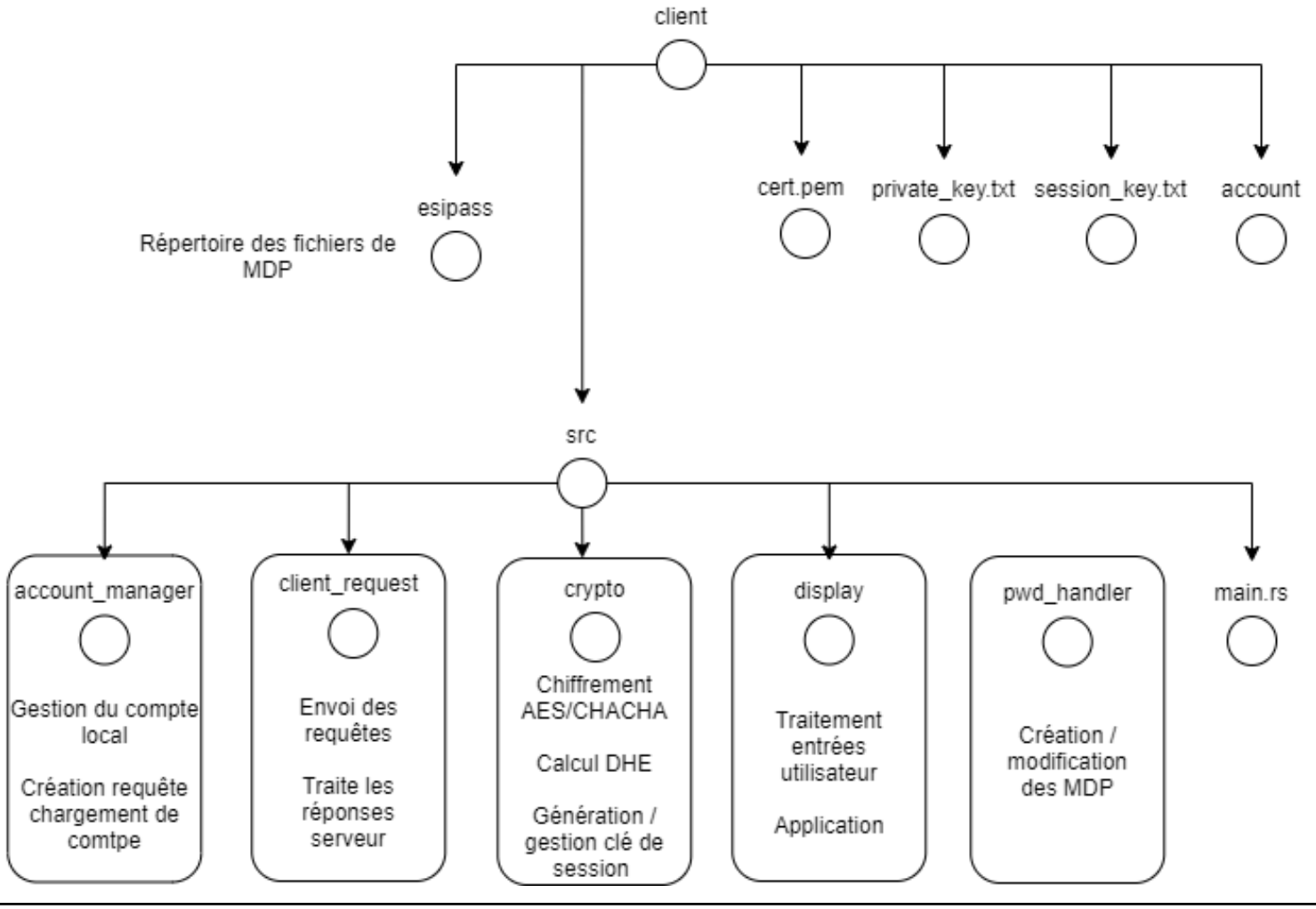

Figure 9 : arborescence du client

Tous les fichiers de MDP sont stockés/chiffrés avec un secret dérivé du MDP dans le répertoire ***esipass***.

Diagrammes d'échanges

Afin de mieux comprendre les appels de méthodes selon les cas d'utilisation, les diagrammes d'échanges ont été adaptés avec notre implémentation du protocole ***Esipass***.

Pour commencer, nous reprenons les premières fonctionnalités appelées par un utilisateur :

- La création d'un compte
- L'ajout d'un nouvel appareil
- Le chargement des informations du compte

Voici un diagramme d'échange montrant les appels de fonctions de notre implémentation :

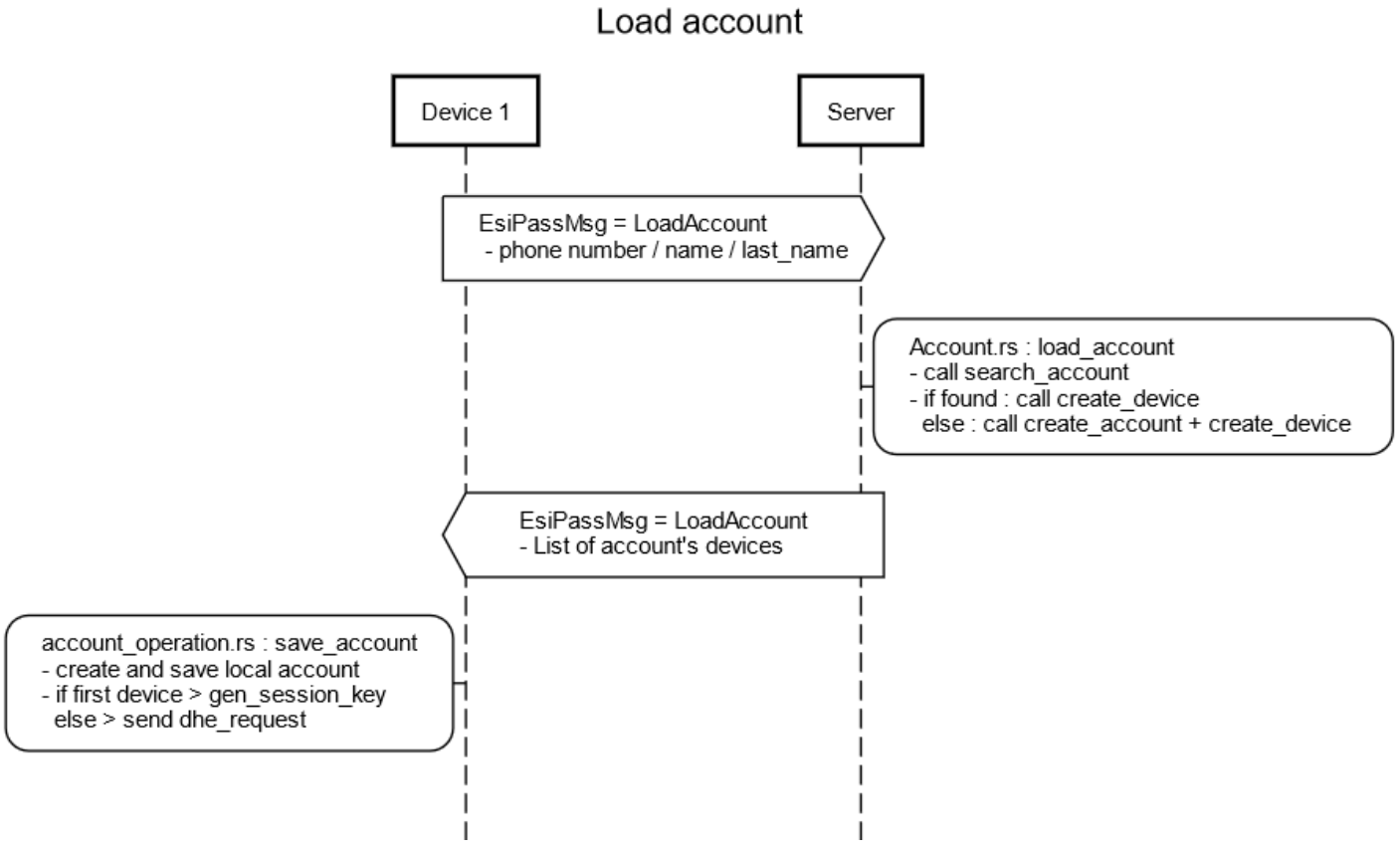

Figure 10 : chargement ou création de compte

Le type ***LoadAccount*** est utilisé à la fois lors de la création de compte et lors de son chargement. La présence de ce dernier sur le serveur détermine la fonction réalisée.

À noter qu'il y a aussi 2 possibilités côté client :

- Dans le cas d'une création de compte, une structure ***LocalAccountData*** est créée et sérialisée dans le fichier account à la racine du package client. Une clé de session est automatiquement créée afin de garder un fonctionnement 'stateless' entre les requêtes.
- Si c'est un chargement de compte, la paire de clés diffie-hellman est générée et la clé publique est envoyée dans une requête d'initialisation de DHE.

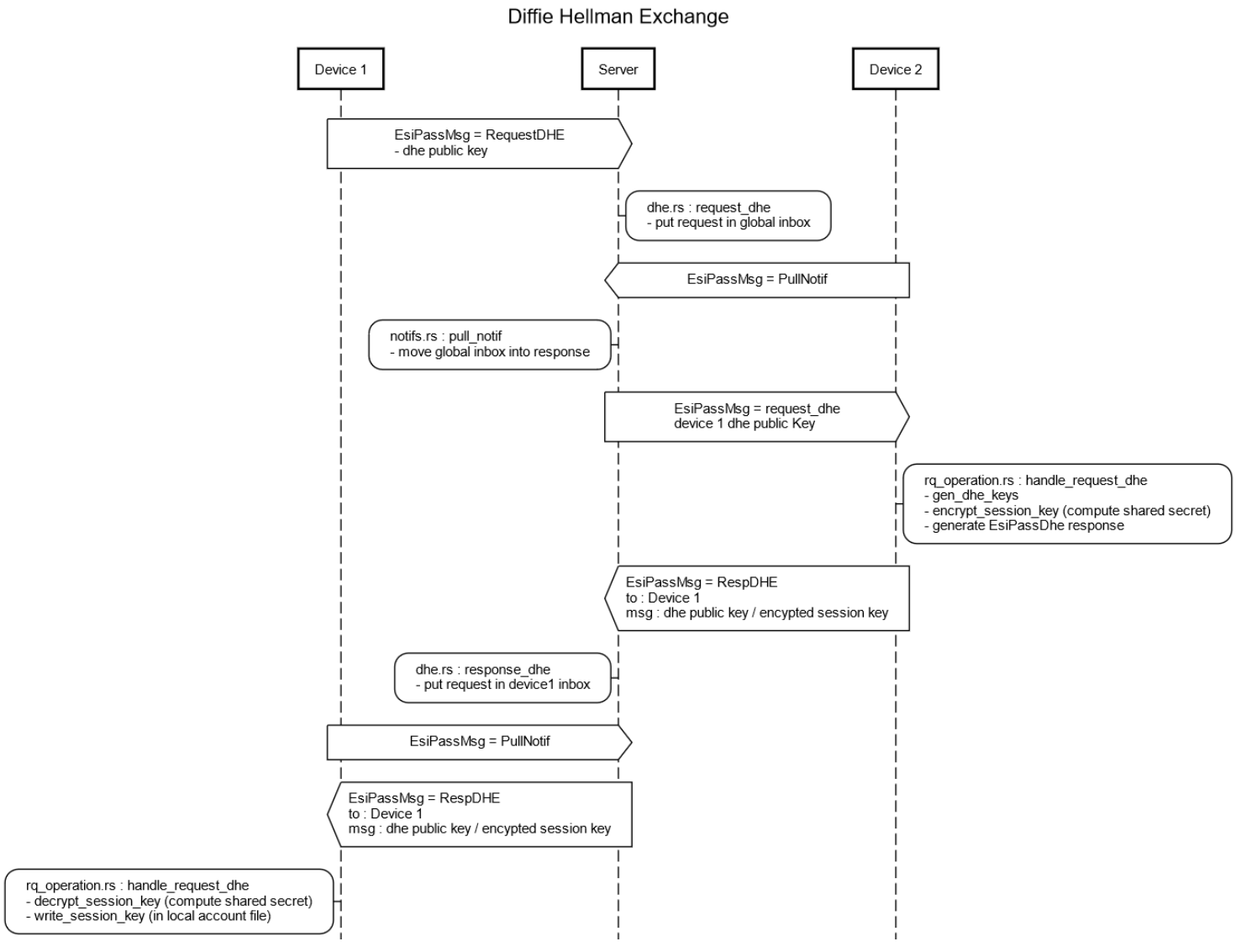

Figure 11 : procédure d'échange diffie hellman

Lors de notre DHE, il est intéressant de transmettre la clé de session dans la réponse du Device 2 afin d'éviter une requête supplémentaire, car l'objectif principal du DHE est de partager cette clé de session. Ainsi la clé de session est déchiffrée juste après avoir calculé le secret partagé.

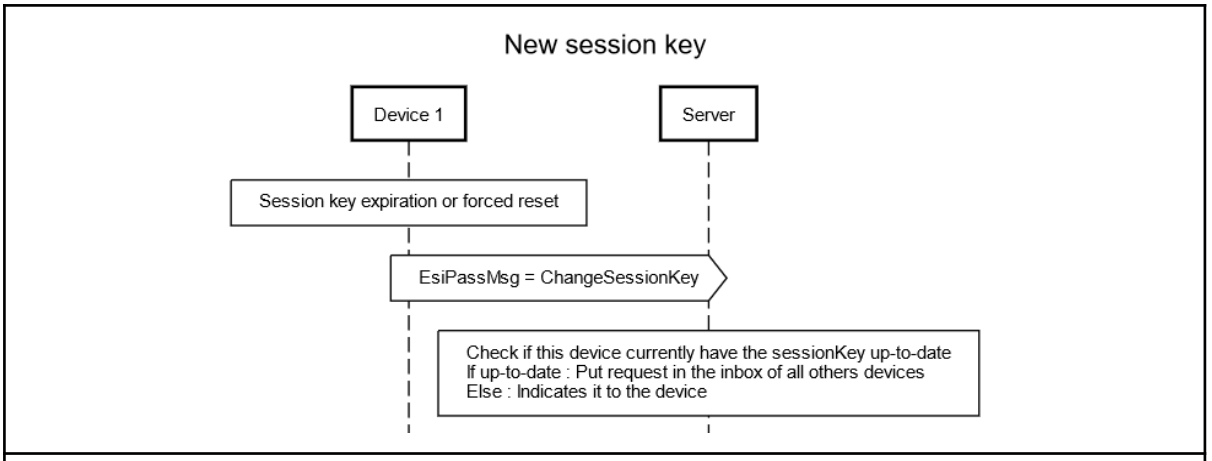

Figure 12 : renouvellement de la clé de session

L'application cliente doit vérifier la durée de vie de la clé de session (non définie dans le prototype) et finir par demander son renouvellement. Le serveur accepte la demande de changement de clé de session uniquement si l'appareil en question possédait une clé de session déjà à jour, ainsi, deux appareils ne peuvent pas modifier la clé de session successivement.

Si le changement de la clé de session est refusé, c'est qu'un autre appareil à déjà commencé la démarche. La clé de session sera alors mise à jour localement à l'issue d'un ***pull_notif***.

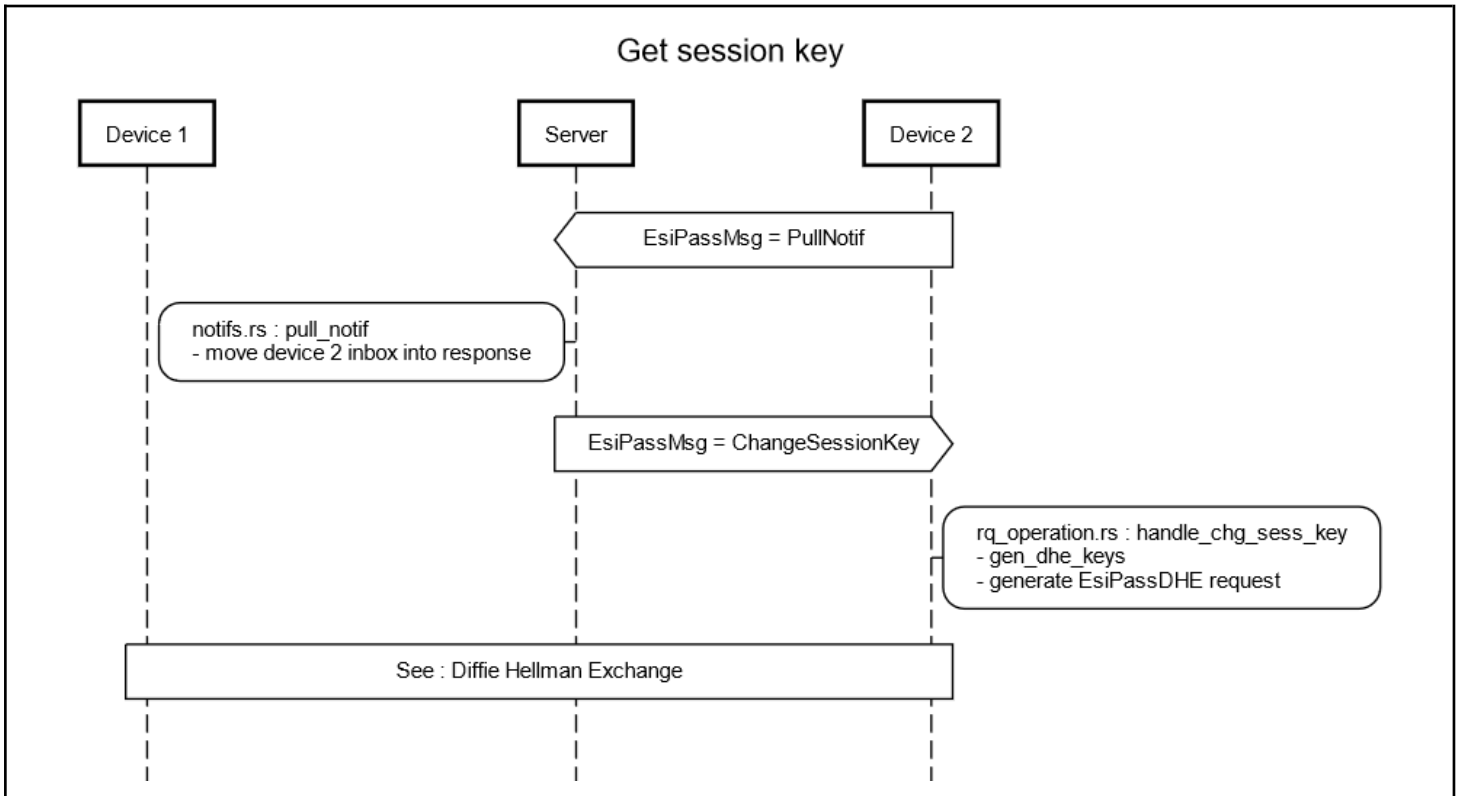

Figure 13 : récupération de la clé de session

Le serveur maintient un flag ***session_key_up_to_date*** pour tous les appareils. Celui du device 2 passe donc de “false à “true” à la fin de cet échange. Ici, Device 1 est le pair de l’échange Diffie Hellman, cela signifie que sa clé de session est à jour.

Regardons maintenant les mises à jour de MDP et comment ces actions sont transférées aux autres clients via le serveur.

Voici un diagramme d’échange montrant la trace des appels de fonction lorsqu’un client veut pousser une mise à jour au serveur :

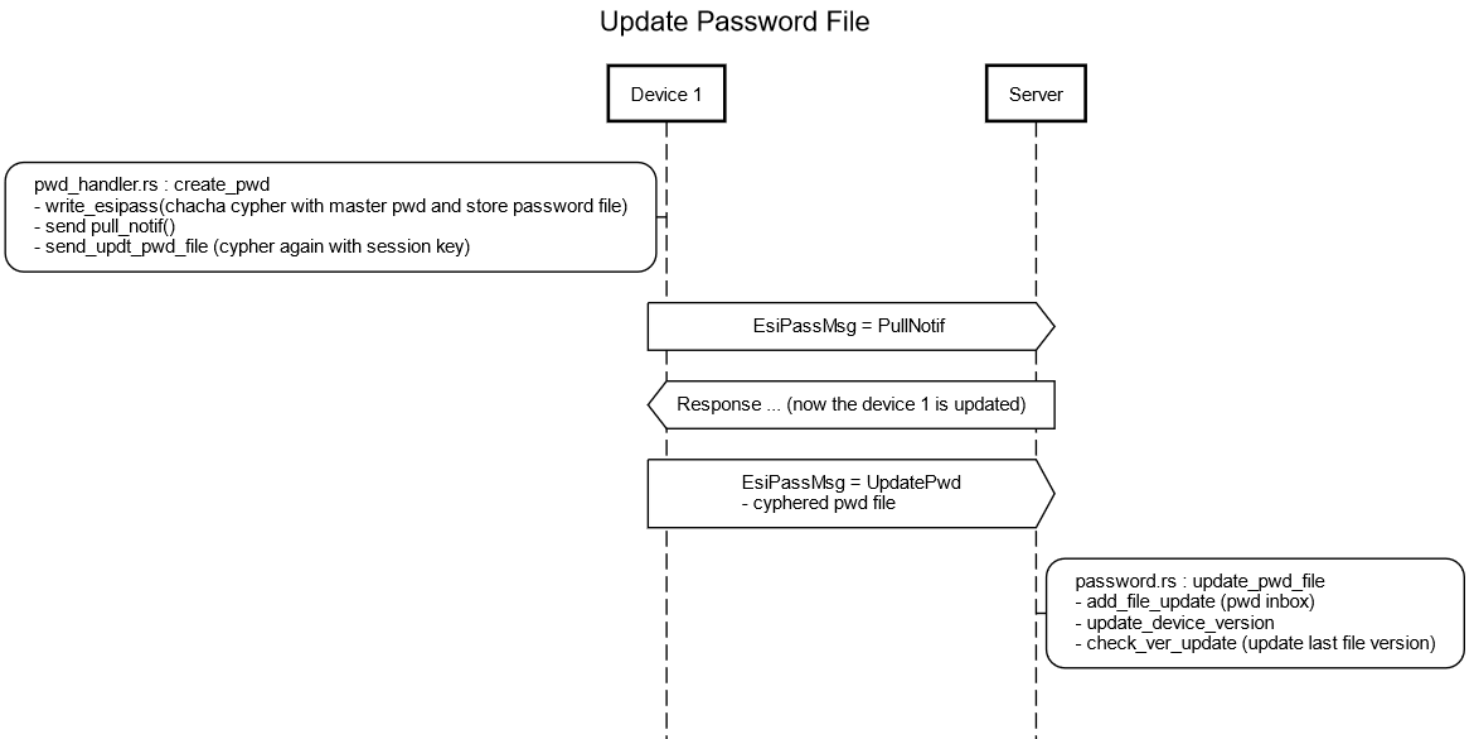

Figure 14 : envoi de mise à jour de MDP

Avant d’être envoyée sur le serveur une mise à jour de MDP est chiffrée à l’aide du MDP maître puis sauvegardée dans le répertoire ***Esipass***. La mise à jour est alors de nouveau chiffrée par une clé de transfert.

Le ***pull_notif*** avant l’envoi de la mise à jour de MDP permet d’éviter les problèmes de désynchronisation du versionnement du fichier de MDP.

En l’état actuel du serveur, il n’est pas vérifié que l’appareil qui apporte une mise à jour de MDP est bien à jour. Cela peut poser problème dans le cas où, entre le ***pull_notif*** et l’envoi de la mise à jour, un autre appareil réalise lui aussi une mise à jour de MDP.

Théoriquement, un seul utilisateur ne devrait faire la mise à jour que d'un seul MDP à la fois. Mais pour la robustesse et la sûreté de fonctionnement de l'application, il vaut mieux traiter ce cas.

Passons maintenant à la récupération des MDP. Voici un diagramme d'échange montrant la trace des appels de fonction lorsqu'un client veut récupérer des mises à jour auprès serveur :

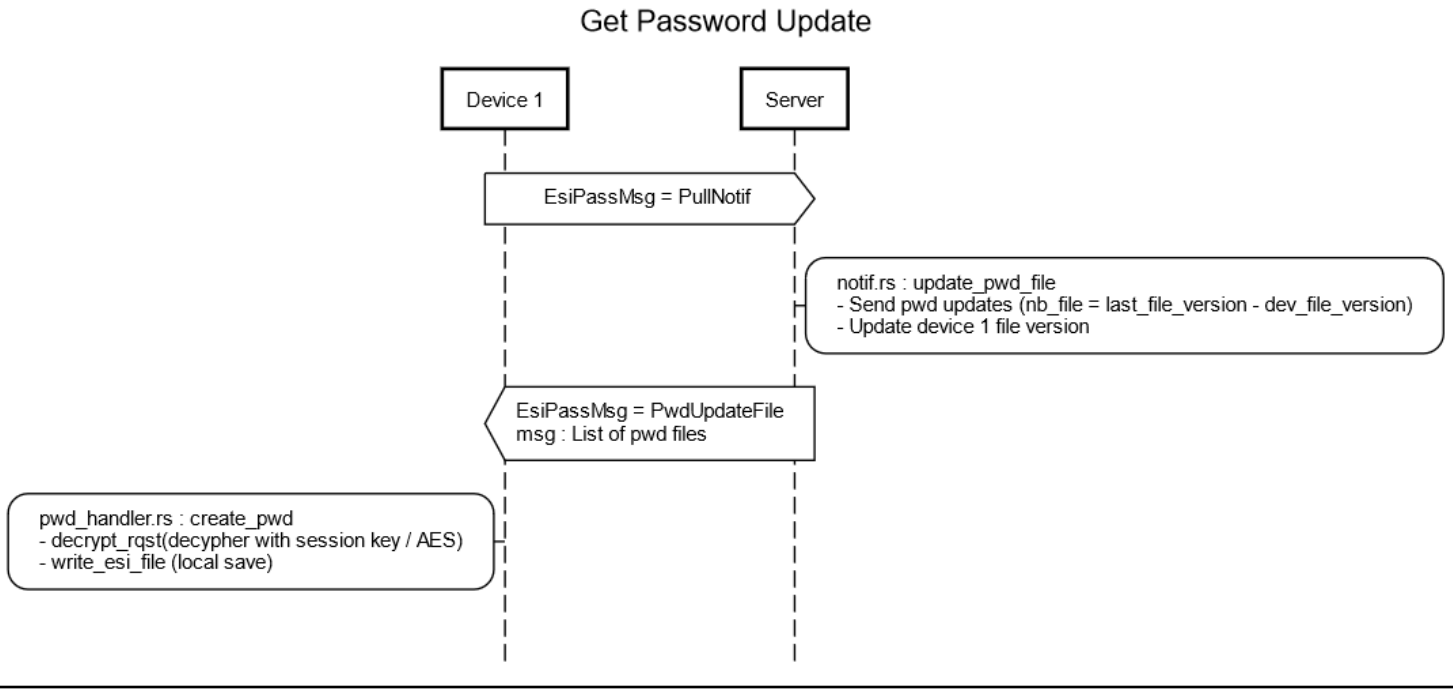

Figure 15 : récupération d'une mise à jour de MDP

Lors d'un ***pull_notif***, l'appareil reçoit uniquement les mises à jour de MDP qui lui manquent. Pour cela, le serveur garde un numéro de mise à jour de MDP pour chaque appareil, qui est incrémenté à chaque fois qu'il en récupère une nouvelle. Il garde aussi le nombre global de mises à jour de MDP. On peut ainsi, par une simple différence entre le numéro de l'appareil et le nombre global, savoir combien il en manque à l'appareil et lui envoyer uniquement celles-ci.

On vérifie aussi s'il y a des mises à jour sur le serveur qui ont été récupérées par tous les appareils du compte afin de les supprimer.

On fait également attention au cas où le serveur contiendrait déjà des mises à jour de mot passe et qu'il y a un changement de clé de session. On tag donc les mises à jour de MDP déjà présentes sur le serveur comme étant réalisées avec l'ancienne clé de session. Ensuite, lors d'un ***pull_notif***, on envoie d'abord à l'appareil les mises à jour de MDP qui lui manquent, mais uniquement parmi celles qui ont été chiffrées avec l'ancienne clé de session, suivie de l'indication qu'il y a eu un changement de

clé de session. Lors d'un prochain ***pull_notif,*** après avoir reçu la nouvelle clé de session, il recevra alors toutes les mises à jour qui lui manquent, chiffrées avec la nouvelle clé de session.

De plus, s'il manque au client des mises à jour de MDP qui ne sont plus présentes sur le serveur (nouvel appareil ou supprimé par le serveur car trop ancienne), on place dans l'inbox générale une demande de tous les MDP pour ce client. Le premier appareil à jour contactant le serveur enverra tous les MDP qui seront placées dans l'inbox perso de l'appareil désynchronisé qu'il récupérera au prochain ***pull_notif***.

Fonctionnalités restantes

Plusieurs briques de notre protocole n'ont pas pu être mises en place dans la preuve de concept par manque de temps.

Concernant le protocole de communication

Challenge : Lors de la création d'un compte ou d'un nouvel appareil, envoyer sur le numéro de téléphone renseigné par l'utilisateur un code qu'il doit rentrer dans l'application. Cela permet de s'assurer que l'utilisateur est bien le propriétaire du compte et de limiter le multi-compte ainsi que la création de bots et potentiellement les attaques DDOS. On pourrait également se servir d'une adresse mail, mais maintenant la quasi-totalité des gens ont un numéro de téléphone. Il a l'avantage d'être plus difficile à créer comparé à une adresse mail et aussi d'être plus difficilement compromis car, il faut voler le téléphone pour recevoir le code, là où une adresse mail peut être accessible de partout.

Double ratchet : Le protocole Signal dérive la clé de transfert servant au chiffrement des messages avec l'utilisation du Double Ratchet. Il permet d'assurer la forward secrecy ainsi que la future secrecy s'il est réinitialisé à chaque fois. Signal réalise un nouveau DHE pour chaque message afin de réinitialiser le Double Ratchet et d'avoir cette future secrecy. Dans notre cas, pour des raisons d'optimisation de capacité de stockage server, on partage une clé de session en utilisant un DHE pour assurer la confidentialité de l'envoi, puis on s'en sert pour initialiser un double KDF pour générer les clés de transfert de la manière suivante :

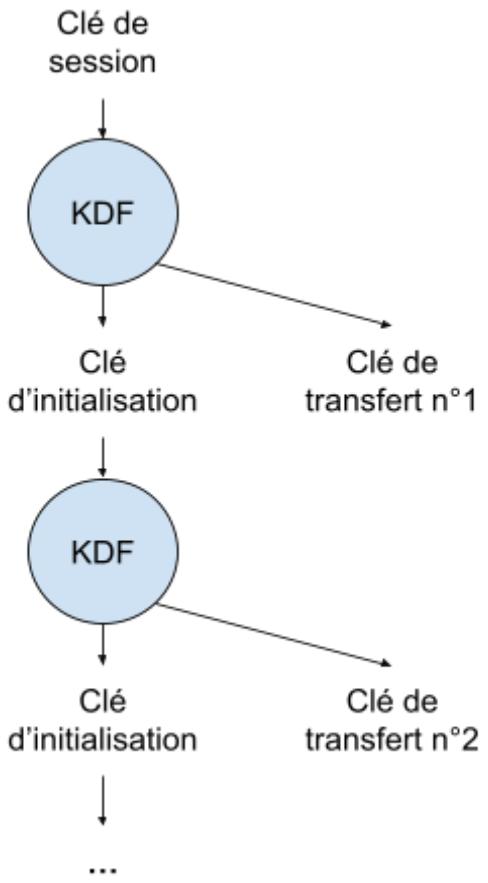

Figure 16 : double KDF

On a donc la future secrecy uniquement lorsque l'on change de clé de session et entre les MDP chiffrés avec des clés de transfert dérivées de clé de session différentes.

Renouvellement automatique de la clé de session : Paramétrer le client pour qu'il demande le changement de clé de session au bout d'un certain temps fixé et/ou d'un nombre d'utilisation. Le changement de clé de session permet d'avoir le futur secrecy entre les mises à jour chiffrées avec des clés différentes. On ne veut pas utiliser la clé de session pendant trop longtemps, car plus on l'utilise longtemps plus on laisse de temps à un potentiel attaquant de casser la clé et on ne veut pas non plus lui donner l'opportunité d'avoir beaucoup de messages chiffrés avec une même clé donc la limiter en nombre d'utilisation. Le mieux serait une combinaison des deux : réinitialiser la clé de session au bout de X temps ou au bout de Y utilisation.

Authentification des messages : Permettre au serveur d'authentifier les appareils réalisant des requêtes en créant une paire de clés asymétrique lors de la création d'un appareil qu'il utilisera pour signer ses messages. Le serveur stock la clé publique liée à l'appareil lors de sa création.

Concernant l'application en elle-même

Vérification du Master password : Lorsque l'utilisateur rentre son Master password, vérifier qu'il s'agit bien du bon afin de lui indiquer s'il se trompe et aussi de ne pas essayer de déchiffrer les fichiers de MDP avec un mauvais MDP. Pour cela, on peut, lors de la création du compte, chiffrer une chaîne de caractères connue (par exemple une suite de 0) et à chaque fois que l'utilisateur renseigne son Master password, déchiffrer cette chaîne et vérifier que cela correspond bien à l'attendu. Ce procédé est inspiré de la fonction crypt utilisée par linux pour gérer le fichier shadow. Le premier appareil créé (donc à la création du compte) chiffre cette chaîne avec le Master password renseigné à la création et elle est ensuite envoyé à chaque nouvel appareil comme étant le premier fichier de MDP.

Gestion des périphériques : Il n'est pas possible de supprimer un appareil ou bien de changer son nom.

Gestion des actions concurrentes sur un même compte : Le serveur est multi-threadé et il n'y a pas de gestion d'actions concurrentes. Il peut donc y avoir deux appareils qui ajoutent une mise à jour de MDP. Seule la modification pour laquelle l'enregistrement des mises à jour se fera en dernier sera prise en compte.
En Rust, la solution serait d'utiliser un Arc<Mutex>. Le mutex permet d'obtenir un accès unique à une adresse de la mémoire. Arc signifie atomic reference counter, grâce à cela une référence à une structure peut être partagée sur plusieurs threads tout en gardant le contrôle sur le nombre de threads ayant accès au contenu de la mémoire.

Changement théorique

Il n'y a pas de double chiffrement master password et session key sur les fichiers stockés en local. L'idée initiale était de ne pas pouvoir déchiffrer les MDP sans avoir eu accès à un appareil du client en chiffrant le fichier de MDP également avec la clé de session (en plus du MDP maître), qui n'est connue que des appareils et pas du client lui-même. Mais étant donné que les MDP sont stockés chiffrés seulement sur l'appareil du client (tout comme la clé de session) et sont transmises au serveur chiffrées avec une clé de transfert, il faut forcément accéder à un appareil du client pour obtenir le fichier de MDP chiffré avec le MDP maître et dans ce cas l'attaquant accède aussi à la clé de session.

Bibliographie

Sésame - Signal :
https://signal.org/docs/specifications/sesame/#introduction

Double Ratchet - Signal :
https://signal.org/docs/specifications/doubleratchet/

Multi Party Off The Record Messaging :
https://www.cypherpunks.ca/~iang/pubs/mpotr.pdf

Asynchronous Group Messaging with Strong Security Guarantees :
https://eprint.iacr.org/2017/666.pdf

What's Up With Group Messaging? - Computerphile :
https://www.youtube.com/watch?v=Q0_IcKrUdWg

Decentralized-Password-Manager - relex12 :
https://relex12.github.io/fr/Decentralized-Password-Manager

Chacha20 :
https://www.cryptography-primer.info/algorithms/chacha/
https://loup-vaillant.fr/tutorials/chacha20-design
https://medium.com/asecuritysite-when-bob-met-alice/96-bit-nonces-too-small-try-xchacha20-and-rust-b773bb8c9bff

Key Derivation Function :
https://cryptobook.nakov.com/mac-and-key-derivation/kdf-deriving-key-from-password

P2P (Peer to Peer) : définition simple et exemples d'utilisation :
https://www.journaldunet.fr/web-tech/dictionnaire-du-webmastering/1203399-p2p-peer-to-peer-definition-traduction-et-acteurs/

Peer-To-Peer (P2P) Networking System :
https://www.vocal.com/video/p2p-network

Peer to Peer Computing :
https://www.tutorialspoint.com/Peer-to-Peer-Computing

Peer-to-Peer Communication Across Network Address Translators :
https://pdos.csail.mit.edu/papers/p2pnat.pdf