Université Paris Cité

LPM — PGF
Projet

Jean-Baptiste Yunès
Jean-Baptiste.Yunes@u-pariscite.fr

3 novembre 2026

---

1 Introduction et Contexte

Ce projet consiste à concevoir et développer une application de partage de fichiers décentralisée reposant sur un réseau pair-à-pair.

L'objectif principal est d'exploiter les primitives natives du langage Go (goroutines, canaux, synchronisation, entrées/sorties réseau non-bloquantes) afin de construire un système capable de gérer des communications directes (point-à-point) et diffusées (broadcast).

2 Consignes

Imprégnez-vous du sujet en le lisant attentivement pour bien en comprendre les détails. En particulier sur ce que doivent maintenir comme structures les nœuds et les accès concurrents. Le sujet contient de nombreux problèmes à résoudre, à vous de jouer.

L'interface utilisateur n'est pas précisée, mais le mode texte suffira (Go ne propose pas d'interface graphique donc laissez tomber).

Il est conseillé de réaliser ce projet en groupe de 3 ou 4 personnes (ni plus ni moins) car il y a beaucoup de travail de réflexion, de codage et de test.

Vous pouvez vous aider d'une intelligence artificielle à condition de ne pas lui demander de produire le code à partir du sujet. Toute utilisation d'une intelligence artificielle devra être signalée et l'intégralité de l'interaction avec elle devra être jointe au code.

Le code doit être produit dans un dépôt git (le gitlab de l'UFR). La progression de votre travail doit apparaître dans les logs des commits. Il ne s'agit donc pas de produire une version finale poussée dans le git en toute fin. Je dois pouvoir examiner l'avancement de votre travail. Le code doit être commenté (vous pouvez vous aider d'une IA pour commenter le code...).

Vous êtes en M2, donc du niveau ingénieur, il vous est donc demandé de produire un code « professionnel » avec tout ce que cela implique (documentation, manuel d'utilisation, etc). Passez au niveau supérieur de « projet étudiant pour une UE ». Le code doit être correctement structuré, c'est impératif ; pas de code monolithique spaghetti. La structure doit être apparente. Je vous encourage à faire de la revue de code entre les membres du groupe mais aussi entre groupes. La revue de code est un impératif du monde professionnel... !

Vous pouvez commencer comme vous le voulez mais une progression raisonnable pourrait ressembler à :

1. Réseau élémentaire : Création de deux nœuds, connexions TCP, échange de messages JSON.
2. Découverte et propagation : Insertion de nouveaux nœuds, tables de pairs, diffusion des requêtes, détection des doublons.
3. Indexation et recherche : Indexation des fichiers locaux, calcul des empreintes et recherche distribuée.
4. Transfert des fragments : Découpage, recherche des propriétaires, téléchargement et vérification des fragments.
5. Distribution et concurrence : Téléchargement parallèle, réplication opportuniste et gestion des déconnexions.
6. Robustesse et qualité : Tests de charge, tests de pannes.

Les 4 premières étapes constituent un socle fonctionnel minimal déjà bien conséquent. Certaines parties fonctionnelles doivent pouvoir faire l'objet de tests unitaires, d'intégration, etc (découpage, empreintes, vérification, échange de messages, etc).

3 Fonctionnalités Attendues

3.1 Démarrage

Le démarrage consiste en la création d'un nœud initial. Le réseau est alors réduit à ce seul nœud.

3.2 Insertion d'un nœud

Lorsqu'un nœud souhaite s'insérer dans le réseau, il se connecte à un nœud de référence. Cette connexion permet aux deux nœuds de se connaître et de s'ajouter mutuellement dans leur table de nœuds connus.

Le nœud nouvellement inséré dans le réseau envoie alors à son nœud de référence le message `GET_PEERS` qui permet au nœud de référence de lui envoyer sa table des nœuds connus (ou un morceau de sa table) via un message `PEER_LIST`. Suite à cette réponse, le nœud nouvellement créé initie une connexion permanente aux nœuds de sa propre table de nœuds connus.

3.3 Découpage et intégrité des fichiers

Les fichiers possédés par un nœud seront contenus dans un répertoire donné. Un fichier est identifié dans le système par le hachage de son contenu (algorithme SHA-256).

Afin de permettre un échange modulaire et d'éviter de saturer les connexions avec des flux trop lourds, les fichiers ne sont pas transférés d'un seul bloc, mais par fragments de taille fixe.

1. **Découpage :** Tout fichier mis à disposition est divisé en blocs de taille fixe (par exemple 256 Ko). Le dernier bloc peut être de taille inférieure si la taille totale du fichier n'est pas un multiple exact.
2. **Indexation :** Chaque fragment se voit attribuer un identifiant séquentiel unique (un index entier allant de 0 à $N-1$ si N fragments).
3. **Empreintes cryptographiques :** Pour garantir l'intégrité des données sans faire confiance aux pairs distants, une empreinte `SHA-256` est calculée pour chaque bloc individuel. Une table de hachage associant chaque index de fragment à son hachage est ainsi générée pour le fichier.
4. **Vérification de l'intégrité :** À la réception d'un bloc par le biais d'une connexion TCP directe, le client calcule immédiatement son hachage `SHA-256` et le compare à la référence attendue. Si l'empreinte correspond, le bloc est validé et écrit à sa position sur le disque ; dans le cas contraire, il est rejeté et immédiatement redemandé.

3.4 Recherche d'un fichier

Un nœud qui cherche un fichier envoie une requête `SEARCH_FILE` en indiquant un nom de fichier. Cette requête est transmise récursivement à tous les nœuds du réseau (en évitant les boucles).

Tout nœud qui possède un fichier de ce nom renvoie une réponse `HAVE_FILE` contenant le hachage global du fichier. Le nœud qui recherche le fichier obtient donc une liste de hachages, il lui incombe donc de faire un choix entre ces différents hachages (deux hachages différents représentent deux fichiers de même nom mais de contenus différents) avant de demander un transfert.

3.5 Transfert d'un fichier

Une fois le fichier choisi, le nœud envoie à l'un des pairs qui possède ce fichier une requête `GET_HASHES` qui lui permet d'obtenir la liste des hachages des fragments du fichier.

Une fois ces hachages obtenus, il envoie des requêtes `WANT_HASH` récursivement à tous les nœuds du réseau. Un nœud qui possède le fragment le renvoie avec une réponse de type `HAVE_HASH`. Le nœud qui a initié le transfert reçoit alors possiblement plusieurs réponses (plusieurs nœuds peuvent posséder le fragment). On opère alors un transfert direct avec l'un de ces nœuds via `GET_HASH` qui répond par `SEND_HASH`. Le nœud reçoit le contenu du fragment et vérifie qu'il est correct. Si ce n'est pas le cas (corruption) il essaie avec un autre nœud qui a déclaré posséder le fragment.

Une fois le fragment reçu, il le redistribue à l'un de ses voisins directs de façon aléatoire (de sorte que le contenu d'un fichier soit distribué aléatoirement dans le réseau). Une fois tous les fragments obtenus il reconstitue le fichier d'origine.

4 Format des requêtes

Les requêtes sont toutes au format JSON.

```
{
  "type": "GET_PEERS"
}
```

```
{
  "type": "PEER_LIST",
  "peers": [
    { "ip": "192.168.1.15", "port": 4001 },
    { "ip": "192.168.1.22", "port": 4001 }
  ]
}
```

```
{
  "type": "SEARCH_FILE",
  "msg_id": "id-unique",
  "filename": "toto.jpg",
  "origin_id": "ip:port"
}
```

```
{
  "type": "HAVE_FILE",
  "filename": "toto.jpg",
  "file_hash": "e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855",
  "origin_id": "ip:port"
}
```

```
{
  "type": "GET_HASHES",
  "file_hash": "e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855"
}
```

```
{
  "type": "FILE_HASHES",
```

```
  "file_hash": "e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855",
  "chunks": 3,
  "chunk_hashes": [
    "hash_chunk_0_hex...",
    "hash_chunk_1_hex...",
    "hash_chunk_2_hex..."
  ]
}
```

```
{
  "type": "WANT_HASH",
  "msg_id": "id-unique",
  "chunk_hash": "hash_chunk_1_hex...",
  "origin_id": "ip:port"
}
```

```
{
  "type": "HAVE_HASH",
  "chunk_hash": "hash_chunk_1_hex...",
  "origin_id": "ip:port"
}
```

```
{
  "type": "GET_HASH",
  "chunk_hash": "hash_chunk_1_hex..."
}
```

```
{
  "type": "SEND_HASH",
  "chunk_hash": "hash_chunk_1_hex...",
  "data": "SGVsbG8gV29ybGQgLS0gQ2h1bmsgZGF0YQ=="
}
```

Note : le champ `data` est l'encodage Base64 des données du fragment.

Les erreurs auront le format :

```
{
  "type": "ERROR",
  "message": "Do not have the requested hash"
}
```