Université de Paris

Protocoles Sécurisés

Master Mathématique MIC

2e année

PROJET

Etudiants :

Berthet Pierre-Augustin

Renard Marc

Professeur :

Chroboczek Juliusz

Décembre 2021

Table des matières

I Introduction

L'objectif du projet est d'implémenter un système de fichiers distribué. Nous allons dans ce rapport présenté nos solutions de programmation, aussi bien pour la partie obligatoire que pour les modules supplémentaires que nous avons développé. Pour rappel, ce projet est basé sur une architecture P2P hybride, avec un serveur coordinateur qui permet aux pairs de se "découvrir" et d'initier des communications pour ce protocole.
L'ensemble de ce projet a été implémenté en langage Golang.

II Partie obligatoire

II.1 Découverte des pairs

La première étape pour pouvoir accéder aux fichiers stockés sur un autre pair est déjà de voir si le pair en question est disponible. Le serveur coordinateur détient une liste de tous les pairs connectés, à 180 secondes près, ainsi que leurs adresses. Il faut donc les obtenirs.
Pour se faire, on passe sur un modèle *Client-Serveur* et on discute avec le serveur via un protocole REST basé sur HTTPS.
Pour cela, on a développé 3 fonctions :

- **func HttpRequest(method, addr string, client http.Client) ([]byte, error)** : cette fonction nous permet d'effectuer une requête REST et retourne la réponse. A noter qu'on gère ici les différentes erreurs : impossibilité de contacter l'adresse HTTPS, d'émettre la requête ou encore une réponse incorrecte.
- **func ParseREST(body []byte) [][]byte** : cette fonction parse la réponse envoyée sous la forme d'un tableau de chaînes d'octets. Il s'agit juste d'une mise en forme de la donnée obtenue, qui aura son importance dans la troisième fonction :
- **func PeerSelector(ids [][]byte, client http.Client) ([][]byte, string)** : cette dernière fonction permet à l'utilisateur, via notamment l'usage de *fmt.Scanf*, de choisir dans une liste de pairs celui qu'il souhaite contacter. Le paramètre *ids* de cette fonction est en fait issu d'un appel à *ParseREST* appliqué lui même à la sortie du premier *HttpRequest*. A l'intérieur même de cette fonction, on fait appel aux deux précédentes afin de contacter l'adresse HTTPS */peers/choix/addresses* et d'obtenir l'ensemble des adresses du pair choisi de nouveau sous la forme d'un tableau de chaînes d'octets.

Nous pouvons donc grâce à ces fonctions contacter le serveur, faire un *GET* de la liste des pairs, choisir un pair et faire un *GET* de sa liste d'adresses.

II.2 Enregistrement auprès du serveur

Dans un second temps, on souhaite s'enregistrer auprès du serveur en tant que pair. Pour cela, on suit une procédure stricte. Nous allons dans cette partie détailler notre implémentation.

II.2.1 La structure *Message*

Le projet stipule que les messages doivent avoir une structure particulière, comme stipulée ci dessous :

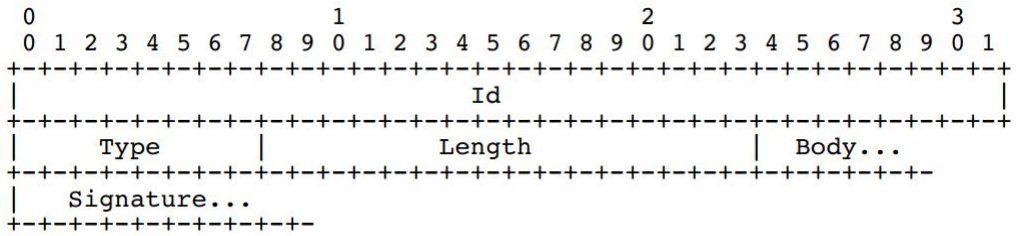

FIGURE 1 – Le format de Message tel que décrit dans le sujet

On a une structure relativement simple avec différents attributs :

- Id : un identifiant constitué de 4 octets. Celui-ci sera renouvelé à chaque nous message envoyé à notre initiative (qui ne fait pas suite à une requête d'un autre pair)
- Type : un octet indique le type de message, par exemple 0 pour *Hello* ou 254 pour *Error*
- Length : deux octets qui indiquent la taille en octet du champ Body
- Body : un champ de Length octets qui contient les données à transmettre par ce message
- Sign : champ optionnel de 64 octets qui lorsqu'il est utilisé contient la signature de Id||Type||Lenght||Body

II.2.2 Signature du message

Nous avons pris la décision de signer tous nos messages. En conséquence, nous avons adapté notre constructeur pour qu'il effectue automatiquement la signature d'un nouveau Message. Le mécanisme de signature est donc présent dans les fonctions suivantes :

— **func NewMessage(I []byte, T []byte, B []byte, privK *ecdsa.PrivateKey) Message** : il s'agit du constructeur. Il contient plusieurs fonctionnalités : contrôle des tailles de l'identité et du type, calcul automatique de la longueur de *Body*, mais surtout la possibilité de signer le message. Etant donné une clé privée ECDSA, le constructeur signe automatiquement le message et ajoute la signature dans le champ *Sign*. A noter que si la clé est le pointeur *nil*, le message n'est pas signé et le champ *Sign* vide.

— **func BytesToMessage(tab []byte, pubK *ecdsa.PublicKey) Message** : ici, on convertit une chaîne d'octets en Message. Etant donné que cette fonction est surtout utilisée pour convertir des données reçues par notre client à notre format, nous avons rajouté un contrôle de signature. Etant donnée une clé publique (celle de l'émetteur), cette fonction vérifie que la signature du message est authentique. A noter que, comme précédemment, si la clé entrée est *nil*, on considère que le message n'est pas signé. Le Message renvoyé par cette fonction est d'ailleurs la version non signée du Message reçu : une fois la signature contrôlée, elle ne nous est plus d'aucune utilité.

II.2.3 Envoi et réception d'un Message

L'envoi et la réception de messages sont gérée par deux fonctions :

— **MessageSender(conn *net.UDPConn, mess Message)**

— **MessageListener(conn *net.UDPConn, sended Message, repeat bool, pubK *ecdsa.PublicKey) Message**

La fonction **MessageSender** permet d'envoyer le message mess sur la connexion conn. Pour cela elle convertit le message en un slice d'octet puis écrit les octets de ce slice sur la connexion conn.

La fonction **MessageListener** permet elle la réception de messages.
Le paramètre sended doit être le dernier message envoyé avant de se mettre en écoute. Le paramètre repeat est un booléen qui permet, s'il est à *true*, de renvoyer plusieurs fois le dernier message envoyé. Enfin, pubK, s'il est différent de *nil* est la clef publique du pair contacté. Celle-ci sert à vérifier la signature si le pair l'implémente.

La fonction se met en écoute sur la connexion conn. Elle écoute plusieurs fois avec un temps d'attente exponentiel. Si à la fin de ces écoutes elle n'a toujours pas eu de réponse, et si le paramètre repeat est *true* alors elle renvoie le dernier message et se remet en écoute. Si après plusieurs tentatives elle n'a toujours pas eu de réponse, elle abandonne et créé une fausse réponse avec le type ERROR (254) qui sera retournée. Si au contraire elle a réceptionné un message (sous forme d'un slice d'octet) alors

elle utilise la fonction **BytesToMessage(tab []byte, pubK *ecdsa.PublicKey) Message** pour reconstruire un objet de type message correspondant au slice d'octets reçu et enfin le renvoyer.

II.2.4 Gestion et contrôle des erreurs

Nous avons deux fonctions pour détecter et traiter des éventuelles erreurs sur les messages :

- **func ErrorMessageSender(mess Message, str string, conn *net.UDPConn)** : Cette fonction génère automatiquement un Message d'erreur et l'envoie à la connexion UDP *conn* à partir d'un message *mess* erroné. Son type de retour est automatiquement mis à 254 et son champ *Body* contient *str*.
- **func TypeChecker(mess Message, typ int16) bool** : Une fonction simple qui contrôle que le type d'un message donné correspond bien à celui attendu.

Ces deux fonctions travaillent souvent en tandem, **TypeChecker** pour détecter une anomalie, **ErrorMessageSender** pour la réponse automatique qui s'ensuit.

II.2.5 Échange initial avec le serveur

On commence donc par ouvrir une connexion UDP avec le serveur. Une fois celle ci ouverte, on utilise les fonctions décrites dans la partie précédente et on lui envoie un message *Hello*. Ce message a un type à 0, et comporte au début de son champ *Body* les extensions supportées dans notre pair. Le champ réservé aux extensions fait 4 octets. Le reste de *Body* contient notre nom de pair.On se met alors en écoute du serveur.
Si ce dernier a bien reçu le message $Hello$, il va nous notifier d'un $HelloReply$ (type à 128). On reste alors en écoute du serveur, ce dernier devant nous notifier d'un $PublicKey$ (type 1). On y répond par un $PublicKeyReply$ (type 129), contenant éventuellement notre clé publique.
On se remet en écoute, le serveur nous envoyant son *Root* (type 2) juste après cet échange. On lui répond alors par un *RootReply* (type 130), contenant un *hash* de notre racine (cette dernière étant vide si l'on a rien à exporter). Une fois ces échanges terminés, nous voilà enregistrés en tant que pair !

II.2.6 Les Subroutines

Nous utilisons deux subroutines dans notre programme :

- **HelloRepeater(conn *net.UDPConn, ourPrivKey *ecdsa.PrivateKey, bobK *ecdsa.PublicKey)**
- **dataReceiver(client http.Client, privateKey *ecdsa.PrivateKey, bobK *ecdsa.PublicKey, pubK []byte)**

La subroutine **HelloRepeater** nous permet de rester enregistré auprès du serveur. Le premier paramètre conn doit être le *net.UDPConn qui a servi à s'enregistrer sur le serveur, ce sera donc un canal dédié uniquement à maintenir notre enregistrement sur le serveur. Les deux autres paramètres sont la clef qui nous sert à signer, et la clef qui nous sert à vérifier les signatures du serveur.

La subroutine **dataReceiver** est la partie avec laquelle l'utilisateur interagit. Elle répète en boucle les étapes du II.3 :

- Affichage de la liste des pairs enregistrés sur le serveur
- Choix du pair à contacter par l'utilisateur
- Tentative de connexion au pair
- Parcours de l'arbre de Merkle du pair
- Téléchargement de données

II.3 Téléchargement de données sur un autre pair

II.3.1 Connexion au pair

Les adresses du pair avec qui on veut échanger sont récupérées à l'aide du requête GET envoyée au serveur. Nous parcourons la liste des adresses jusqu'à trouver une adresse à laquelle nous arrivons à nous connecter en UDP. Si nous n'arrivons pas à établir de connexion, nous tentons un NATTraversal (détaillé dans une autre partie).

Si nous n'avons pas réussi à nous connecter en UDP avec le pair, nous abandonnons la démarche.
Dans le cas contraire, nous commençons par récupérer le hash de la racine (*root*) à l'aide d'un requête GET. À cette étape, nous nous trouvons donc virtuellement au niveau de *root* qui est un dossier.

II.3.2 Parcours de l'arbre de Merkle

Une fois connectés au pair, nous appliquons ensuite la démarche suivante tant que nous nous trouvons virtuellement dans un répertoire :

— envoi d'un message *GetDautm* en UDP. Le message en question contient le hash du répertoire dans le quel nous nous trouvons virtuellement.
— réception d'un message *Datum* en UDP. Ce message contient le nom et le hash de chaque noeud (fichier File ou FigFile, ou réoertoire Directory)
— nous proposons à l'utilisateur de choisir quel chemin suivre : soit en descendant dans un nouveau répertoire, sois en se dirigeant vers un fichier. Nous laissons aussi la possibilité à l'utilisateur de télécharger le dossier courant avec tout ce qu'il contient (ce qui fait sortir de la boucle dans laquelle nous nous trouvons).
— si l'utilisateur a choisi de se diriger dans un répertoire, alors nous récupérons le hash correspondant à ce répertoire et nous recommençons au début de la boucle

Si la boucle précédente n'a pas été interrompue par le téléchargement d'un dossier complet, alors ce qui déclenche la sortie de celle-ci est le fait que nous nous trouvions virtuellement au niveau d'un fichier, et dans ce cas nous téléchargeons ce fichier.

Tous les téléchargements qui vont suivre se feront à l'intérieur d'un dossier créé à cet effet et nommé download_from_peer où peer est le nom du pair choisi.

II.3.3 Téléchargement d'un fichier File ou BigFile

Un fichier File est composé d'un unique Chunk, il est donc accessible à l'aide d'un seul message *GetDatum* à l'aide du hash qui l'identifie dans l'arbre. La requête *GetDatum* avec le hash de ce fichier aura une réponse *Datum* dont le champ Body sera composé du hash de la requête suivi des données du fichier elles-mêmes. Nous créons alors un fichier dans le répertoire dédié de notre machine, nommé par le nom du fichier et nous y écrivons les données du message *Datum* reçu en réponse de notre message *GetDatum*.

Dans le cas d'un BigFile, composé lui de plusieurs Chunks, le champ Body du message *Datum* reçu en réponse à notre message *GetDatum* est composé du hash de la requête suivi d'une suite de hash qui désignent chacun soit un BigFile soit un File. Nous parcourons alors récursivement toutes les composantes du BigFile de départ, dans l'ordre dans lequel les hashs sont écrits et à chaque fois que nous arrivons dans un File, nous concaténons les données reçues dans le message *Datum*

correspondant, aux données déjà récupérées. Une fois tout ce sous arbre parcouru, nous créons un fichier portant le nom du BigFile de départ dans le répertoire dédié, et nous y écrivons toutes les données réceptionnées.

II.3.4 Téléchargement d'un dossier

Un dossier contient des dossiers et des fichiers. Le champ Body du message *Datum* reçu en réponse à notre message *GetDatum* est composé du hash de la requête suivi d'une suite de paires nom||hash. Nous parcourons alors récursivement toutes les composantes du dossier de départ, dans l'ordre dans lequel les hashs sont écrits. À chaque changement de strate, le chemin vers le répertoire courant virtuel est mis à jour. Ainsi les sous dossiers peuvent être créés et le dossier final téléchargé est une image fidèle de l'arborescence de l'arbre chez le pair.

II.3.5 Vérification de l'intégrité

Quelque soit le type de noeud désigné par un hash, le champ Body du message *Datum* en réponse du message *GetDatum* associé à ce hash est structuré de la manière suivante : hash(data)||data. Ainsi à chaque message *Datum* reçu, nous vérifions l'intégrité des données en calculant le hash des données data et en le comparant au hash demandé.

III Module de chiffrement des communications

Il s'agit d'une extension que nous avons développée. Elle s'articule autour de 2 algorithmes de chiffrements, pour un total de 6 fonctions. A noter que nous en avons fait un module Go, ce qui rend son utilisation et son installation faciles.

III.1 Structure de l'extension

Cette extension repose sur deux phases. Dans un premier temps, on effectue un échange Diffie-Hellman (ECDH) signé. Les signatures étant vérifiées par rapport à celles dont dispose le serveur, on résiste donc à des attaques par homme du milieu (*Man-in-the-middle attack*).
Une fois le secret partagé établi, on passe à la seconde phase. Dans celle ci, on chiffre l'entièreté d'un message avant de l'envoyer à l'aide du chiffrement AES128 en mode GCM. A la réception, on déchiffre donc selon ce même mode GCM et algorithme AES128.
Dernier détail, le secret partagé après ECDH n'est pas utilisé tel quel par AES128 GCM. On le passe d'abord par la fonction de hashage SHA256. On obtient alors un mot de 256 bits. Les 128 premiers bits servent de clé pour AES128 GCM, les 128 derniers servent pour l'authentification présente dans le GCM. Cela a pour avantage d'avoir une donnée authentifiée par chacun des deux intervenants, et dont eux seuls ont la connaissance, tout en étant spécifique à leur échange.

Pourquoi AES128 plutôt qu'AES256 ?
AES128 est plus rapide qu'AES256. Comme on veut échanger des données de manière sécurisée mais sans trop sacrifier en terme de "latence", on prend donc AES128.

Pourquoi GCM plutôt que CBC ?
Le mode GCM a un avantage majeur par rapport à CBC. CBC ne prend pas en charge l'authentification. Il faut donc le coupler avec un HMAC (au format *Encrypt-then-MAC*). Or, cela prend plus de temps qu'un GCM. Aussi, si GCM est vulnérable sur son nonce cryptographique (qui ne fait que 96 bits et présente une faiblesse s'il est réutilisé deux fois pour un même message), CBC requiert d'avoir une donnée à chiffrer qui soit d'une longueur multiple de 128 bits. Et donc de rajouter un padding, source potentielle de vulnérabilités.

Et pourquoi pas ECB + HMAC ?
Les messages de taille inférieure à 128 bits (ou 16 octets) sont rares, voir inexistants dans le cadre d'un échange de données entre deux pairs. On aura donc presque toujours au moins 2 blocs de 128 bits à chiffrer. Aussi, et c'est sans doute l'argument à retenir, dans le cadre d'un échange fait de plusieurs messages, en utilisant ECB il est

nécessaire de changer la clé A CHAQUE MESSAGE, ce qui fait donc que l'on compare pour des sécurités équivalentes la latence d'un GCM contre un ECDH+ECB avec PKCS#7(padding)+HMAC. GCM l'emporte encore une fois.

III.2 Fonctions et utilisation

Notre module comporte 6 fonctions :

- **func PubKeyToByte(pub ecdsa.PublicKey) []byte** : convertit une clé publique ECDH en chaines d'octets
- **func ByteToPubKey(b []byte) ecdsa.PublicKey** : convertit une chaîne d'octets en clé publique ECDH
- **func ECDHGen() ([]byte, ecdsa.PrivateKey)** : génère une clé privée ECDH et une chaîne d'octets qui représente une clé publique ECDH (et donc prête à constituer le *Body* d'une requête UDP)
- **func ECDHSharedGen(data []byte, privat ecdsa.PrivateKey) []byte** : A partir d'une chaîne d'octets reçue et de la clé privée, cette fonction génère le secret partagé, lui aussi en chaîne d'octets. A noter 2 points importants : la chaîne reçue est supposée être une clé publique de la courbe P256() selon notre format (X sur les 32 premiers octets, Y sur les 32 derniers), et second point le secret partagé est constitué uniquement par l'abscisse du point obtenu. En effet, ne pas prendre l'ordonnée réduit la complexité de l'attaque brute force par deux uniquement, le point (X, .) de la courbe n'ayant qu'au plus 2 ordonnées possibles du fait des propriétés des courbes elliptiques.
- **func AESEncrypt(dh []byte, data []byte) []byte** : chiffrement AES128 GCM tel que décrit à la sous section précédente. Cette fonction renvoie le message sous la forme d'une chaîne d'octets, prête à être envoyée.
- **func AESDecrypt(dh []byte, data []byte) []byte** : déchiffrement AES128 GCM. Cette fonction renvoie le message si la clé est correcte.

Point intéressant sur le déchiffrement, la fonction utilisée dedans (*gcm.Open*) à une consommation lissée. On évite ainsi des attaques par canaux auxiliaires.

L'utilisateur doit utiliser le module comme suit :

1. Il doit tout d'abord se générer un bi-clé ECDH, soit pour tous les pairs, soit un par pair, à sa convenance. (Nous recommandons 1 par pair)
2. Il envoie la clé publique du bi-clé au pair et attend la réponse (elle aussi contenant une clé publique)
3. Avec cette réponse, il génère un secret partagé à l'aide de sa clé privée.

4. Il peut dorénavant chiffrer l'entièreté de sa correspondance à l'envoi, et la déchiffrer à la réception.

IV Modules inachevés

Dans cette section, nous allons vous présenté les extensions sur lesquelles nous avons travaillé, mais qui n'ont pas pu être prêtes à temps pour le rendu du projet.

IV.1 NATTraversal

Nous avons tenté d'implémenter la traversée de NAT, à l'aide de la fonction suivante :

— **func NATTravMessage(peeraddr [][]byte, connJCH *net.UDPConn, privK *ecdsa.PrivateKey, bobK *ecdsa.PublicKey) *net.UDPConn**

Cette fonction suit le protocole suivant : elle envoie un message de type $NatTraversalRequest$ au serveur, puis attend une seconde. Elle envoie alors un $Hello$ à la première adresse du pair que l'on veut joindre et se met en écoute d'un $Hello$. Si on le reçoit, on envoie un $HelloReply$, et le NAT est traversé. Sinon, on tente l'adresse suivante, jusqu'à épuisement des adresses.
Durant nos tests, nous avons parfaitement reçu les messages du serveur de type $NATTraversal$. Malheureusement, la percée de NAT a échoué à chaque test.

IV.2 Arbre de Merkle

IV.2.1 La structure Node

Dans le cadre du projet, les données sont structurées sous la forme d'un arbre de Merkle. Nous avons donc pour cela une structure Node, contenant les champs suivants :

— content []byte : Ici sont contenues les données
— checksum []byte : le checksum SHA256 de content
— chunk bool : un booléen déterminant si le fichier est ou non un Chunk
— directory bool : un booléen déterminant si le fichier est ou non un répertoire
— root *Node : un pointeur sur le Node parent
— son []*Node : un tableau de Nodes enfants
— name []byte : le nom du Node, non vide uniquement pour le BigFile root d'un fichier et les répertoires

IV.2.2 Stratégie d'implémentation

Notre plan est, au lancement du client, de générer son arbre de Merkle par rapport à un dossier spécifique qui contiendra nos données.

Ensuite, on construit la structure de l'arbre en partant du dossier racine, puis on descend dans les sous dossiers et les sous fichiers, construisant ainsi les relations entre les différents Node récursivement. Une fois cette étape terminée, on remplit les champs checksum et cont en partant de nouveau de la racine récursivement.
On peut également précalculer la structure de l'arbre localement à partir de la taille en octets du fichier que l'on veut inclure. En effet, il suffit de décomposer le fichier en chunks, puis de calculer la décomposition en base 32 du nombre de chunks pour un remplissage optimal, comme le montre le schéma ci dessous :

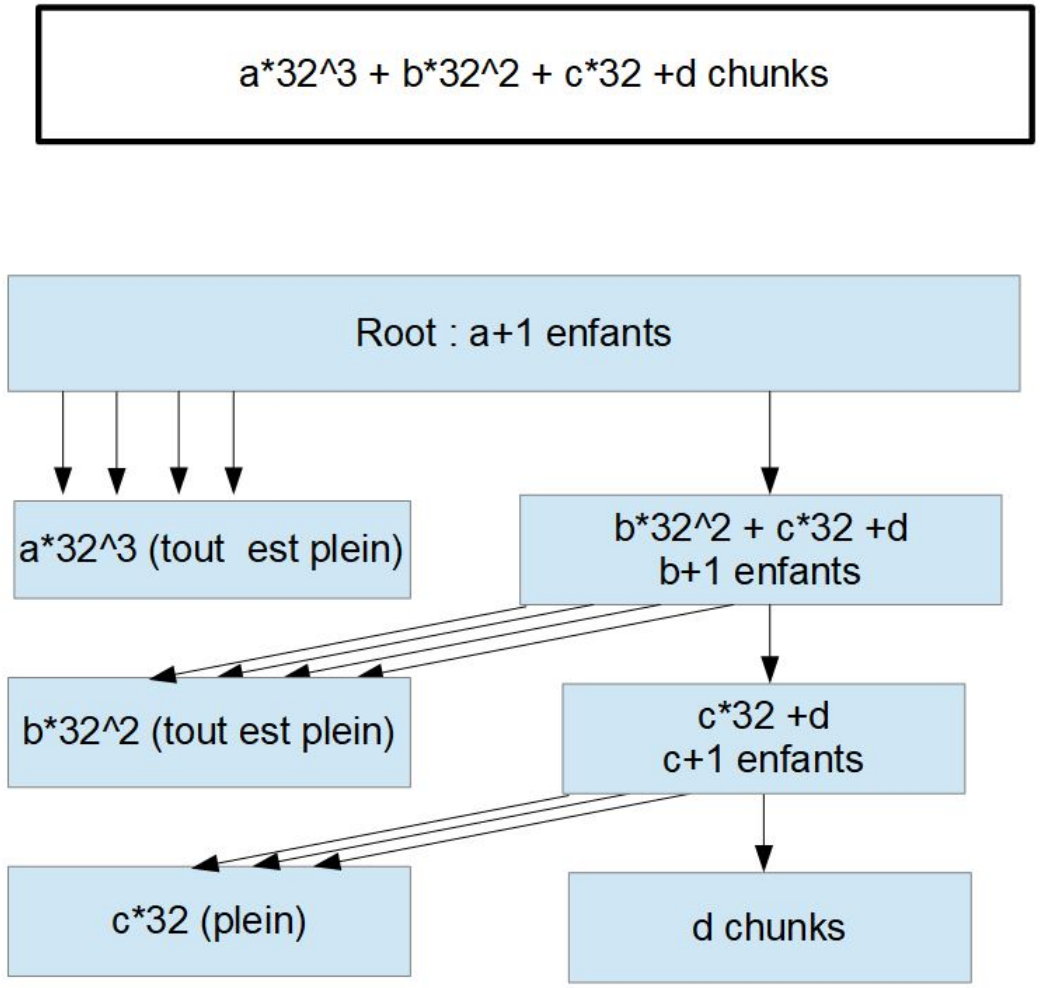

FIGURE 2 – Remplissage optimal en base 32

Dans ce schéma, chaque rectangle bleu représente un BigFile, ceux de la partie gauche sont remplis (soit de BigFile remplis, soit de 32 chunks), ceux de la partie

droite ne le sont pas. Néanmoins la structure est optimale. Ce précalcul rend considérablement plus simple la structuration en arbre de Merkle d'un fichier, car elle permet de structurer depuis la racine (éventuellement même de paralléliser) puis de calculer les hash en redescendant depuis les feuilles, ce qui garantit que les hash soient corrects.

Nous en sommes arrivés à avoir une implémentation fonctionnelle, capable de construire la structure pour un dossier contenant des sous-dossiers ainsi que de petits fichiers (on rencontre des problèmes de stack overflow sur des fichiers plus gros, probablement lié à notre implémentation du précalcul qui utilise des *int*).
Néanmoins, du fait de notre incapacité à traverser les NAT, nous n'avons pu partagé au reste des pairs les facettes intrigantes de notre humour particulier d'étudiants cryptologues.

V Conclusion

Dans le cadre de ce projet, nous avons implémenté avec succès les points suivants :

— La récupération de la liste des pairs stockées sur le serveur

— La récupération de données stockées sur le serveur

— La signature de nos messages et le contrôle de la signature des messages reçus

A cela s'ajoute le module cryptographique nous permettant de chiffrer de bout en bout la communication avec un autre pair, également fonctionnel. Nous avons également implémenté mais avec des tests infructueux la traversée de NAT. A cela s'ajoute notre implémentation des arbres de Merkle, que nous avons laissée dans un fichier séparé du projet.

En vous remerciant des votre attention.