Rapport de projet – Distributed Read-Only File System

MONA Dylan, BARKAN Ines

24 janvier 2026

Résumé

Ce projet consiste en l'implémentation d'un système de fichiers distribué en lecture seule reposant sur un protocole hybride combinant un serveur central REST et des communications pair-à-pair en UDP. Chaque pair exporte un arbre de fichiers représenté par une structure de Merkle tree, garantissant l'intégrité des données.

Notre implémentation vise à respecter le protocole spécifié tout en assurant une communication robuste entre pairs, même dans des contextes réseau contraints. Elle intègre également des mécanismes de chiffrement permettant d'assurer la confidentialité des échanges, ainsi que des extensions graphiques destinées à améliorer l'expérience utilisateur et la lisibilité des interactions avec le système.

Table des matières

1 Introduction

Ce projet a pour objectif de mettre en place un système de fichiers distribué en lecture seule.

Il repose sur un protocole hybride combinant :
— Un serveur central REST pour l'enregistrement et la découverte des pairs.
— Des communications pair-à-pair en UDP pour l'échange de données.
Chaque pair expose un arbre de fichiers représenté par un Merkle tree, permettant de vérifier l'intégrité des données sans signatures systématiques lors des transferts.

2 Architecture générale et choix techniques

2.1 Langage et environnement d'exécution

Le projet a été implémenté en **Go**, un langage adapté à la programmation réseau et aux systèmes distribués. Go offre une gestion efficace des communications UDP ainsi que des primitives de concurrence légères (goroutines et canaux), facilitant la gestion simultanée de multiples échanges pair-à-pair.

La bibliothèque standard fournit également les primitives cryptographiques nécessaires au projet, notamment ECDSA sur la courbe P-256 et SHA-256.

2.2 Organisation modulaire du projet

Le projet est organisé en plusieurs dossiers pour séparer les responsabilités.

`Client` contient la logique réseau et protocolaire (UDP, transactions, handshake, NAT, validation des messages).

`ClientStorage` gère les données et le Merkle tree, ainsi que la création et la reconstruction des fichiers et répertoires.

`Key` regroupe la génération, le chargement et la sauvegarde des clés cryptographiques.

Les fichiers téléchargés sont stockés dans `output`, tandis que `UI` contient l'interface graphique, isolée du reste du système.

2.3 Gestion des clés

Chaque pair possède une identité cryptographique stable, matérialisée par une paire de clés ECDSA basée sur la courbe elliptique `P-256` (NIST). Cette courbe est imposée par le protocole et offre un bon compromis entre sécurité et performance.

Lors du démarrage, le pair tente de charger une paire de clés existante depuis le disque à l'aide de la fonction `LoadKeyPair`. Les clés sont stockées localement dans le dossier `keys/`, sous la forme de deux fichiers encodés en PEM : `priv.pem` pour la clé privée et `pub.pem` pour la clé publique. Si aucune clé n'est trouvée, une nouvelle paire est générée via la fonction `GenerateKeyPair`, puis sauvegardée à l'aide de `SaveKeyPair`. Ce mécanisme garantit la persistance de l'identité cryptographique du pair entre plusieurs exécutions.

La clé publique est sérialisée dans un format compact de 64 octets, correspondant aux coordonnées (x, y) du point sur la courbe elliptique, chacune encodée sur 32 octets (`SerializePublicKey`). À la réception, une clé publique est reconstruite à partir de ce format binaire via la fonction `ParsePublicKey`, qui vérifie la validité du format avant utilisation.

3 Protocole client–serveur

3.1 Enregistrement et authentification auprès du serveur

L'enregistrement auprès du serveur central se fait via une requête HTTP `PUT` sur l'endpoint `/peers/<nom>/key` contenant la clé publique. Cette opération permet au serveur de connaître l'identité du pair et de prévenir les collisions de noms ou le détournement de clé. Une fois la clé enregistrée, le pair peut commencer à interagir avec le reste du réseau en utilisant les mécanismes de communication pair-à-pair.

Seules les clés publiques sont partagées via le serveur, tandis que la clé privée reste strictement locale, garantissant l'authenticité et la signature des messages envoyés. La fonction `keepAlive` maintient cette connexion avec le serveur en envoyant un message PUT toutes les 28 minutes.

3.2 Découverte et suivi des pairs

À l'initialisation, le pair récupère la liste des pairs connus depuis le serveur via l'endpoint `/peers/`. Pour chaque pair, la liste de ses adresses UDP est obtenue via `/peers/<nom>/addresses`.

Ces adresses sont stockées dans une `map` interne `Peers`, où chaque clé correspond au nom du pair et chaque valeur est une structure contenant notamment :

— la liste des adresses possibles (`Addresses`) ;
— la clé publique (`PublicKey`), qui est initialement vide et sera assignée lors de la première connexion réussie ;
— l'adresse active (`ActiveAddr`), utilisée pour tous les échanges ultérieurs ;
— l'état de connexion (`Peer.State`), qui devient PeerAssociated après réception d'un `HelloReply` suite à l'envoi d'un `Hello` et qui vaut PeerDiscovered à l'état initial.

Les adresses vides ou les pairs n'ayant pas effectué de handshake avec le serveur sont ignorés pour limiter le traitement inutile et maintenir la cohérence du réseau.

3.3 Rafraîchissement périodique de la liste des pairs

La liste de pairs est automatiquement rafraîchie toutes les 30 secondes. Pour réduire le trafic réseau, le pair utilise l'en-tête HTTP `If-None-Match`, récupérant uniquement les changements depuis la dernière requête. Lors de ce rafraîchissement, les nouvelles adresses sont assignées aux pairs existants et les pairs absents de la liste sont supprimés de la map.

4 Architecture réseau pair-à-pair

4.1 Modèle de communication UDP

Les échanges pair-à-pair du système reposent exclusivement sur le protocole UDP. Les requêtes HTTP sont uniquement utilisées pour interagir avec le serveur central, notamment pour la découverte des pairs, la récupération des clés publiques et des adresses réseau.

UDP ne garantissant ni la fiabilité, ni l'ordre, ni l'unicité des messages, ces propriétés sont entièrement prises en charge au niveau applicatif. Le protocole implémenté s'appuie ainsi sur des mécanismes explicites de gestion des transactions, de retransmission et de validation des messages afin d'assurer des échanges fiables.

Chaque pair utilise une unique socket UDP pour l'ensemble des communications entrantes et sortantes. Tous les paquets reçus sont traités de manière centralisée, tandis que l'envoi des messages est orchestré par le système de transactions, garantissant une gestion cohérente des échanges simultanés avec plusieurs pairs.

4.2 Gestion centralisée de la réception des paquets

Tous les messages entrants sont capturés par `CaptureMessage`, qui lit les paquets UDP et les redirige vers le `Routeur`. Le `Routeur` distingue les messages selon leur type :

— les requêtes (`type` ≤ 127) sont envoyées vers `requestChan`
— les réponses (`type` > 127) sont envoyées vers `responseChan`.

Les goroutines `RequestHandler` et `ResponseHandler`, dans les fichiers respectifs `requestHandler.go` et `responseHandler.go`, consomment ces canaux et appellent les fonctions spécifiques à chaque type de message.

4.3 Structure des messages et sérialisation

Les messages suivent un format binaire simple et extensible :

ID	TYP	LEN	BODY	SIG(64)
4B	1B	2B	variable	optionnel

Le corps (`BODY`) contient les informations spécifiques au type de message, par exemple : `Hello` contient le nom du pair et éventuellement une clé publique Diffie-Hellman.

La sérialisation et la désérialisation sont assurées par `BuildMessage` et `parseRecvMessage`.

Pour les NAT Traversal Requests, le `BODY` encode l'adresse IP et le port du pair cible (`ParseNATBody`).

4.4 Validation des messages et sécurité

Chaque message reçu, signé conformément aux sujets du projet, est soumis à plusieurs étapes de validation avant d'être traité. Les paquets provenant de pairs inconnus, bannis ou non encore connectés sont ignorés.

Les messages critiques du protocole sont protégés par une signature cryptographique ECDSA. Une signature garantit :

— **l'authenticité** du message, en assurant qu'il a bien été émis par le détenteur de la clé privée associée au pair ;

— **l'intégrité** du message, toute modification du contenu rendant la signature invalide.

Lors de l'envoi, le message est d'abord haché à l'aide de SHA-256, puis signé avec la clé privée ECDSA du pair (`SignMessage`). La signature est ensuite ajoutée à la fin du paquet.

À la réception, la signature est vérifiée à l'aide de la clé publique du pair émetteur (`VerifyMessage`, `VerifSign`).

Tout message dont la signature est absente ou invalide **lorsque celle-ci est requise** par le protocole est rejeté.

Le protocole vérifie également la cohérence transactionnelle des échanges : toute réponse ne correspondant à aucune transaction active est ignorée, afin d'éviter le traitement de messages obsolètes ou injectés.

Enfin, pour les messages transportant des données, notamment les réponses aux `DatumRequest`, une vérification d'intégrité est effectuée. Ce type de message n'a pas besoin de signature. Le hash reçu est comparé au hash demandé, puis le hachage des données reçues est recalculé et comparé (`VerifyDataIntegrity`). Les données ne sont acceptées que si l'ensemble de ces vérifications réussit.

5 Gestion des transactions et fiabilité

5.1 Modèle de transaction

Chaque message envoyé au-dessus d'UDP est associé à une transaction, représentant un message en vol dont on attend une réponse ou une confirmation. Les transactions sont stockées dans une table globale indexée par l'identifiant unique du message.

Chaque entrée contient notamment :

— l'identifiant du message,

— le pair concerné et son adresse,

— le type de message,

— l'instant d'envoi,

— le timeout courant,

— le nombre de retransmissions restantes,

— l'état de la transaction (en attente, à retransmettre, terminée, etc.).

Certains champs de la transaction sont optionnels et ne sont utilisés que pour des types de messages spécifiques.

Par exemple, le champ **DhPriv** (`*ecdsa.PrivateKey`) est utilisé uniquement pour un message `Hello` contenant un échange Diffie–Hellman, tandis qu'il est absent pour des messages de contrôle tels que `Ping` ou un `Hello` simple.

La création d'une transaction se fait automatiquement lors de l'envoi d'un message. Elle permet de suivre l'état du message et d'assurer sa fiabilité malgré le caractère non fiable du transport UDP.

5.2 Timeouts, retransmissions et backoff

Chaque transaction est associée à un délai d'expiration (timeout) qui définit le temps maximal d'attente d'une réponse après l'envoi d'un message. Si aucune réponse n'est reçue avant l'expiration de ce délai, la transaction est considérée comme expirée.

Lorsqu'un timeout survient, le protocole applique un mécanisme de retransmission automatique :
— le message est renvoyé au pair concerné,
— le timeout est augmenté selon une stratégie de backoff exponentiel,
— le compteur de retransmissions restantes est décrémenté.

Ce mécanisme permet de s'adapter dynamiquement aux conditions du réseau, en limitant la surcharge en cas de congestion ou de latence élevée. Si le nombre maximal de retransmissions est atteint sans réception de réponse valide, la transaction est abandonnée et le message est considéré comme perdu.

Des stratégies spécifiques peuvent être appliquées selon le type de message. Par exemple, pour les messages liés au handshake ou à la traversée de NAT, l'expiration d'une transaction peut entraîner un changement d'adresse active ou le déclenchement d'une nouvelle tentative de traversée de NAT.

L'ensemble de ces opérations est assuré par un mécanisme périodique de maintenance des transactions, garantissant la libération des ressources et la robustesse globale du protocole.

5.3 Correspondance requête–réponse

La correspondance entre une requête et sa réponse repose sur l'identifiant unique du message, présent dans l'en-tête de chaque paquet. Lorsqu'une requête est envoyée, son identifiant est enregistré dans la transaction associée.

À la réception d'un message de type réponse (type > 127), le système, via la fonction **ResolveTransaction** du fichier **transactions.go**, recherche une transaction existante portant le même identifiant :
— si une transaction correspondante est trouvée, la réponse est associée à cette transaction,
— la transaction est alors marquée comme terminée et retirée de la table,
— les traitements spécifiques au type de réponse sont exécutés (mise à jour d'état du pair, validation des données, poursuite d'un téléchargement, etc.).

Cette fonction change l'état de la transaction à TxDone. Autrement dit, la transaction est terminé et sera supprimé par la goroutine périodique `CleanupTransactionLoop`.

Les réponses ne correspondant à aucune transaction active sont ignorées. Ce comportement permet d'éviter les effets indésirables liés à des paquets obsolètes, dupliqués ou malveillants, et contribue à la sécurité et à la cohérence du protocole.

Ce mécanisme de correspondance explicite garantit qu'aucune réponse n'est traitée sans requête préalable valide, renforçant ainsi la fiabilité des échanges au-dessus d'un transport UDP non fiable.

Remarque Certaines requêtes n'ont pas besoin d'une transaction : c'est le cas du message `Ping`

6 Handshake, authentification et gestion des pairs

6.1 Procédure Hello / HelloReply

L'établissement d'une connexion logique entre deux pairs repose exclusivement sur l'échange d'un couple de messages `Hello` / `HelloReply`. Lorsqu'un pair reçoit une requête `Hello`, il répond systématiquement par un `HelloReply`. De plus, si l'émetteur n'est pas encore connu comme connecté, le pair initie également l'envoi d'un message `Hello` en retour afin de prendre connaissance de son identité et d'établir une relation symétrique.

L'envoi d'un message `Hello` peut se faire de deux manières :
— via la fonction `SendHello`, utilisée lorsque le pair possède déjà une adresse UDP active, par exemple suite à la réception d'un message `NatTraversalRequest` ou d'un `Hello` entrant ;
— via la fonction dédiée `HelloToPeer`, qui teste successivement les différentes adresses UDP connues pour un pair, notamment en cas d'échec de communication. Lorsque toutes les adresses ont été testées sans succès, une procédure de traversée de NAT est déclenchée.

Lorsqu'un `HelloReply` est reçu et validé (signature cryptographique correcte et cohérence de l'identité du pair), l'adresse source du message est enregistrée comme *adresse active* du pair. Cette adresse est ensuite utilisée pour l'ensemble des échanges ultérieurs tant que l'association reste valide.

Le handshake assure également l'authentification du pair distant, puisque les messages `Hello` et `HelloReply` sont systématiquement signés et vérifiés à l'aide des clés publiques obtenues via le serveur.

Enfin, cette procédure permet également au serveur de découverte de prendre connaissance de l'adresse UDP effective du pair (couple IP/port).

6.2 État de connexion des pairs

Chaque pair connu est représenté par une structure interne contenant son état de connexion. Un pair est considéré comme *connecté* uniquement après un handshake réussi, c'est-à-dire après réception d'un `HelloReply` valide.

L'état de connexion d'un pair inclut notamment :

— **Adresse UDP active** : l'adresse source validée lors du dernier `HelloReply` reçu, utilisée pour tous les échanges ultérieurs ;
— **Indicateur de connexion logique** : booléen indiquant si le pair est considéré comme connecté ;
— **Date du dernier message reçu** (`LastSeen`) : timestamp mis à jour à chaque réception de message du pair ;

La maintenance est assurée individuellement pour chaque pair. Lorsqu'un pair est connecté, c'est-à-dire à la réception d'un message `HelloReply`, une routine de maintenance dédiée est lancée pour ce pair.

La fonction `MaintenancePerPeer` s'exécute alors de manière périodique et assure le suivi de son état de connexion :

— un message `Ping` est envoyé toutes les 2 minutes afin de vérifier que le pair est toujours actif ;
— si aucune activité n'est observée depuis plus de 6 minutes, sur la base du champ `LastSeen`, le pair est considéré comme déconnecté et la fonction `DeconnectPeer` est appelée.

Ce mécanisme permet un suivi autonome de chaque pair et une détection fiable des déconnexions, sans dépendre d'une maintenance globale.

7 Traversée de NAT

7.1 Principe général

La traversée de NAT (Network Address Translation) est un mécanisme permettant à deux pairs situés derrière des routeurs NAT différents de communiquer directement via UDP. Les NAT modifient les adresses et ports source des paquets sortants, ce qui empêche un pair distant d'initier directement une connexion entrante.

Le protocole mis en place repose sur l'utilisation de messages UDP spécifiques (`NatTraversalRequest` et `NatTraversalRequest2`) pour informer les pairs des adresses et ports à utiliser, et exploite les ouvertures temporaires dans les NAT créées par les messages entrants.

7.2 Rôle de l'intermédiaire

Un intermédiaire peut alors relayer les messages.

Le processus est le suivant :

1. Le pair initiateur envoie un `NatTraversalRequest` vers le serveur ou un pair intermédiaire.
2. L'intermédiaire reçoit la demande et retransmet un `NatTraversalRequest2` au pair cible avec l'adresse source à utiliser.
3. Le pair cible, en recevant ce message, connaît l'adresse et le port exacts à utiliser pour communiquer avec l'initiateur.

Cette interaction permet de contourner les limitations imposées par le NAT.

7.3 Interaction avec le handshake

Lorsqu'un pair nous envoie un `Ping` alors qu'il n'est pas encore connecté à notre instance, le pair profite de ce message pour initier un handshake. La réception d'un `Ping` crée une ouverture temporaire dans le NAT du pair distant, ce qui permet l'envoi d'un message sortant qui sera correctement reçu.

Dans ce cas, la fonction `SendHello` est appelée pour envoyer un `Hello` au pair. Ce message est associé à une transaction, stockée dans la map `Transactions`, et sera retransmis jusqu'à `Retries` fois si aucune réponse n'est reçue. Chaque retransmission utilise un *backoff exponentiel* : le délai avant chaque nouvelle tentative double par rapport à la précédente, réduisant la charge réseau et adaptant la fréquence des envois à la latence observée.

Cette méthode garantit que le handshake peut réussir même derrière un NAT strict, en exploitant l'ouverture temporaire générée par le message entrant. La combinaison de la réception du `Ping`, de l'envoi d'un `Hello`, des retransmissions et du backoff exponentiel permet d'établir une connexion stable et authentifiée.

8 Gestion des données et Merkle Tree

8.1 Structure du Merkle Tree

Le système de stockage utilise un *Merkle Tree* pour représenter les fichiers et répertoires de manière hiérarchique et sécurisée. Chaque nœud du Merkle Tree est représenté sous la forme d'un tableau d'octets dont le premier octet encode le type du nœud (`Chunk`, `Directory`, `Big` ou `BigDirectory`). Ce type permet au système d'interpréter correctement la structure du nœud lors de la reconstruction des données. Comme le type fait partie des données prises en compte dans le calcul du hash SHA-256, deux nœuds de types ou de structures différentes produisent nécessairement des hash distincts, évitant ainsi toute ambiguïté lors de l'identification des nœuds. Les nœuds sont stockés dans une map globale `MerkleMap`, indexée par le hash hexadécimal du nœud.

Les types de nœuds implémentés sont :

— **Chunk :** feuille contenant un morceau de données de taille fixe (1024 octets). Chaque chunk est identifié par son hash.
— **Directory :** représente un répertoire contenant jusqu'à `MaxDirEntries` entrées. Chaque entrée associe le nom du fichier ou sous-répertoire au hash de son nœud enfant.
— **Big :** nœud intermédiaire regroupant plusieurs chunks ou nœuds intermédiaires pour gérer de grands fichiers.
— **BigDirectory :** nœud intermédiaire regroupant plusieurs répertoires, utilisé lorsque le nombre d'entrées d'un répertoire dépasse la limite maximale.

La fonction `BuildMerkleNode(path string)` construit récursivement le Merkle Tree pour un fichier ou un répertoire donné. Pour les fichiers, elle découpe le contenu en chunks et crée des nœuds `Big` si nécessaire. Pour les répertoires, elle construit des nœuds `Directory` ou `BigDirectory` selon le nombre d'entrées.

8.2 Calcul et vérification des hash

Chaque nœud est associé à un hash calculé avec SHA-256. Les fonctions principales sont :

— `Sha([]byte) []byte` : calcule le hash SHA-256 d'un tableau d'octets.
— `HashChunk(data []byte) []byte` : hash d'un chunk de données.
— `HashDirectory(entries []DirectoryEntry) []byte` : hash d'un répertoire.
— `HashBig(hashes [][]byte) []byte` et `HashBigDirectory(hashes [][]byte) []byte` : combinent plusieurs hashes pour former un nœud intermédiaire.

La vérification de l'intégrité du Merkle Tree se fait via la fonction `VerifyMerkle (rootHash []byte)`, qui parcourt récursivement tous les nœuds à partir de la racine et vérifie que chaque nœud est présent et correct.

8.3 Fenêtre glissante pour les transferts de données

Pour améliorer l'efficacité et la fiabilité des transferts de données entre pairs, le pair utilise un mécanisme de *fenêtre glissante* (`SlidingWindow`). Avant l'implémentation de ce mécanisme, toutes

les requêtes `DatumRequest` étaient envoyées simultanément, sans contrôle de congestion, ce qui provoquait de nombreuses pertes de paquets et rallongeait fortement le temps de téléchargement, dépassant 4 minutes pour des fichiers de taille modérée.

Il est important de noter que la fenêtre glissante est utilisée uniquement pour les `DatumRequest`, car ce sont les requêtes les plus lourdes et les plus nombreuses. Les autres types de messages, comme `Hello`, `Ping` ou `NatTraversalRequest`, restent envoyés de manière classique, car leur poids et fréquence sont faibles et n'entraînent pas de surcharge réseau.

Le principe de la fenêtre glissante est d'autoriser un nombre limité de `DatumRequest` en vol simultanément, ajusté dynamiquement en fonction des réponses reçues et des succès précédents. Cela réduit la surcharge réseau, limite les pertes et améliore le temps de transfert.

Fonctions principales

- `CanSend()` : appelée avant l'envoi d'une requête `DatumRequest`, elle vérifie si le nombre de requêtes en vol (`InFlight`) est inférieur à la taille actuelle de la fenêtre (`Size`). Si oui, la requête peut être envoyée.
- `OnSend()` : appelée dès qu'une requête est envoyée, elle incrémente `InFlight`.
- `OnSuccess()` : appelée lorsqu'une réponse est reçue correctement, elle décrémente `InFlight` et augmente progressivement la taille de la fenêtre (AIMD) jusqu'à un maximum `Max`.
- `OnTimeout()` : appelée lorsqu'une requête expire ou que les retransmissions sont épuisées. Elle décrémente `InFlight` et réduit la taille de la fenêtre de moitié (backoff exponentiel), jusqu'au minimum `Min`, afin de limiter les pertes supplémentaires.

Ordonnancement des requêtes Le `DatumScheduler` est une boucle qui parcourt la `DatumQueue`, un canal contenant les jobs de téléchargement de données. Pour chaque job :

1. Le pair correspondant à l'adresse cible est récupéré.
2. Le scheduler attend qu'une place soit disponible dans la fenêtre du pair en appelant `CanSend()`. La vérification se fait toutes les 500 microsecondes pour ne pas bloquer le scheduler.
3. Une fois la place disponible, l'identifiant de la requête est généré (`GenerateId()`), le message `DatumRequest` est construit (`BuildDatumRequest`) et envoyé via `SendMessage`.
4. La transaction correspondante est créée (`CreateTransaction`) pour assurer la retransmission en cas de perte.
5. `OnSend()` est appelée pour mettre à jour l'état de la fenêtre.

Impact sur le temps de transfert Avec la fenêtre glissante, les `DatumRequest` sont limitées et contrôlées en fonction de la congestion réseau et des réponses reçues. La combinaison du backoff exponentiel et de l'AIMD permet une adaptation dynamique à chaque pair. Le temps de téléchargement passe ainsi de plus de 4 minutes à seulement 2-4 secondes.

8.4 Téléchargement et reconstitution des fichiers

Pour reconstruire un fichier ou un répertoire à partir du Merkle Tree, on utilise la fonction `RebuildNode(hash []byte, path string)` :

- **Chunk :** les données sont écrites directement dans le fichier cible.
- **Directory :** crée les sous-répertoires correspondants et appelle récursivement `RebuildNode` pour chacun de ses enfants.
- **Big :** reconstruit les fichiers volumineux en écrivant successivement les chunks enfants dans le fichier, en appelant récursivement `WriteBigToFile` pour gérer les nœuds intermédiaires.
- **BigDirectory :** parcourt récursivement les répertoires enfants en invoquant `RebuildNode`.

Cette reconstruction garantit que l'arborescence du système de fichiers ainsi que le contenu des fichiers sont fidèlement restaurés à partir des hashes du Merkle Tree.

Lorsqu'un document est téléchargé depuis un *pair*, celui-ci est automatiquement créé dans un dossier portant le nom du pair concerné. Cette organisation permet d'identifier clairement l'origine de chaque fichier téléchargé et facilite la gestion des données provenant de plusieurs sources.

Remarque : l'ensemble des dossiers correspondant aux pairs se trouve dans le répertoire `OUTPUT`. Ce dossier contient déjà plusieurs exemples de répertoires de pairs ayant été téléchargés précédemment, illustrant ainsi le fonctionnement du mécanisme de téléchargement et de reconstitution.

9 Chiffrement et échanges sécurisés

9.1 Motivations et modèle de menace

Le réseau pair-à-pair repose sur des échanges directs entre nœuds via un canal de communication non sécurisé, en l'occurrence UDP. Un attaquant passif peut ainsi observer le trafic réseau et tenter d'en déduire le contenu. Le chiffrement a donc pour objectif principal d'assurer la confidentialité des données échangées entre pairs.

Bien que le modèle de menace se limite principalement à l'écoute passive, le protocole intègre des mécanismes limitant certaines attaques actives. La modification ou l'injection de messages est rendue inefficace par la vérification des signatures et des hash cryptographiques, l'usurpation d'identité est empêchée par la signature des messages `Hello`, et les attaques par rejeu sont limitées grâce à l'utilisation d'identifiants de transaction uniques.

9.2 Échange de clés Diffie–Hellman

Avant de pouvoir chiffrer les données échangées, les pairs doivent établir une clé de chiffrement partagée. Cette clé est dérivée lors de la phase de *handshake* à l'aide d'un échange de clés Diffie–Hellman éphémère. Chaque pair génère une clé privée temporaire et n'échange que les paramètres publics nécessaires à la dérivation du secret partagé.

Le secret obtenu n'est jamais transmis sur le réseau et sert exclusivement à la génération d'une clé symétrique de session. Cette clé, propre à chaque connexion, est renouvelée à chaque nouvel échange, ce qui limite l'impact d'une éventuelle compromission. Elle est conservée localement par les deux pairs pour toute la durée de la session.

Afin de prévenir toute attaque de type *man-in-the-middle*, chaque message Hello est signé avec la clé privée ECDSA du pair. La vérification de cette signature par le destinataire permet de garantir que les paramètres Diffie–Hellman proviennent bien du pair attendu et n'ont pas été altérés par un tiers.

Ce mécanisme combiné permet d'assurer la propriété de *forward secrecy* : même si une clé de session venait à être compromise ultérieurement, aucune information ne pourrait être exploitée pour déchiffrer des communications passées.

9.3 Chiffrement des données

Les données échangées entre pairs sont chiffrées à l'aide de l'algorithme AES en mode GCM. Ce chiffrement symétrique est utilisé pour protéger le contenu des messages applicatifs, notamment les données du Merkle tree et les blocs de fichiers transférés.

Le chiffrement est appliqué au corps des messages ainsi qu'aux hachages associés, tandis que les informations nécessaires au routage demeurent en clair. Ce choix vise à prévenir certaines attaques par dictionnaire : si le hachage restait visible, un pair disposant déjà du hash pourrait tenter de deviner le contenu du message en comparant les valeurs observées. Dans ce cas, la confidentialité des échanges ne serait plus garantie.

Chaque message est chiffré à l'aide de la clé de session établie lors de l'échange Diffie–Hellman, garantissant que seuls les pairs participants peuvent accéder aux données transmises. Grâce à l'utilisation de clés de session éphémères, les communications passées demeurent confidentielles même en cas de compromission ultérieure d'une clé, assurant ainsi une confidentialité durable tout en conservant un coût computationnel compatible avec un environnement pair-à-pair.

10 Interface graphique

L'interface graphique a été conçue principalement comme un outil de démonstration et de validation du fonctionnement du protocole pair-à-pair. Les choix d'ergonomie et de découpage des actions ont donc été orientés vers la visibilité et la compréhension des mécanismes internes (journalisation détaillée, séparation explicite des requêtes). Dans une version destinée à un usage réel, certains logs seraient supprimés ou simplifiés, et les actions `ASK ROOT` et `ASK MERKLE` seraient fusionnées en une unique opération de **téléchargement**. Cette action déclencherait automatiquement, de manière transparente pour l'utilisateur, la récupération du Merkle Tree si nécessaire avant le transfert

effectif des données.

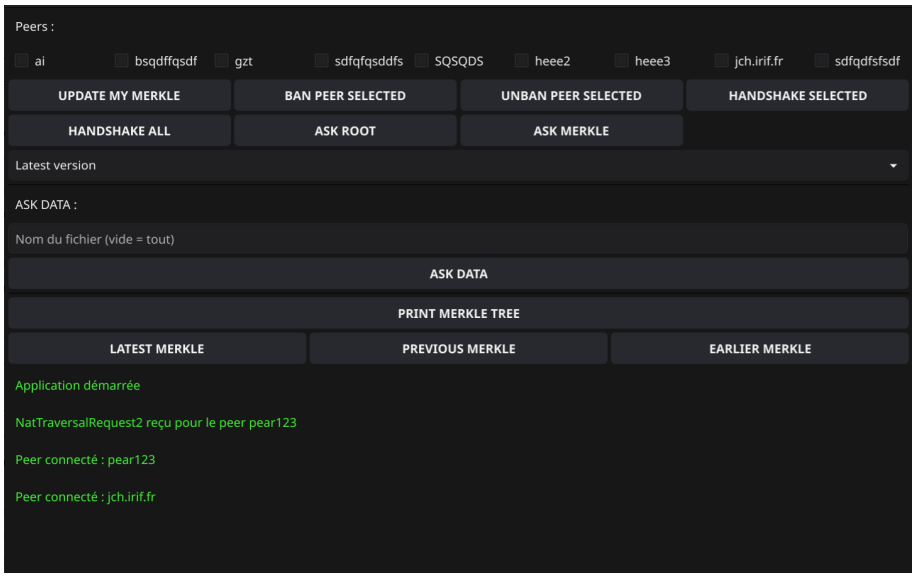

FIGURE 1 – Aperçu de l'Interface Graphique

10.1 Architecture de l'interface

L'interface graphique est développée avec le framework `Fyne` en Go et est initialisée via la fonction `StartGUI(conn, priv)` dans `gui.go` du dossier `Ui`. Elle est organisée autour de trois composants principaux :

- **Liste de pairs** : implémentée avec un `CheckGroup`, elle permet de sélectionner un ou plusieurs pairs. La liste est automatiquement rafraîchie toutes les 30 secondes par la fonction `autoRefreshPeers()` qui appelle `GetPeerListIfChanged()` et met à jour l'affichage.
- **Boutons d'action** : regroupés en zones logiques, ces boutons permettent d'effectuer des actions réseau ou locales sur les pairs sélectionnés.
- **Journal d'activité** (`Logger`) : composant `RichText` affichant les événements en temps réel avec des couleurs différenciées selon le type (`Info`, `Warn`, `Error`). Les logs sont mis à jour de manière asynchrone via la méthode `Logger.append()`.

10.2 Mode d'utilisation

L'interface est conçue pour être intuitive et permet de gérer les pairs, le Merkle Tree et les transferts de fichiers. Voici comment utiliser chaque fonctionnalité :

1. **Démarrage** : lancer l'application comme indiqué dans le README.md. La fenêtre principale affiche la liste des pairs, les boutons d'action, et un journal pour suivre les événements.
2. **Sélection de pairs** : choisir un ou plusieurs pairs dans le `CheckGroup`. Pour certaines opérations (comme télécharger un fichier spécifique), un seul pair doit être sélectionné.
3. **Handshake** :
   - `Handshake Selected` : effectue un handshake avec les pairs sélectionnés via `HelloToPeer()`.
   - `Handshake All` : effectue un handshake avec tous les pairs non connectés.
4. **Mise à jour du Merkle local** : `UPDATE MY MERKLE` reconstruit le Merkle Tree du répertoire local (`DATA_DIRECTORY`) avec `UpdateMyMerkle()`, supprime l'ancien arbre si nécessaire et met à jour `RootHash`.
5. **Restauration des versions du Merkle local** :
   - `LATEST MERKLE` : restaure la **dernière version** disponible du Merkle Tree.
   - `PREVIOUS MERKLE` : restaure la **version précédente** immédiatement avant la dernière.
   - `EARLIER MERKLE` : restaure la **version avant-dernière**, c'est-à-dire celle d'avant la version précédente.
6. **Gestion des pairs bannis** :
   - `BAN PEER SELECTED` : ajoute les pairs sélectionnés à la liste des bannis.

— `UNBAN PEER SELECTED` : retire les pairs sélectionnés de la liste des bannis.

7. **Requête ROOT et MERKLE** :
   — `ASK ROOT` : envoie une requête ROOT (`AskRootSelectedPeers()`) pour obtenir le hash racine du Merkle Tree d'un pair.
   — `ASK MERKLE` : envoie une requête MERKLE (`AskMerkleSelectedPeers()`) pour télécharger l'arbre complet d'un pair. Peut prendre du temps selon la taille du Merkle.

   Pour ces deux commandes, on peut sélectionner plusieurs pairs et ainsi envoyer les requêtes simultananément.
8. **Téléchargement de données** :
   — Saisir le nom du fichier dans le champ `Nom du fichier`.
   — Sélectionner la version à télécharger via le menu déroulant `LATEST_VERSION` par défaut, mais on peut choisir `PREVIOUS_VERSION` ou `SECOND_LAST_VERSION`).
   — Cliquer sur `ASK DATA` : appelle `AskDataPeer()` qui télécharge soit le fichier indiqué, soit tout le Merkle Tree du pair **si aucun fichier n'est spécifié**. Le téléchargement est effectué via `DownloadFileGUI()` et reconstruit les fichiers sur le disque.
9. **Affichage du Merkle Tree d'un pair** : `PRINT MERKLE TREE` affiche l'arbre du pair sélectionné dans le journal via `PrintPeerMerkle()` et `PrintTreeGUI()`.
10. **Journal** : toutes les actions et événements réseau apparaissent dans le `Logger` en temps réel, avec des codes couleur pour distinguer les informations (`Info`), avertissements (`Warn`) et erreurs (`Error`).

Résumé du comportement par défaut :
— Version des données : `la plus récente ( LATEST_VERSION )` est utilisée par défaut.
— Si aucun fichier n'est précisé pour `ASK DATA`, tout le Merkle Tree du pair est téléchargé.
— Un handshake préalable est nécessaire avant ROOT ou MERKLE pour assurer la connexion et l'échange sécurisé.

11 Autres Fonctionnalités

11.1 Accès aux fichiers restreint

Bien que les messages échangés entre pairs soient chiffrés, le chiffrement ne garantit la confidentialité des données que durant leur transport. Il n'empêche pas, à lui seul, qu'un pair non autorisé puisse accéder aux contenus une fois ceux-ci reçus. À l'échelle de la logique métier, il est donc nécessaire de contrôler explicitement quels pairs sont autorisés à interagir avec nos données et à en demander le contenu.

Dans notre protocole, la décision d'autoriser ou non un pair à interagir avec les fichiers partagés relève explicitement de l'utilisateur.

Le pair met à disposition une fonctionnalité de bannissement manuel permettant à l'utilisateur de contrôler l'accès de chaque pair à ses données. Cette gestion repose sur une liste de pairs bannis, maintenue localement par le pair. L'utilisateur peut, à tout moment, ajouter ou retirer un pair de cette liste à l'aide des actions *Ban* et *Unban* de l'interface graphique.

Lorsqu'un pair est présent dans la liste de bannissement, ses messages sont ignorés et il ne peut plus initier de nouvelles requêtes, notamment les demandes de données ou de Merkle Tree. Ce mécanisme est entièrement réversible : un pair précédemment banni peut être réautorisé instantanément sans nécessiter de redémarrage ou de reconfiguration du protocole.

11.2 Gestion des versions des fichiers des pairs

Chaque pair conserve un historique limité de ses fichiers via un root Merkle actuel (`Peer.Root`) et les trois derniers roots reçus (`Peer.Listroots`). Lorsqu'un nouveau root est reçu, la fonction `AddRootToPeer(pair, hash)` ne l'ajoute que s'il est différent du root actuel. Si nécessaire, `addListRoot` supprime l'ancien root pour ne conserver que les trois dernières versions. Un événement `OnPeerEvent` notifie l'interface utilisateur en cas de changement.

Pour le téléchargement, l'utilisateur peut sélectionner une version spécifique via `selectVersionRoot (pair, version)` : la dernière version (`Root`), la version précédente ou l'avant-dernière. La fonction `AskDataPeer` utilise ce root pour reconstruire les fichiers avec `DownloadFileGUI`.

11.3 Notification de changements de données des pairs

Une fonctionnalité de notre système est de prévenir l'utilisateur lorsqu'un *Merkle tree* d'un pair a changé. La fonction `CheckRoots` envoie toutes les 3 minutes une requête (`RootRequest`) aux pairs connectés, à l'exception des pairs bannis, pour vérifier si leur *Merkle tree* a été modifié. Si un changement est détecté, un message est affiché dans l'interface pour informer l'utilisateur.

Considérons r le nombre de pairs concernés. Chaque aller-retour de `RootRequest` consomme environ 32 octets (en négligeant l'en-tête, la requête a un body vide et la réponse 32 octets).

— Consommation par heure :

$$\begin{aligned} \text{consommation\_horaire} &= 32 \times \frac{60}{3} \times r \\ &= 640 \times r \quad \text{octets par heure} \end{aligned}$$

— Consommation par jour :

$$\begin{aligned} \text{consommation\_journalière} &= 640 \times r \times 24 \\ &= 15360 \times r \quad \text{octets par jour} \\ &\approx 15 \text{ kB par pair et par jour} \end{aligned}$$

Il est important de noter que l'application n'est pas forcément utilisée 24 heures sur 24, ce qui réduit la consommation réelle.

Lorsqu'un utilisateur effectue un *Ask Merkle*, si aucun changement du *Merkle tree* n'est détecté pour le pair concerné, les données déjà stockées localement peuvent être utilisées pour le téléchargement. Cela fonctionne comme un système de cache, évitant ainsi de redemander inutilement toutes les données au pair et réduisant la consommation réseau et le temps de téléchargement.

Toutefois, l'interrogation systématique des pairs toutes les 3 minutes devient déraisonnable pour un grand nombre de pairs, en raison de la charge réseau et CPU. Pour pallier cela, on pourrait augmenter l'intervalle de polling, vérifier les pairs par rotation, ou mettre en place un mécanisme où les pairs notifient les autres pairs uniquement en cas de changement de leur *Merkle Tree*.

12 Résumé des fonctionnalités

Fonctionnalités minimales	**Fonctionnalités étendues**
Enregistrement auprès du serveur et maintien de l'association sur une durée non bornée (keep-alive)	Support des **Big Directories** (répertoires de taille arbitraire)
Mise à disposition de fichiers pour d'autres pairs, avec ou sans NAT	Liste des pairs mise à jour en temps réel avec gestion du cas *if none match*
Mise à disposition de répertoires contenant moins de 16 entrées	Gestion de plusieurs versions des données d'un pair
Téléchargement de fichiers depuis un pair non derrière un NAT	Notification automatique lors d'un changement du Merkle Tree d'un pair
	Développement d'une interface graphique complète (Fyne)
	Possibilité de télécharger un fichier unique ou l'ensemble des données
	Chiffrement des données échangées
	Échange de clés via Diffie–Hellman
	Fenêtre coulissante pour le contrôle du flux et des requêtes en vol
	Handshake et requêtes simultanées vers plusieurs pairs (ROOT, MERKLE, données)
	Gestion des pairs bannis (ban / unban)
	Optimisation du téléchargement des données (cache, réutilisation des blocs déjà présents)
	Système de version pour nos données (EARLIER MERKLE, LASTEST MERKLE ...ect)

Table 1 – Fonctionnalités minimales requises et extensions implémentées

13 Limites et améliorations possibles

13.1 Gestion de la concurrence

La gestion de la concurrence sur la liste des pairs constitue une limite importante de notre implémentation. Les mutex n'ont pas été utilisés de manière optimale : certaines opérations réseau ont été exécutées à l'intérieur des sections critiques, et faute de temps, nous avons retiré les mutex, ce qui est une mauvaise pratique mais a permis de faire fonctionner le système.

13.2 Gestion des structures de données

Les structures de données actuellement utilisées sont relativement volumineuses et pourraient être simplifiées par une meilleure factorisation.

De plus, la structure `pair` semble perfectible. Le choix de stocker certaines informations spécifiques à chaque pair contactant le système mérite d'être reconsidéré.

Concernant la map des pairs bannis, nous estimons que cette approche présente des limites. En effet, lorsqu'un pair est supprimé de la liste des pairs connus, il ne peut pas l'être de la liste des pairs bannis et la taille de la map peut devenir conséquente à grande échelle.

13.3 Sécurité et confidentialité des échanges

La confidentialité des échanges n'est pas totalement garantie dans l'état actuel du système. En effet, seuls les hachages et les valeurs échangées sont chiffrés. Un attaquant pourrait ainsi exploiter la taille des messages pour inférer la nature des données échangées, par exemple distinguer un

répertoire d'un fichier. De plus, la corrélation possible entre une requête et sa réponse peut révéler le type d'information demandée, sans toutefois en exposer le contenu exact.

Ce risque a été volontairement accepté dans notre conception. En effet, bien qu'un attaquant puisse tirer certaines informations indirectes, il lui est impossible d'accéder au contenu réel des données échangées. Par ailleurs, toute tentative de modification des données ou des hachages est systématiquement détectée, entraînant le rejet du paquet concerné. Ainsi, aucune altération silencieuse des données n'est possible.

Dans le cadre des `datumRequest`, nous avons considéré que l'intégrité et la confidentialité des données étaient prioritaires par rapport à l'authenticité stricte de l'émetteur. L'objectif principal étant de recevoir les données correspondant à la requête initiale, ce compromis apparaît acceptable dans le contexte de notre système.

13.4 Fonctionnalités non opérationnelles

Le bouton `Handshake All` ne fonctionne pas complètement : les messages `Hello` et les requêtes NAT sont envoyés, mais seuls un ou deux pairs apparaissent connectés.

14 Conclusion

Ce projet a permis de satisfaire l'ensemble des attendus fonctionnels du protocole, en mettant en œuvre un système de partage de fichiers pair-à-pair reposant sur des *Merkle Trees* garantissant l'intégrité des données. La communication entre pairs a été rendue robuste grâce aux mécanismes de handshake, de traversée de NAT et de gestion des requêtes.

Au-delà de ces exigences minimales, plusieurs optimisations ont été intégrées, notamment une fenêtre glissante pour limiter les `DatumRequest` simultanées, l'ajout de mécanismes de confidentialité pour sécuriser les échanges, ainsi qu'une interface graphique offrant des options avancées à l'utilisateur.

Néanmoins, l'implémentation présente certaines limites. La séparation entre la logique métier et l'interface graphique pourrait être améliorée, et certaines structures de données, en particulier celles liées à la gestion des pairs, gagneraient à être simplifiées et mieux factorisées, certains champs n'étant plus strictement nécessaires. Enfin, aucun code n'a été copié ; nous nous sommes inspirés de diverses références [1] et l'IA a été utilisée uniquement pour améliorer l'affichage du débogage ou commenter le code sous notre supervision.

1. Voir section 14

Annexe

L'interface graphique de l'application permet de visualiser en temps réel l'activité des pairs et des échanges de données. Elle affiche des messages pour prévenir l'utilisateur des événements importants et du déroulement du protocole.

Parmi ces messages, on peut observer :

— La réception d'un `RootReply`, indiquant le téléchargement d'une version du Merkle Tree depuis un pair.
— La réception d'un `NatTraversalRequest2`, signalant qu'une traversée de NAT est en cours ou qu'un pair tente de se connecter via un intermédiaire.
— Les modifications des données d'un pair distant : l'utilisateur est notifié lorsque les fichiers ou le Merkle Tree d'un pair ont changé.
— Les erreurs ou échecs de communication : par exemple, l'échec d'un `Hello` ou d'une traversée de NAT, ou encore des messages d'erreur lors du téléchargement de données.

Les figures présentées dans cette annexe donnent un aperçu de l'interface graphique et des messages affichés, illustrant le suivi des opérations de téléchargement, de handshake, et de traversée de NAT.

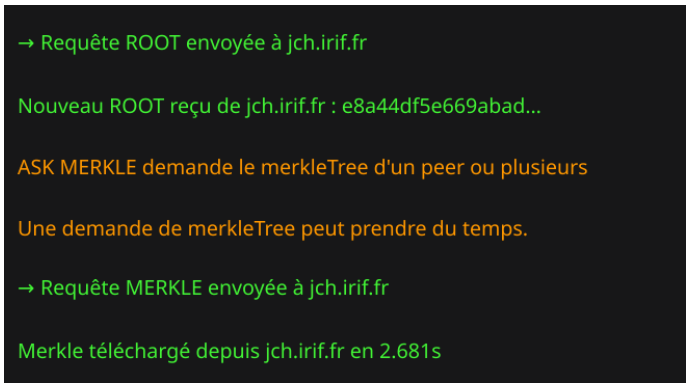

FIGURE 2 – Téléchargement des données du pair "jch.irif.fr"

FIGURE 3 – Téléchargement des données du pair "PF4"

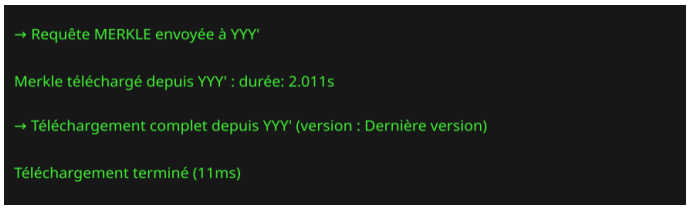

FIGURE 4 – Téléchargement des données du pair "YYY'"

Si le `RootHash` n'a pas changé, il est inutile de redemander les données au pair, car elles sont déjà présentes dans la `MerkleMap`. On peut le voir en observant la durée de téléchargement.

Il est possible que, par moment, la tentative d'envoi d'un `Hello` expire (timeout) alors que le message est finalement reçu peu après. Dans ce cas, l'application traite le message dès sa réception et met à jour l'état du pair.

On peut recevoir un NatTraversalRequest2 d'un pair à tout moment.

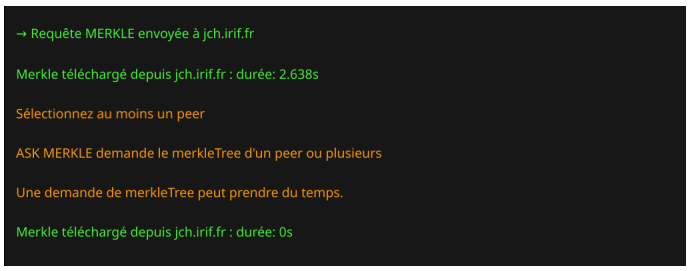

FIGURE 5 – Demande de datum pour le pair jch.irif.fr à deux reprises

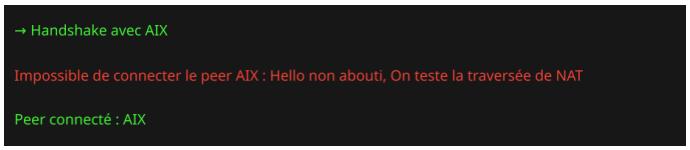

FIGURE 6 – Connexion par traversée de NAT avec "AIX"

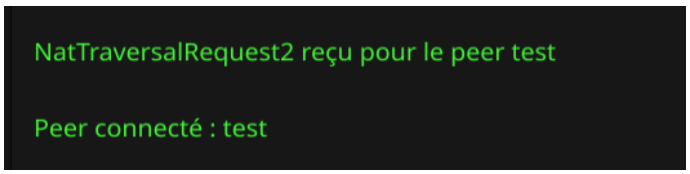

FIGURE 7 – Exemple de log pour un NatTraversalRequest2 reçu

Lorsqu'un pair auquel nous sommes connectés modifie son arborescence, nous en sommes automatiquement informés. La figure 10 illustre un exemple de notification pour le pair nommé `goroutines`.

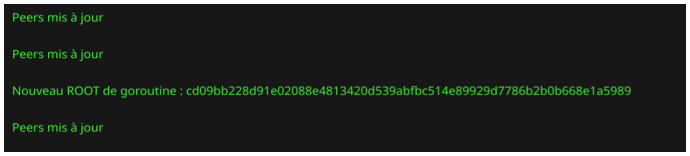

FIGURE 8 – Notification d'un changement de l'arborescence

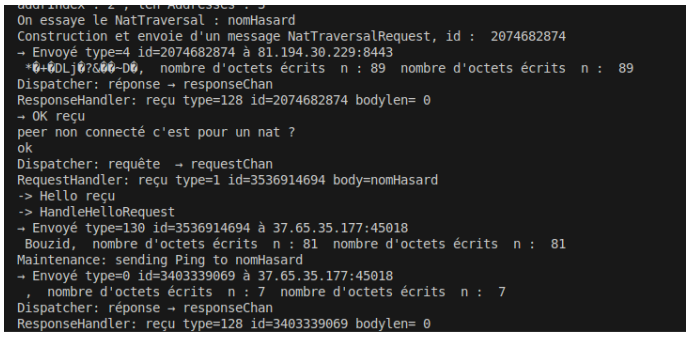

FIGURE 9 – HandShake après une traversée de NAT avec "nomHasard"

Sur cette image, nous illustrons une tentative de connexion à un pair. Lorsqu'un pair est contacté, l'application teste l'ensemble de ses adresses disponibles et marque comme active celle qui répond en premier. Cela permet de sélectionner automatiquement l'adresse la plus réactive pour établir la communication.

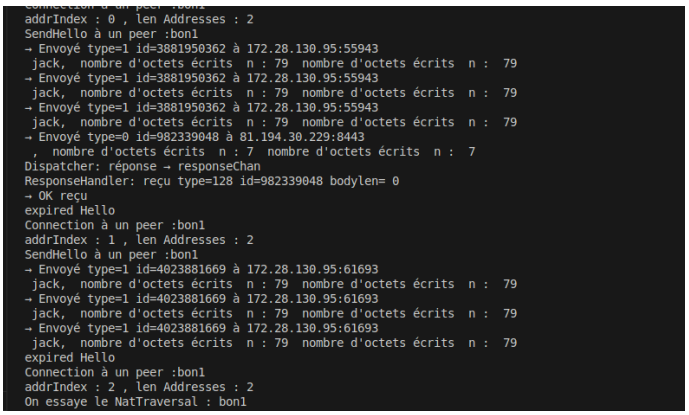

FIGURE 10 – Hello pour chaque addresse du pair bon1

References

- **Transaction ID et résolution** : Voir RFC 1035, Section 4.1.1, voir aussi RFC 5389, STUN.
- **Fenêtre glissante et limitation du flux** : Voir RFC 1122, Section 4.2.3.3.
- **Réévaluation d'adresse pour NAT Traversal** : la gestion des changements d'adresse pour les transactions Hello ou NatTraversal est inspirée des techniques STUN/TURN pour traverser les NAT. Voir RFC 5389, STUN.
- **Autres références utiles** :
  - https ://www.bittorrent.org/beps/bep$_0$003.*html*
  - https ://www.usenix.org/legacy/event/hotsec08/tech/full$_p$*apers/piatek/piatek.pdf*