

Projet Pluridisciplinaire d'Informatique Intégrative

TNFS

Loris Alexandre
Sangoan Brigué
Hugo Pagniez

Année 2021–2022

Déclaration sur l'honneur de non-plagiat

Je soussigné(e),

Nom, prénom : Alexandre, Loris

Élève-ingénieur(e) régulièrement inscrit(e) en 1^e année à TELECOM Nancy

Numéro de carte de l'étudiant(e) : 31903210

Année universitaire : 2021–2022

Auteur(e) du document, mémoire, rapport ou code informatique intitulé :

Telecom Nancy File System

Par la présente, je déclare m'être informé(e) sur les différentes formes de plagiat existantes et sur les techniques et normes de citation et référence.

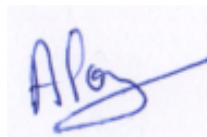
Je déclare en outre que le travail rendu est un travail original, issu de ma réflexion personnelle, et qu'il a été rédigé entièrement par mes soins. J'affirme n'avoir ni contrefait, ni falsifié, ni copié tout ou partie de l'œuvre d'autrui, en particulier texte ou code informatique, dans le but de me l'accaparer.

Je certifie donc que toutes formulations, idées, recherches, raisonnements, analyses, programmes, schémas ou autre créations, figurant dans le document et empruntés à un tiers, sont clairement signalés comme tels, selon les usages en vigueur.

Je suis conscient(e) que le fait de ne pas citer une source ou de ne pas la citer clairement et complètement est constitutif de plagiat, que le plagiat est considéré comme une faute grave au sein de l'Université, et qu'en cas de manquement aux règles en la matière, j'encourrais des poursuites non seulement devant la commission de discipline de l'établissement mais également devant les tribunaux de la République Française.

Fait à Nancy, le 23 mai 2022

Signature :



Déclaration sur l'honneur de non-plagiat

Je soussigné(e),

Nom, prénom : Brigué, Sangoan

Élève-ingénieur(e) régulièrement inscrit(e) en 1^e année à TELECOM Nancy

Numéro de carte de l'étudiant(e) : 31902075

Année universitaire : 2021–2022

Auteur(e) du document, mémoire, rapport ou code informatique intitulé :

Telecom Nancy File System

Par la présente, je déclare m'être informé(e) sur les différentes formes de plagiat existantes et sur les techniques et normes de citation et référence.

Je déclare en outre que le travail rendu est un travail original, issu de ma réflexion personnelle, et qu'il a été rédigé entièrement par mes soins. J'affirme n'avoir ni contrefait, ni falsifié, ni copié tout ou partie de l'œuvre d'autrui, en particulier texte ou code informatique, dans le but de me l'accaparer.

Je certifie donc que toutes formulations, idées, recherches, raisonnements, analyses, programmes, schémas ou autre créations, figurant dans le document et empruntés à un tiers, sont clairement signalés comme tels, selon les usages en vigueur.

Je suis conscient(e) que le fait de ne pas citer une source ou de ne pas la citer clairement et complètement est constitutif de plagiat, que le plagiat est considéré comme une faute grave au sein de l'Université, et qu'en cas de manquement aux règles en la matière, j'encourrais des poursuites non seulement devant la commission de discipline de l'établissement mais également devant les tribunaux de la République Française.

Fait à Nancy, le 23 mai 2022

Signature :



Déclaration sur l'honneur de non-plagiat

Je soussigné(e),

Nom, prénom : Pagniez, Hugo

Élève-ingénieur(e) régulièrement inscrit(e) en 1^e année à TELECOM Nancy

Numéro de carte de l'étudiant(e) : 110902578

Année universitaire : 2021–2022

Auteur(e) du document, mémoire, rapport ou code informatique intitulé :

Telecom Nancy File System

Par la présente, je déclare m'être informé(e) sur les différentes formes de plagiat existantes et sur les techniques et normes de citation et référence.

Je déclare en outre que le travail rendu est un travail original, issu de ma réflexion personnelle, et qu'il a été rédigé entièrement par mes soins. J'affirme n'avoir ni contrefait, ni falsifié, ni copié tout ou partie de l'œuvre d'autrui, en particulier texte ou code informatique, dans le but de me l'accaparer.

Je certifie donc que toutes formulations, idées, recherches, raisonnements, analyses, programmes, schémas ou autre créations, figurant dans le document et empruntés à un tiers, sont clairement signalés comme tels, selon les usages en vigueur.

Je suis conscient(e) que le fait de ne pas citer une source ou de ne pas la citer clairement et complètement est constitutif de plagiat, que le plagiat est considéré comme une faute grave au sein de l'Université, et qu'en cas de manquement aux règles en la matière, j'encourrais des poursuites non seulement devant la commission de discipline de l'établissement mais également devant les tribunaux de la République Française.

Fait à Nancy, le 23 mai 2022

Signature :



Projet Pluridisciplinaire d'Informatique Intégrative

TNFS

**Loris Alexandre
Sangoan Brigué
Hugo Pagniez**

Année 2021–2022

TELECOM Nancy
193 avenue Paul Muller,
CS 90172, VILLERS-LÈS-NANCY
+33 (0)3 83 68 26 00
contact@telecomnancy.eu

Table des matières

Table des matières	v
1 Introduction	1
2 Étude du fonctionnement d’IPFS	2
2.1 Gestion des données	2
2.1.1 Adressage des données	2
2.1.2 Hachage	2
2.1.3 Immuabilité des données	3
2.1.4 Persistence des données et recyclage de la mémoire	3
2.2 Merkle DAG	4
2.3 P2P : Table de hachage distribuée et routage	5
3 Conception et implémentation de l’application	7
3.1 Fonctionnement du système	7
3.2 Gestion des fichiers	8
3.2.1 Adressage du contenu	8
3.2.2 Division en blocs	10
3.3 Implémentation réseau	13
3.3.1 Les pairs	13
3.3.2 Les échanges réseau	14
3.3.3 Algorithme général d’un pair	16
4 Tests	17
5 Gestion de projet	20
5.1 Répartition des tâches	20
5.2 Communication	20
5.3 Gestion des sources	20
6 Conclusion	21
Bibliographie / Webographie	22
Liste des illustrations	23

Listings	24
Acronymes	25

1 Introduction

Dans le cadre du second Projet Pluridisciplinaire d'Informatique Intégrative de l'année, nous avons réalisé une réplique du système de fichier décentralisé **IPFS**.

IPFS est un système distribué de fichiers pair à pair conçu par Juan Benet. Le principe est simple : on associe une adresse statique (un identifiant) au fichier que l'on souhaite héberger sur le système, et on peut grâce à celui-ci y accéder quand l'on souhaite, peu importe où il est stocké. Pour créer l'identifiant, on se base sur le contenu du fichier et on la passe à travers une fonction de hachage, ce qui permet d'avoir un identifiant unique pour chaque fichier. Ce système permet alors de mettre en place un mécanisme de décentralisation, ce qui permet d'éviter les point unique de défaillance (ou "Single Point Of Failure" en anglais) car les données peuvent être hébergées à plusieurs endroits à la fois.

En d'autres termes, lorsqu'un utilisateur souhaitera récupérer un fichier, il va d'une part se connecter automatiquement à travers **IPFS** à des personnes ayant déjà le fichier en question, et une fois récupérée, il sera lui aussi dans la liste de personnes ayant ce fichier et en devient donc un hébergeur.

Tout d'abord, nous allons présenter les concepts clés de l'**IPFS** : l'identification du contenu, l'utilisation de graphe (**DAG**), et l'accès au contenu via le pair-à-pair (**P2P**). Nous verrons ensuite la phase d'implémentation de ces mécanismes ainsi que les tests effectués pour vérifier la qualité. Enfin, nous aborderons la gestion de ce projet.

2 Étude du fonctionnement d'IPFS

2.1 Gestion des données

2.1.1 Adressage des données

Chaque élément enregistré dans **IPFS** est relié à un **CID** ("Content Identifier" ou "Identifiant de Contenu" en français). Ce dernier ne fournit pas l'information d'où est-ce qu'est sauvegardé l'élément auquel il est rattaché, mais permet néanmoins de le retrouver car il est construit à partir du contenu de cet élément. Bien sûr, la taille de cet identifiant est minime par rapport au contenu de l'élément qu'il définit. Le fait qu'il soit basé sur le contenu d'un élément implique que n'importe quel changement sur ce contenu engendrera un changement de **CID**. De la même façon, un même fichier sauvegardé sur deux différents noeuds **IPFS** aura le même **CID**.

On distingue deux versions de format de cet identifiant. La première, nommée CIDv0, est un multihash encodé en base 58 et est toujours utilisé pour un grand nombre d'opérations **IPFS** basiques. Autrement dit, si un **CID** est composé de 46 caractères et qu'il commence par "Qm", c'est qu'il s'agit d'un CIDv0. La seconde version, nommée CIDv1, est composée de multiples identifiants préfixant le contenu haché qui permettent d'apporter un grand nombre d'informations sur celui-ci. Premièrement, il y a un préfixe multibase qui spécifie le type d'encodage utilisé pour encoder le reste du **CID**, ensuite un identifiant de version du **CID** et enfin un identifiant multicodec qui indique le format de l'élément qu'identifie le **CID**. Cette version va bientôt être considérée comme par défaut par le projet **IPFS**.

2.1.2 Hachage

En cryptographie, les fonctions de hachage permettent de prendre n'importe quel type de donnée en entrée afin d'en retourner une valeur à taille fixe. Cette valeur de retour dépend bien évidemment de l'algorithme de hachage utilisé. Ces haches peuvent être encodées en différentes bases. Comme vu précédemment, dans la version 1 du **CID**, **IPFS** supporte ces différents types d'encodage à travers la définition du préfixe multibase.

Les haches sont définis par plusieurs caractéristiques qu'il ne faut pas négliger. En effet, le résultat du hachage d'un même message par la même fonction de hachage sera toujours identique (déterminisation); n'importe quelle modification d'un message, même minime, produira un hache complètement différent (indépendance); il est impossible de générer un même hache avec deux messages différents (unicité); une fois un message haché, il n'est plus possible de retrouver le message initial (sens unique). Ces caractéristiques signifient qu'un hache peut être utilisé pour identifier n'importe quelle sorte de donnée puisqu'il est unique et de taille raisonnablement courte, ce qui permet de le faire naviguer à travers le réseau sans conséquences de performances. Cette dernière remarque est critique au bon fonctionnement de l'**IPFS** car il est essentiel de rechercher rapidement un fichier à travers le réseau et il ne serait pas concevable de faire sans un identifiant unique à chacun de ces fichiers.

2.1.3 Immuabilité des données

Un objet immuable est un élément dont l'état ne peut être altéré ou modifié une fois qu'il est créé. A travers **IPFS**, cela signifie, qu'une fois qu'un fichier est ajouté au réseau, il n'est pas possible de le modifier sans également modifier son **CID**. On rencontre alors un sérieux problème lorsqu'il s'agit de satisfaire ce besoin. Le **CID** est donc immuable peu importe le contexte dans lequel il est utilisé. On considère ce **CID** comme un pointeur vers un élément du réseau. Pour pouvoir utiliser des objets immuables dans un environnement muable, il faut ajouter un autre principe en surcouches. On peut par exemple citer le système de l'**IPNS**, pour "InterPlanetary Name System" qui permet d'associer à une adresse fixe un **CID** qui peut être modifié à travers le temps, ce qui permet de modifier une donnée et d'y accéder via la même adresse que la version précédente.

2.1.4 Persistence des données et recyclage de la mémoire

L'un des objectifs de l'**IPFS** est de préserver l'histoire de l'humanité en permettant aux utilisateurs de sauvegarder leurs données en minimisant la possibilité qu'elles disparaissent accidentellement. **IPFS** permet en réalité de conserver chacune des versions du fichier sauvegardé et de le retrouver facilement.

Lorsqu'un fichier est téléchargé, il est automatiquement enregistré dans le cache pour permettre d'y accéder plus rapidement la prochaine fois. Ce système dépend en réalité du nombre de nœuds qui souhaite participer au cache et au partage des données. Si on part sur ce principe, le stockage est infini donc les nœuds ont besoin de faire de la place pour mettre en cache les nouvelles ressources fraîchement téléchargées en supprimant les plus anciennes ou les moins populaires. Ce principe est appelé 'garbage collection'.

C'est une forme de gestion automatique des ressources largement utilisée dans n'importe quelle solution de développement logicielle. La 'garbage collection' tente de recycler la mémoire utilisée par de la donnée qui n'est plus exploitée. Au sein de l'**IPFS**, ce principe est utilisé pour libérer de l'espace mémoire d'un nœud en supprimant les données qui ne nécessitent plus d'être conservées.

Pour garantir qu'un fichier persiste dans l'**IPFS**, il est nécessaire d'utiliser le principe d'accrochage. L'accrochage permet de gérer manuellement l'espace mémoire d'un nœud pour retenir de la donnée et éviter que la garbage collection ne la supprime/remplace. Un cas d'usage majeur de cette fonctionnalité est la conservation indéfini d'un fichier.

2.2 Merkle DAG

On a vu précédemment que pour chaque fichier que l'on ajoute au système **IPFS**, une adresse d'identification lui a été donnée. Mais en réalité, il y a une autre étape qui est effectuée entre le moment d'import et le moment où on nous donne l'identifiant : le découpage du fichier en de multiples sous-parties.

En réalité, un **CID** ne référence pas un fichier, mais un nœud dans un graphe orienté acyclique que l'on appelle **DAG** (Directed Acyclic Graph). Chaque nœud possède alors :

- Une adresse d'identification (**CID**)
- Des données (**le payload**)
- Une liste de sous-nœuds (**les fils du nœud courant**)

Regardons de plus près le Merkle DAG suivant :



FIGURE 2.1 – Exemple de Merkle DAG

On voit donc sur l'illustration que nous avons voulu ajouter un fichier de 506 kb à notre système **IPFS**, comme il dépasse la taille maximale par défaut de l'**IPFS**, c'est-à-dire 256 kb, il a alors été découpé en deux nœuds (0 et 1). Ces deux nœuds contiennent alors une quantité de données de respectivement 256 kb et 250 kb (en additionnant les quantités, on retrouve bien la taille du fichier initial). En haut de l'arbre, nous avons le nœud parent qui contient le **CID** qui est l'unique référence du fichier complet, et on retrouve également les différents liens vers ses sous-nœuds. Ce principe permet alors de propager des données plus rapidement, en effet, une personne n'est plus obligée d'héberger l'intégralité du fichier pour le rendre accessible.

Ce principe permet alors de propager des données plus rapidement. En effet, une personne n'est plus obligée d'héberger l'intégralité du fichier pour le rendre accessible. Dans le cas présent, on peut par exemple imaginer qu'une personne stocke le nœud 1 sur son ordinateur, et que le nœud 0 soit stocké chez un autre utilisateur. Lors de la récupération du fichier par un tiers, **IPFS** récupérera les différents nœuds liés à un **CID**, et cela où qu'ils soient stockés.

2.3 P2P : Table de hachage distribuée et routage

Comme vu précédemment, on sait que les différentes données sont hébergées par petit bout et que chaque personne peut en avoir une partie. Pour trouver quel pair héberge le contenu auquel on souhaite accéder, on utilise un système de table de hachage. Une table de hachage est une base de données dans laquelle chaque clé est associée à une valeur. Lorsque l'on souhaite récupérer le contenu lié à un certain **CID**, on cherche à savoir quels pairs l'héberge, on récupère alors cette information via cette table de hachage. Une fois que les pairs sont identifiés, on utilise une autre table de hachage pour trouver leurs locations et leurs distances : c'est la table de routage.

Cependant, **IPFS** utilise une autre version : les tables de hachage distribuées (**DHT**). Cela permet de l'utiliser à travers toutes les pairs du réseau. Pour construire cette table de hachage distribuée et notamment gérer la table de routage, **IPFS** utilise l'algorithme Kademlia.

Dans Kademlia, chaque pair du réseau est identifié de façon unique (dans le cas d'**IPFS**, on utilise tous les nombres de 0 à 2^{256-1}). Chaque pair dispose alors d'une adresse pour pouvoir être contacté ainsi que de sa propre table de routage contenant les adresses des pairs qu'elle connaît. Au démarrage, un pair va automatiquement se connecter à des pairs prédéfinies afin de leur demander des informations sur les autres pairs qu'ils connaissent.

Dans la table de routage, on retrouve des informations sur les autres pairs réseaux. Cependant, on ne peut pas stocker les informations sur toutes les pairs du réseaux. En effet, les pairs viennent et partent du réseau, il serait donc difficile de maintenir à jour la table de routage. De plus, la taille de pairs stockées est limitée au nombre de bits qui composent l'identifiant unique. La table de routage ne stocke donc que les pairs qui sont à 2^N de distance, N étant le nombre maximum de pairs à enregistrer. On peut représenter les pairs du réseau via un arbre binaire tel qu'il suit :

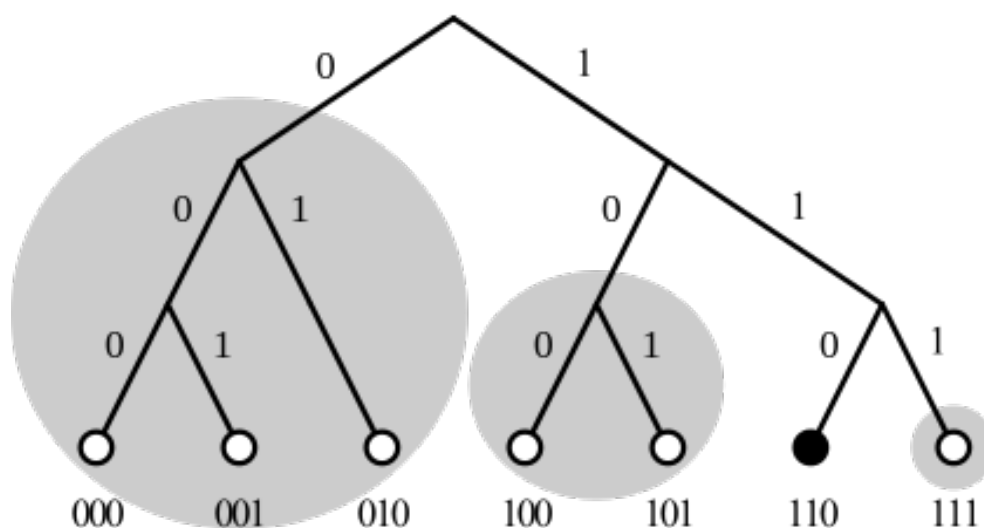


FIGURE 2.2 – Arbre binaire Kademlia

Ici, on peut représenter des sous-groupes de pairs. Chaque pair doit référencer dans sa table de routage au moins une pair de chaque sous-groupe. Cela permet de retrouver chaque pair lors de l'algorithme de recherche. Cependant, la distance reste une notion abstraite.

Kademlia définit alors sa propre fonction de distance, qui n'a aucun rapport avec la distance physique entre deux pairs. La fonction distance utilisée est l'opérateur XOR entre les identifiants de pairs. Par exemple, si on cherche à déterminer la distance entre le pair 10 et le pair 13, on obtient une distance de 7. En effet, $10 \text{ XOR } 13 = 7$.

On peut détailler le calcul en passant en binaire : $001\ 010 \text{ XOR } 001\ 101 = 000\ 111$

Ainsi, si on prend un réseau de 16 pairs et que le pair 0 souhaite faire un appel au pair 13, on va calculer la distance entre les pairs présentes dans la table de routage et le pair 13. On a donc $1 \text{ XOR } 13 = 12$, $2 \text{ XOR } 13 = 15$, $4 \text{ XOR } 13 = 9$, $8 \text{ XOR } 13 = 5$. On effectue alors un appel au pair 8 pour lui demander de trouver le pair 13. Il va réaliser la même opération avec les pairs de sa table de routage, et on effectue ce processus jusqu'à trouver le chemin vers le pair 13.

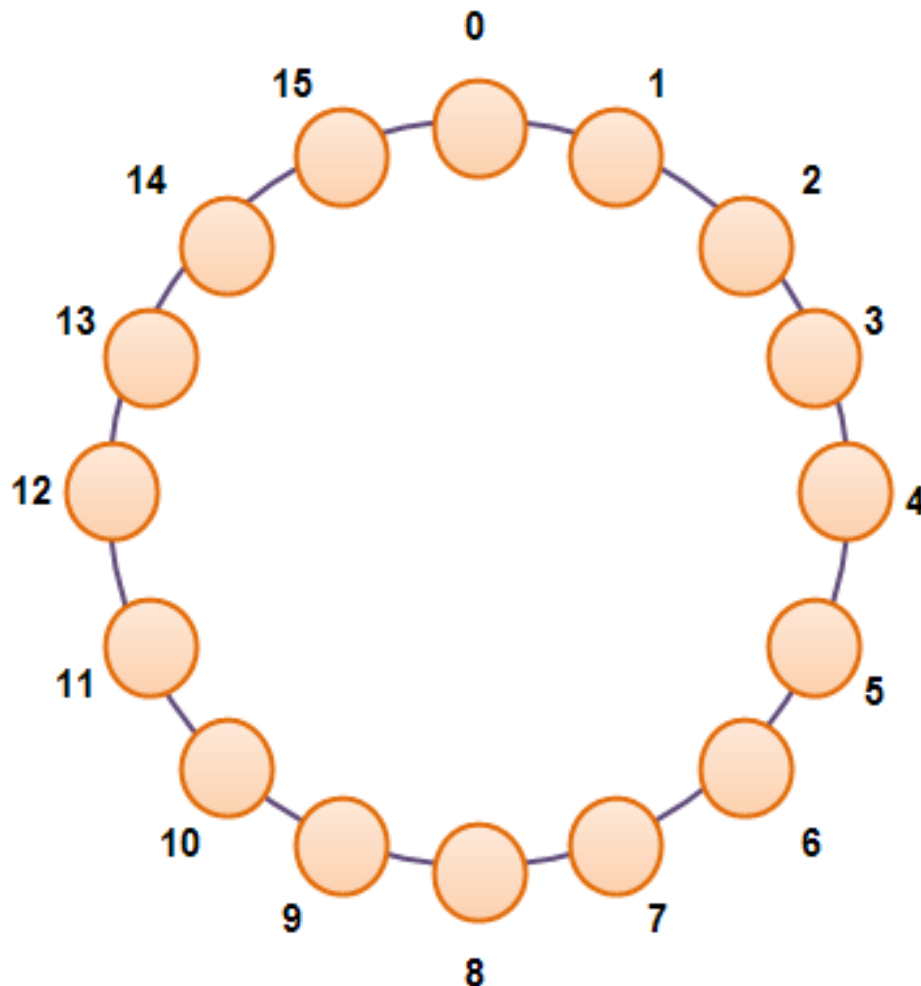


FIGURE 2.3 – Réseau de 16 pairs

Dans le cas où un pair viendrait à quitter le réseau, on va chercher à le pair le plus proche du pair sortant pour le remplacer. Grâce à cet algorithme, on peut retrouver rapidement un contenu. D'un point de vue mathématique, la complexité de la recherche est estimée à $O(n \log(n))$.

3 Conception et implémentation de l'application

Notre projet est divisé en deux parties, la première que nous présenterons dans ce rapport concerne l'ensemble du système de gestion de fichier : que ce soit l'adressage, le hashage, ou la division en blocs. Nous parlerons ensuite de l'implémentation réseau afin de faire communiquer les pairs, et de pouvoir échanger des données. Nous parlerons également de l'algorithme général d'un pair, afin de comprendre la logique et le fonctionnement global.

Mais dans un premier temps, nous allons vous présenter les fonctionnalités implémentées dans notre projet et la façon de les utiliser.

3.1 Fonctionnement du système

Pour la réalisation de notre projet, nous avons décidé de le découper en deux exécutable :

- **tnfs-server** : permet d'écouter les demandes venant des autres pairs et de leurs envoyer les données demandées, il est important de laisser tourner cette application afin d'être disponible pour les autres personnes du réseau et de permettre le bon fonctionnement de TNFS.
- **tnfs** : coeur de l'application, on peut entre autre grâce à celui-ci ajouter des fichiers à TNFS et en télécharger. Pour cela, on passe plusieurs arguments lors du lancement de l'exécutable afin de réaliser l'opération que l'on souhaite.

Voici les commandes disponibles dans l'application cliente de TNFS :

- **tnfs add <filename>** : permet d'ajouter un fichier au système TNFS. L'exécution de cette commande retournera alors un identifiant. On peut alors donner cet identifiant à un autre utilisateur afin qu'il puisse, via TNFS, récupérer le fichier.
- **tnfs get <identifiant>** : permet de récupérer, à travers le réseau TNFS, le fichier lié à l'identifiant passé en paramètre.
- **tnfs infos <identifiant>** : permet de récupérer les informations correspondant à l'identifiant passé en paramètre, comme par exemple la taille du fichier ou son nom.
- **tnfs peer add <IP> :<PORT>** : permet d'ajouter un utilisateur à la liste des pairs à contacter lors de la récupération d'un CID. Pour cela, on indique son IP et le port sur lequel écoute le tnfs-serveur de l'utilisateur en question.
- **tnfs peer remove <IP> :<PORT>** : permet de retirer un pair de la liste des pairs.
- **tnfs clean peers** : permet de vider la table contenant l'ensemble des pairs à contacter lors de la récupération d'un CID.
- **tnfs clean blocks** : permet de vider la table Redis contenant la correspondance entre les CID et les blocs que nous avons localement.

Nous avons également décidé d'utiliser Redis, un système de gestion de base de données (SGBD) de type NoSQL proposant un système clé-valeur. Cela nous permet d'avoir un endroit où sont centralisées les données afin qu'elles puissent être utilisées par les deux exécutables en même temps. Afin de communiquer avec Redis via notre code, nous utilisons une librairie : hiredis.

3.2 Gestion des fichiers

3.2.1 Adressage du contenu

Dans le but de satisfaire pleinement nos besoins, nous avons décidés d'adapter une version personnalisée du **CID**. Il est issu de plusieurs manipulations informatiques expliquées ci-après.

En premier lieu, on trouve un caractère qui définit le type d'encodage utilisé pour le reste du **CID**, appelé Payload. Ce caractère est choisi méticuleusement en suivant les normes de nommage multibase [5]. Par exemple, dans notre cas, nous avons décidé d'utiliser la Base32, ce qui signifie que chacun de nos **CID** seront préfixés par un « b ».

Une fois décodé en utilisant le type d'encodage défini par le premier caractère, le payload apparaît en réalité sous la forme d'une suite de caractères hexadécimaux. Ils correspondent, successivement, à la version du **CID** ($v1 = 0x01$), au code hexadécimal de l'algorithme de hachage défini par la norme de nommage multicodec [4] utilisé pour hacher le contenu du fichier que référence ce **CID** ($\text{sha-256} = 0x12$), à la taille du hache en octets en hexadécimal ($256 \text{ bytes} = 32 \text{ Octets} = 0x20$) et le reste des caractères est la version hexadécimal du hache.

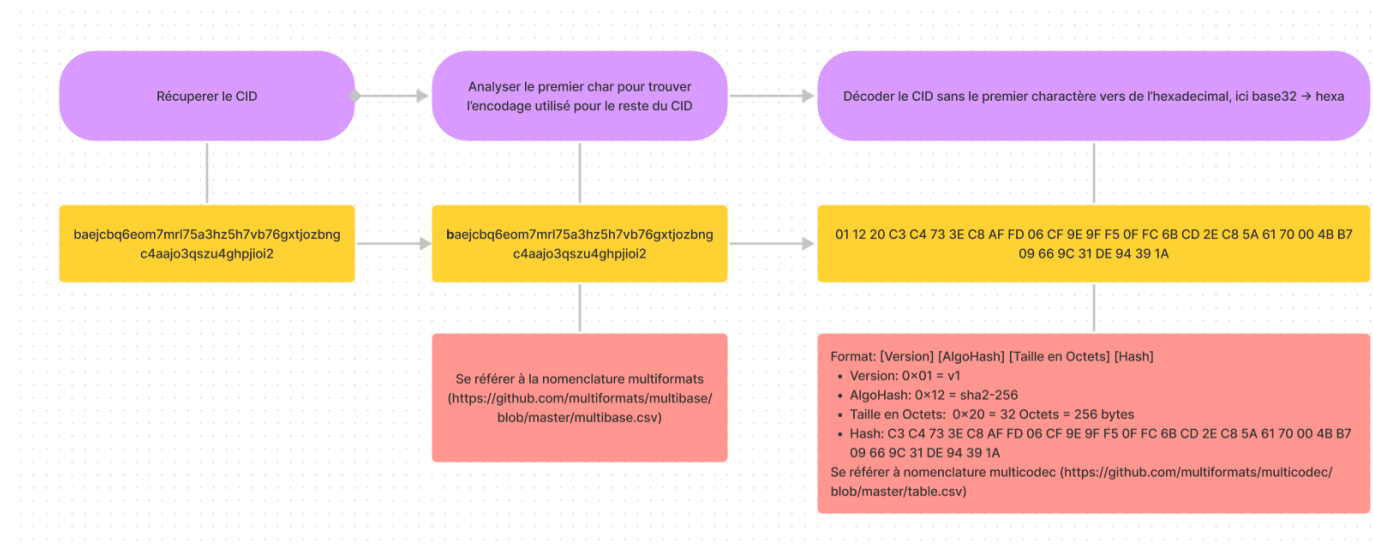


FIGURE 3.1 – Diagramme du processus de décodage du CID

De cette façon, pour garantir l'intégrité d'un fichier il suffira simplement de suivre le processus précédemment décrits pour récupérer le hache du **CID** et de le comparer à celui du fichier courant.

Concernant l'implémentation nous avons décidé d'utiliser des libraires C de hachage en Sha256 et d'encodage en Base32 déjà disponible sur le net. Néanmoins, la composition du **CID** reste effectuée par nos soins programmatalement :

```
1 void generate_cid_from_file_content(char* input, size_t input_length, char*
   cid)
2 {
3     uint8_t *hash = (uint8_t *) malloc(sizeof(uint8_t) * SIZE_OF_SHA_256_HASH);
4     calc_sha_256(hash, input, input_length);
5     char *hash_string = (char *) malloc(sizeof(char) * HASH_LENGTH);
6     hash_to_string(hash, hash_string);
7     char* hash_headers = "011220";
8
9     char* plain_hash = (char *) malloc(sizeof(char) * PLAIN_HASH_LENGTH);
10    sprintf(plain_hash, "%s%s", hash_headers, hash_string);
11    // log_formated_debug("HashString: %s", hash_string);
12    // log_formated_debug("PlainHash: %s", plain_hash);
13
14    char* encoded = (char *) malloc(ENCODED_HASH_LENGTH);
15    base32_encode((const unsigned char*) plain_hash, PLAIN_HASH_LENGTH, (
16    unsigned char*) encoded);
17    encoded[112] = '\0';
18    // log_formated_debug("Encoded: %s", encoded);
19
20    sprintf(cid, "b%s", encoded);
21    // log_formated_debug("Cid: %s", cid);
22
23    free(encoded);
24    free(hash);
25    free(hash_string);
26    free(plain_hash);
27 }
```

Listing 3.1 – Extrait de code de la fonction de composition du CID

3.2.2 Division en blocs

Maintenant que nous pouvons générer un **CID** en fonction d'un fichier, nous avons alors décidé de mettre en place une autre fonctionnalité qui est très importante dans le système IPFS : le découpage en bloc.

Lorsque nous souhaitons ajouter un fichier à TNFS, il est divisé en multiples blocs contenant chacun 256 kilo-octets de données, permettant alors de le faire transiter plus rapidement entre les peers et de le décentraliser plus facilement. Par exemple, si nous décidons d'ajouter un fichier de 12 mega-octets à notre système, nous aurons 47 blocs (12 000 kb/256) contenant chacun une partie des données du fichier, ainsi qu'un autre bloc supplémentaire qui correspond au bloc 'parent', que nous verrons par la suite.

Structure d'un bloc

En ce qui concerne la structure d'un bloc, il faut d'abord savoir qu'il y a deux types de bloc : les blocs 'parents' et les blocs 'fils'. Un bloc 'fils' contient uniquement des données de fichier, alors qu'un bloc 'parent' sert de pointeur et contient entre autres : le nom du fichier initial et la liste des **CID** de tous les blocs contenant les données du fichier initial. C'est donc le **CID** de ce bloc qui est transmis à l'utilisateur lorsqu'il ajoute un fichier à TNFS.

Nous avons choisi d'utiliser une structure commune à ces deux types :

```
1 typedef struct dn {
2     int type;
3     char filename[256];
4     int nbLinks;
5     char links[MAX_LINKS][CID_LENGTH + 1];
6     unsigned char data[BLOCK_LIMIT];
7     int size;
8 } DAGNode;
```

Listing 3.2 – Structure d'un bloc

Voici l'explication pour chacune des variables de la structure :

- **type** : permet de savoir si le bloc est de type 'parent' (type=0), ou de type 'fils' (type=1).
- **filename** : dans le cas d'un bloc 'parent', la variable contient le nom du fichier initial, par exemple 'text.txt'. Dans le cadre d'un bloc 'fils', la variable est nulle.
- **nbLinks** : contient une valeur représentant le nombre de blocs fils qu'à ce bloc.
- **links** : un tableau contenant l'ensemble des **CIDs** des fils de ce bloc.
- **data** : contient une partie des données d'un fichier quand c'est un bloc 'fils' et est vide si ce n'est pas le cas.
- **size** : contient la taille du fichier initial dans le cadre d'un bloc 'parent', ou alors contient la taille des données stockées dans le cadre d'un bloc 'fils'.

Découpage d'un fichier

Dans notre application, la première étape est de découper le fichier passé en paramètre de la commande 'add' (tnfs add <filename>) en plusieurs blocs contenant 256 kb de données maximum. Ce bloc aura un nom basé sur le contenu qu'il contient afin d'avoir un nom unique pour éviter les doublons localement, pour cela, nous hashons le contenu en sha-256.

Une fois le fichier du bloc créé, nous générons son **CID**, puis nous ajoutons celui-ci à un tableau de **CID**, qui va contenir l'ensemble des **CID** contenant les données du fichier initial dans l'ordre de création des blocs, afin de pouvoir le ré-assembler correctement lorsqu'on voudra le récupérer.

Lorsque toutes les données ont été découpées en blocs, nous créons un bloc parent afin d'y stocker : le nom du fichier initial, sa taille, ainsi que le tableau des **CID** 'fils' créé précédemment. Nous générons ensuite son **CID** et c'est celui-ci que nous retournons à l'utilisateur.

Concernant l'algorithme implémenté dans le code, en voici une version en pseudo-code permettant de mieux comprendre la façon dont il marche :

```
1 creer_les_fils(données, taille_données, nombre_blocs_necessaire):
2   bit_maximum <- 256000
3   tableau_cids <- new tableau[nombre_blocs_necessaire]
4
5   // Divison en blocs
6   pour i de 0 à nombre_blocs_necessaire :
7     taille_bloc_courant <-- 0;
8     données_bloc_courant <-- new tableau[bit_maximum]
9
10    pour j de 0 à bytes_maximum :
11      si taille_données == 0:
12        break
13      sinon:
14        données_bloc_courant[j] <- données[(i*bit_maximum) + j]
15        taille_données <- taille_données - 1
16        taille_bloc_courant <- taille_bloc_courant + 1
17
18    // Création du bloc
19    bloc_fils.type <- 1
20    bloc_fils.nbLinks <- 0
21    bloc_fils.size <- taille_bloc_courant
22    bloc_fils.filename <- null
23    bloc_fils.data <- données_bloc_courant
24
25    // Génération du nom du bloc
26    nom_fichier <- generer_nombloc(données_bloc_courant);
27
28    // Écriture du fichier
29    ecrire_fichier(nom_fichier, bloc_fils)
30
31    // Génération du CID
32    cid <- generer_cid(nom_fichier)
33    tableau_cids[i] <- cid
34
35    // Ajout dans la table redis
36    set_redis(cid, nom_bloc)
37
38  retourne tableau_cids
```

Listing 3.3 – Découpage d'un bloc

Utilisation de Redis

Lors de la phase de découpage d'un fichier, nous créons des blocs et leur donnons un nom correspondant à leur contenu en sha-256. Afin de savoir quel **CID** se trouve actuellement sur la machine courante et surtout de savoir quel bloc correspond à quel **CID**, nous avons alors mis en place une base de données Redis.

Chaque fois que nous créons un bloc, nous ajoutons alors une association de la forme **CID => NOM DU BLOC SUR LA MACHINE** à la table Redis, comme nous pouvons le voir sur le pseudo-code ci-dessus.

Le choix de Redis s'explique par le fait que nous avons besoin d'association clé-valeur, et que Redis, en plus d'être un **SGBD** portable et très peu coûteux en ressource, répond à ce besoin.

Ré-assemblage d'un fichier

Grâce au client de TNFS, nous pouvons récupérer un fichier sur l'entièreté du réseau via son **CID**, et cela, grâce à la commande 'tnfs get <CID>'.

La première étape ici était alors d'implémenter une fonction 'lire_bloc' permettant de lire un bloc sur sa machine en fonction d'un **CID**. Pour cela, nous regardons dans la table Redis si nous possédons dans les clés le **CID** que l'on recherche. Si oui, alors on récupère le nom du bloc sur la machine, sinon, on demande aux pairs de notre réseau s'ils peuvent nous donner ce bloc (nous verrons de quelle façon plus tard). Une fois que nous avons le nom du bloc sur la machine, nous le lisons et vérifions qu'il correspond bien au **CID** initial, en régénérant son **CID**. Dans le cas où le **CID** serait mauvais, nous supprimons ce bloc et l'association dans la table redis, et nous essayons de le re-télécharger à travers le réseau TNFS. Dans le cas où le **CID** est correcte, nous lisons la structure du bloc.

Voici en pseudo-code le fonctionnement de l'algorithme :

```
1 lire_bloc(CID)
2   nom_du_bloc <- redis_get(CID)
3
4   si nom_du_bloc == null:
5       /* demande aux pairs s'ils ont le CID,
6          si oui on le récupère + ajout à redis,
7          directement dans la fonction demander_pais */
8       demander_pairs(CID)
9   sinon:
10      bloc_CID <- generer_cid(nom_du_bloc)
11      si bloc_CID == CID:
12          structure <- lire_fichier(nom_du_bloc)
13      sinon:
14          supprimer_redis(CID)
15          supprimer_fichier(nom_du_bloc)
16          demander_pairs(CID)
17          lire_bloc(CID)
18
19  retourne structure
```

Listing 3.4 – Lecture d'un bloc

Une fois cette méthode implémentée, nous pouvions alors mettre en place le téléchargement d'un fichier via son **CID**. Pour cela, c'est simple : on récupère d'abord le bloc grâce à la méthode de lecture du bloc et on vérifie bien s'il est de type 'parent'. Dans le cas où cette condition serait validée, on peut alors commencer à écrire le fichier en lui donnant le nom du fichier dans la structure du bloc parent.

On boucle ensuite sur la liste des **CIDs** que contient ce bloc parent et pour chacun de ces **CIDs** : on lit le bloc et on ajoute les données qu'il contient au fichier que l'on est en train de créer.

3.3 Implémentation réseau

Une fois que l'ensemble du système de gestion des fichiers a été mis en place, nous avons alors pu nous concentrer sur l'implémentation du réseau. Le but est de faire communiquer des pairs entre eux afin de permettre l'échange et la récupération de blocs pour pouvoir récupérer un fichier via son **CID**.

3.3.1 Les pairs

Chaque pair du réseau est identifiée par une adresse IP et un port, en voici alors la structure dans notre code :

```
1 typedef struct p {
2     char* ip;
3     int port;
4 } Peer;
```

Listing 3.5 – Structure d'un pair

Dans le cadre d'une évolution future du système, il sera préférable d'ajouter un ID correspondant aux contraintes de Kademlia, afin d'avoir un système ressemblant plus à IPFS.

Afin que les pairs soient commun au client et au serveur tnfs, nous les stockons dans une table Redis. Pour éviter de les mélanger avec les blocs, on change d'instance : la première instance (0) est donc dédiée aux blocs tandis que la seconde instance (1) est consacrée aux pairs. Dans un premier temps, on préféra stocker les pairs uniquement en local pour simplifier le développement.

Nous avons alors implémenté plusieurs fonctions dans notre code afin de gérer l'ajout de pair ainsi que leur suppression. Nous pouvons également récupérer l'ensemble des pairs que nous avons stocké localement grâce à la fonction `get_all_peers` :

```
1 Peer** get_all_peers(int* results);
```

Listing 3.6 – Définition de `get_all_peers`

Cette méthode nous permet alors d'avoir la liste de tous les pairs enregistrés localement. Pour cela, on utilise la commande 'KEYS *' de Redis à travers hiredis dans notre code. Nous utiliserons cette méthode lorsque nous rechercherons un CID que nous n'avons pas sur notre machine. En effet, lorsqu'on récupère un CID, on regarde d'abord dans notre table Redis si nous n'avons pas le bloc qui lui correspond sur notre machine. Et dans le cas où nous n'avons pas le bloc, nous interrogeons alors à l'ensemble de nos pairs s'ils l'ont afin qu'ils nous envoient ce bloc.

3.3.2 Les échanges réseau

Les principaux échanges à implémenter sont les suivants : échange de fichier et demande de **CID**. Pour ces échanges, le client nécessite une réponse de la part du serveur. On utilisera donc le protocole **TCP** pour les implémenter.

Client

Du côté client, nous avons besoin de contacter nos pairs quand nous n'avons pas un **CID** (plus précisément un bloc) sur notre machine, afin de pouvoir le récupérer auprès des pairs le possédant. L'échange avec les pairs se fait alors une méthode appelée lorsque nous ne trouvons pas le **CID** sur notre machine (méthode lire_bloc).

Afin de nous faciliter le développement, nous avons implémenté plusieurs fonctions qui seront utilisées dans notre méthode de récupération d'un bloc :

Tout d'abord, nous avons une méthode permettant de créer une connexion **TCP** en fonction d'un pair (structure contenant IP et Port comme vu précédemment) passé en paramètre. Elle nous renvoie un descripteur ouvert :

```
1 int create_socket(Peer* p);
```

Listing 3.7 – Définition de create_socket

Nous avons également une méthode permettant de fermer le socket ouvert précédemment en fonction de son descripteur :

```
1 void close_socket(int sockfd);
```

Listing 3.8 – Définition de close_socket

Nous avons également une méthode permettant d'envoyer un message à un serveur en passant en paramètre le descripteur de la connexion **TCP** et le message qu'on souhaite envoyé.

```
1 int send_tcp(int sockfd, char* buffer);
```

Listing 3.9 – Définition de send_tcp

Maintenant, pour ce qui concerne l'échange entre le client et un pair, comme nous pouvons le voir sur la figure 3.2, il est plutôt simple. Le client demande d'abord un **CID** au serveur, qui répond 'Y' s'il possède le bloc, ou 'N' dans le cas contraire.

Dans le cas où le pair posséderait le bloc, le client va alors envoyer 'BLOCK' pour le lui demander, le serveur répondra alors en lui envoyant d'abord la taille en octet du fichier, et enverra ensuite les données du bloc. Une fois le fichier du bloc écrit sur notre machine, on l'ajoute à notre table Redis.

Dans le cas où le pair ne posséderait pas le bloc, le client va alors envoyer 'PEERS' afin de récupérer les pairs que possède le pair distant. Le serveur nous répondra alors en nous donnant le nombre de pairs qu'il va nous envoyer et nous enverra ensuite l'ensemble des pairs qu'il possède. Ces pairs seront alors ajoutés dans notre table de pair et nous les contacterons également pour le bloc que nous recherchons.

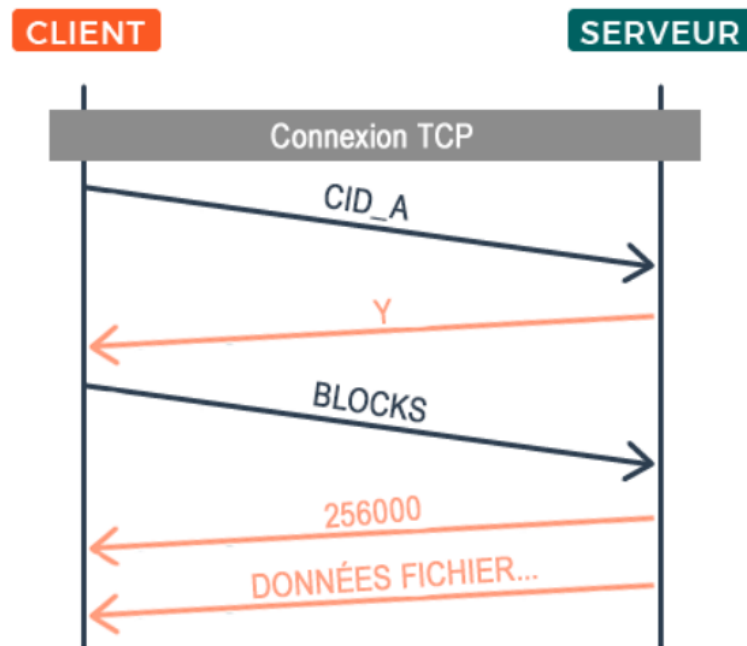


FIGURE 3.2 – Échanges entre le client et le serveur d'un pair

Serveur

Le deuxième exécutable que doit lancer un utilisateur est `tnfs-server`, il correspond à un serveur qui sera à l'écoute des autres pairs en arrière plan. Dans le futur, il est envisageable d'automatiser ce processus à l'initialisation de TNFS.

Ce serveur est donc une application qui attend des requêtes TCP. Afin de gérer plusieurs clients simultanément, nous avons utilisé l'appel système `fork()` à chaque fois que le serveur reçoit une demande de connexion. Pour chaque écoute, on a alors la même communication :

Une fois la connexion TCP établie, le serveur attend de recevoir un **CID** de la part du client, il vérifiera ensuite dans sa table Redis si le bloc associé à ce **CID** est bien hébergé sur la machine. Le serveur enverra alors soit 'Y' s'il possède le bloc, soit 'N'.

Le serveur attends maintenant le message du client :

- soit '**BLOCKS**' : le serveur va alors envoyer au client un premier message pour lui indiquer la taille du fichier qu'il va lui transférer, puis il commencera l'envoi du fichier.
- soit '**PEERS**' : dans ce cas, le client demande au serveur de lui envoyer la liste de ses pairs afin qu'il puisse les contacter par la suite. Le serveur va alors envoyer au client un premier message afin de lui indiquer le nombre de pairs qu'il va lui envoyer, puis il envoie l'ensemble de ses pairs présent dans sa table Redis.

Une fois ces échanges effectués, le serveur et le client ferment la connexion TCP.

3.3.3 Algorithme général d'un pair

Nous avons choisi dans un premier temps d'implémenter un algorithme simple pour faire fonctionner TNFS et tester toutes nos méthodes réseau. Nous avons représenté ce qu'il se passe dans la méthode 'demander_pairs(CID)' lorsque l'utilisateur souhaite récupérer un **CID** qu'il ne possède pas :

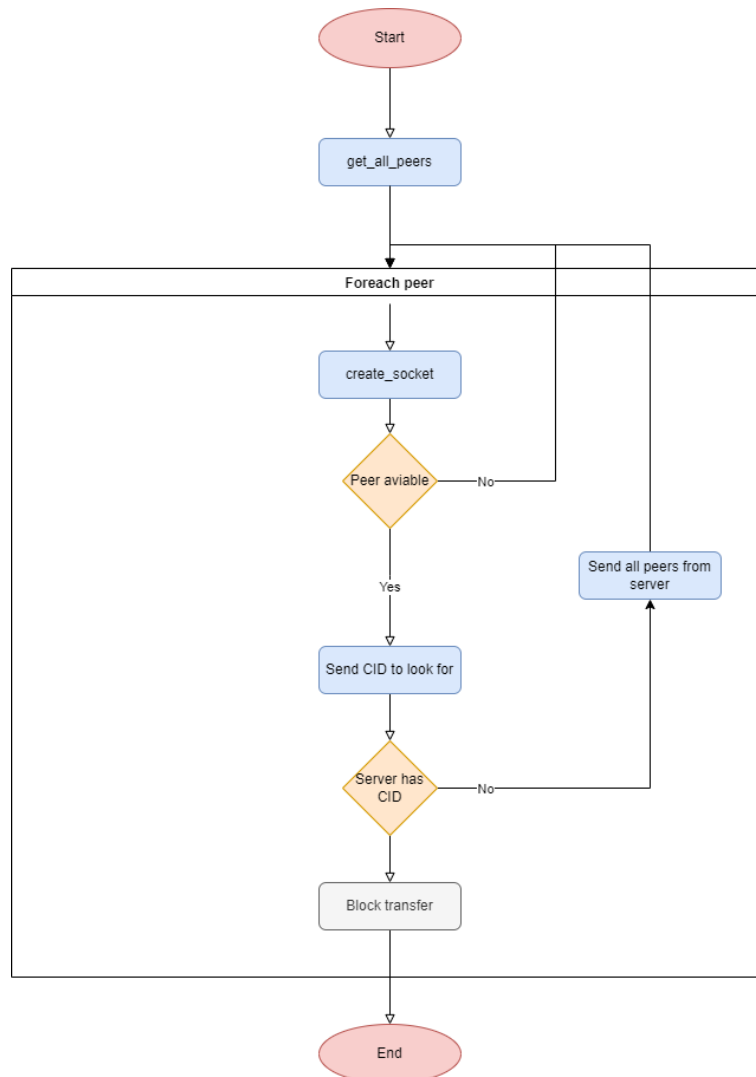


FIGURE 3.3 – Algorithme général d'une pair

Tout d'abord, on récupère tous les pairs connus par l'utilisateur dans sa table Redis. On va ensuite demander à chaque pair si elle dispose du bloc que l'on cherche. Pour cela, on ouvre une connexion vers le pair comme expliqué précédemment. On va alors soit récupérer le bloc recherché et pouvoir alimenter notre table Redis, soit récupérer la liste des pairs que connaît le pair en face et pouvoir les contacter également.

Cette méthode peut être comparée à du brute-force, mais nous permet aisément de tester la plupart des fonctions implémentées et d'avoir une solution fonctionnelle.

4 Tests

Pour tester TNFS, nous allons utiliser tout d'abord tester en local. Pour cela, on configure deux machines virtuelles Ubuntu sur VirtualBox :

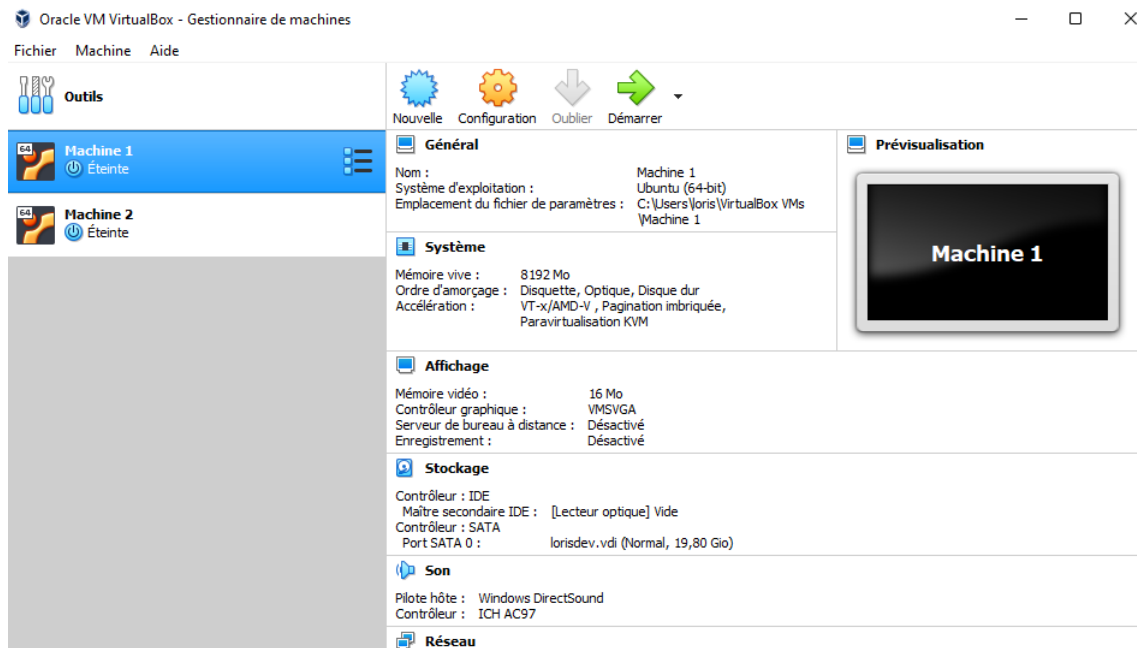


FIGURE 4.1 – Deux machines virtuelles

Pour que ces machines puissent communiquer entre elles, on utilise un mode de communication par pont en réseau :

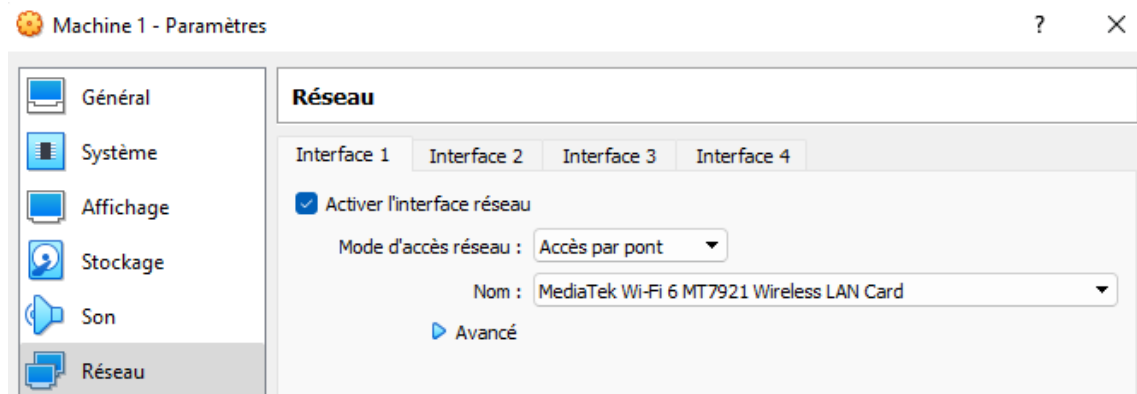


FIGURE 4.2 – Deux machines virtuelles

On allume ensuite les deux machines virtuelles. La première disposera du fichier et la seconde cherchera à récupérer ce fichier. Sur la première machine, on ajoute le fichier de test à IPFS :

```
lorisdev@lorisdevm:~/Projets/ppii-tnfs-2022-grp1/exec$ ./tnfs add test
--- INPUT
- ./blocks/bcf9e65b2e26c9b8efb05d717b247f679cb36283d87c7387b93a69772313fbd6.dat
-
--- OUTPUT
- bGAYTCMRSGAYTENDEGI2TEMRRGM4TAM3EGU3GIYJYMIYWCNRQME3TCYTFGE2WGZTBMFRTONJRMZTGG
ZTCGGYGMDEGQZWNTEGFSGKMTFGZRGGBX
--- INPUT
- ./blocks/79152212fe2670b3b3bd5c9869fe4b65810e82aa7fce497b1293e207d9b72c64.dat
-
--- OUTPUT
- bGAYTCMRSGAZWGMGLGHBTGENLFMU3GIMLGG5TDOYZRGU4DMMLGGFRTGNRXGNQWEZJSGFRWEMLDMQ3GK
MJQM02DQYLCHAYTQMBYMMZTMMJUMZSGGNZT
[21/05/2022-11:35:36][INF]New file test added to tnfs with CID : bGAYTCMRSGAZWGM
LGHBTGENLFMU3GIMLGG5TDOYZRGU4DMMLGGFRTGNRXGNQWEZJSGFRWEMLDMQ3GKMJQM02DQYLCHAYTM
BYMMZTMMJUMZSGGNZT
```

FIGURE 4.3 – Ajout du fichier test

Le contenu du fichier est le suivant : On met ensuite en route le serveur en exécutant ./tnfs-server.



The screenshot shows a file viewer window with a title bar containing 'test' and the path '~/Projets/ppii-tnfs-2022-grp1/exec'. The file content is displayed in a monospaced font:

```
1 je suis le fichier test
2 disponible sur la machine 1
```

FIGURE 4.4 – Ajout du fichier test

On récupère l'adresse IP de la machine en exécutant 'ip addr' :

```
lorisdev@lorisdevm:~/Projets/ppii-tnfs-2022-grp1/exec$ ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 08:00:27:7c:60:50 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.47/24 brd 192.168.1.255 scope global dynamic noprefixroute enp0s3
        valid_lft 86232sec preferred_lft 86232sec
    inet6 2a01:cb11:404:b400:4a47:af2e:a20c:6ed0/64 scope global temporary dynamic
    inet6 2a01:cb11:404:b400:efe4:249e:95bb:375a/64 scope global dynamic mngtmpa
    inet6 fe80::bbe4:f9c:fcc9:36a4/64 scope link noprefixroute
    valid_lft 86358sec preferred_lft 558sec
    valid_lft forever preferred_lft forever
```

FIGURE 4.5 – Affichage de ip addr

On voit ici que l'adresse IP de la machine possédant le fichier est 192.168.1.47. Sur la seconde machine, on exécute donc un `./tnfs peer add 192.168.1.47 :5000`. On peut ensuite exécuter la commande `get` pour retrouver notre fichier sur le réseau :

```
lorisdev@lorisdevm:~/Projets/ppii-tnfs-2022-grp1/exec$ ./tnfs get bGAYTCMRSGAZWG
MLGHBTGENLFMU3GIMLGG5TDOYZRGU4DMMLGGFRTGNRXGNQWEZJSGFRWEMLDMQ3GKMJQM2DQYLCHAYTQ
MBYMMZTMMJUMZSGGNZT
--- INPUT
- ./blocks/bGAYTCMRSGAZWGMLGHBTGENLFMU3GIMLGG5TDOYZRGU4DMMLGGFRTGNRXGNQWEZJSGFRW
EMLDMQ3GKMJQM2DQYLCHAYTQMBYMMZTMMJUMZSGGNZT.dat
-
--- OUTPUT
- bGAYTCMRSGAZWGMLGHBTGENLFMU3GIMLGG5TDOYZRGU4DMMLGGFRTGNRXGNQWEZJSGFRWEMLDMQ3GK
MJQM2DQYLCHAYTQMBYMMZTMMJUMZSGGNZT
--- INPUT
- ./blocks/bGAYTCMRSGAYTENDEGI2TEMRRGM4TAM3EGU3GIYJYMIYWCNRQME3TCYTFGE2WGZTBMFRT
ONJRMZTGGZTCGYGMMDEGQZWGNTGEGFSGKMTFGZRGGNBX.dat
-
--- OUTPUT
- bGAYTCMRSGAYTENDEGI2TEMRRGM4TAM3EGU3GIYJYMIYWCNRQME3TCYTFGE2WGZTBMFRTONJRMZTGG
ZTCGYGMMDEGQZWGNTGEGFSGKMTFGZRGGNBX
[23/05/2022-21:23:51][INF]File downloaded !
lorisdev@lorisdevm:~/Projets/ppii-tnfs-2022-grp1/exec$ cat test
je suis le fichier test
disponible sur la machine 1
```

FIGURE 4.6 – Obtention du fichier test et affichage

On retrouve bien le fichier que nous souhaitons. Nous avons testé avec différents types de fichiers tels que `.pdf`, `.mp4`, `.doc` ... et tous nos tests sont validés.

5 Gestion de projet

5.1 Répartition des tâches

Pour répartir nos tâches et suivre l'avancement de notre projet, nous avons choisi d'utiliser l'outil de gestion de projet Trello. Nous avons créé un tableau [disponible ici](#). Les tâches sont réparties dans cinq catégories : Backlog/To Do/Doing/Waiting For Review/Done.

Dès la phase de recherche, nous nous sommes spécialisés et avons réparti les tâches de la façon suivante :

- Loris : Partie réseau
- Hugo : Hachage de fichier / Logger
- Sangoan : DAG / Découpage en bloc / Réassemblage

Nous avons cependant tous été amenés à toucher aux parties ne nous concernant pas initialement pour travailler sur tous les aspects du projet.

5.2 Communication

Pour communiquer, nous avons essentiellement utilisé l'outil Discord. Il nous permet de discuter en temps réel, d'effectuer les réunions en visioconférence et de travailler en collaboration. Nous avons aussi mis en place un bot "GitLab Logs" pour pouvoir avoir un suivi de répertoire gitlab. Le bot nous affiche toute l'activité en temps réel :

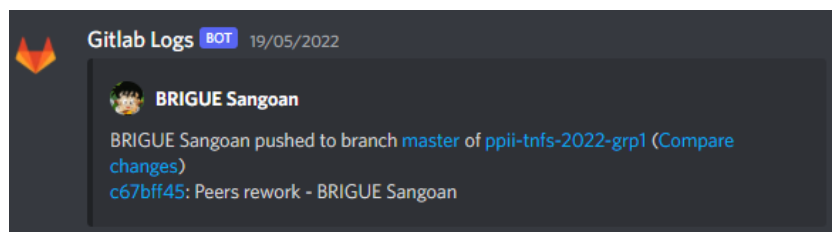


FIGURE 5.1 – Exemple des logs du bot Gitlab

5.3 Gestion des sources

Pour gérer le code source, nous utilisons le répertoire gitlab fourni par l'école. Nous avons décidé de garder une seule branche car nous n'avons jamais travaillé sur deux sujets différents simultanément au cours du projet. Le code est donc [disponible ici](#). La procédure d'installation de TNFS est disponible dans le readme du projet.

6 Conclusion

Afin de créer notre propre version de TNFS, nous avons commencé par étudier le principe de fonctionnement d'IPFS. Nous en avons retenu les 3 principes clés : l'identification du contenu, le découpage du contenu et l'accès au contenu via un réseau décentralisé. Nous avons par la suite implémenté ces concepts. La hachage du fichier permet d'attribuer un identifiant unique pour chaque contenu. Il est ensuite divisé en plusieurs blocs de données. Enfin, on peut le retrouver sur le réseau grâce à la table de hachage.

Nous avons décidé d'implémenter un algorithme de recherche du contenu sur le réseau différent de celui utilisé par IPFS. Cela nous a permis de tester les principales fonctionnalités de TNFS en utilisant notamment plusieurs machines virtuelles sur un réseau local, puis sur un réseau global. Enfin, nous avons abordé la gestion de projet et les différents moyens de communication mis en place pour assurer son bon déroulement. Il serait intéressant dans le futur d'implémenter complètement Kademlia pour reproduire exactement le fonctionnement d'IPFS.

Bibliographie / Webographie

- [1] Peter Maymounkov et David Mazières. Kademlia : A peer-to-peer information system. <https://pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia-lncs.pdf>, 2022. Dernier accès le 06 Mars 2022.
- [2] IPFS. Ipfs documentation. <https://docs.ipfs.io/>, 2022. Dernier accès le 06 Mars 2022.
- [3] Jakob Jenkov. P2p network algorithms. <https://jenkov.com/tutorials/p2p/index.html#p2p-network-algorithms>, 2022. Dernier accès le 06 Mars 2022.
- [4] Multiformats. Répertoire github multicodec. <https://github.com/multiformats/multicodec/blob/master/table.csv>, 13 Novembre 2016. Dernière modification le 19 Mai 2022. 8
- [5] Multiformats. Répertoire github multibase. <https://github.com/multiformats/multibase/blob/master/multibase.csv>, 22 Août 2016. Dernière modification le 22 Avril 2021. 8
- [6] Eleuth P2P. Kademlia algorithm presentation. <https://youtu.be/7o0pfKDq9KE>, 2022. Dernier accès le 06 Mars 2022.

Liste des illustrations

2.1	Exemple de Merkle DAG	4
2.2	Arbre binaire Kademlia	5
2.3	Réseau de 16 pairs	6
3.1	Diagramme du processus de décodage du CID	8
3.2	Échanges entre le client et le serveur d'un pair	15
3.3	Algorithme général d'une pair	16
4.1	Deux machines virtuelles	17
4.2	Deux machines virtuelles	17
4.3	Ajout du fichier test	18
4.4	Ajout du fichier test	18
4.5	Affichage de ip addr	18
4.6	Obtention du fichier test et affichage	19
5.1	Exemple des logs du bot Gitlab	20

Listings

3.1	Extrait de code de la fonction de composition du CID	9
3.2	Structure d'un bloc	10
3.3	Découpage d'un bloc	11
3.4	Lecture d'un bloc	12
3.5	Structure d'un pair	13
3.6	Définition de get_all_peers	13
3.7	Définition de create_socket	14
3.8	Définition de close_socket	14
3.9	Définition de send_tcp	14

Acronymes

CID Content Identifier

CLI Command Line Interface

DAG Directed Acyclic Graph

DHT Distributed Hash Table

IPFS InterPlanetary File System

IPNS InterPlanetary Name System

P2P Pair-à-pair (peer-to-peer)

SGBD Système de gestion de base de données

TCP Transmission Control Protocol