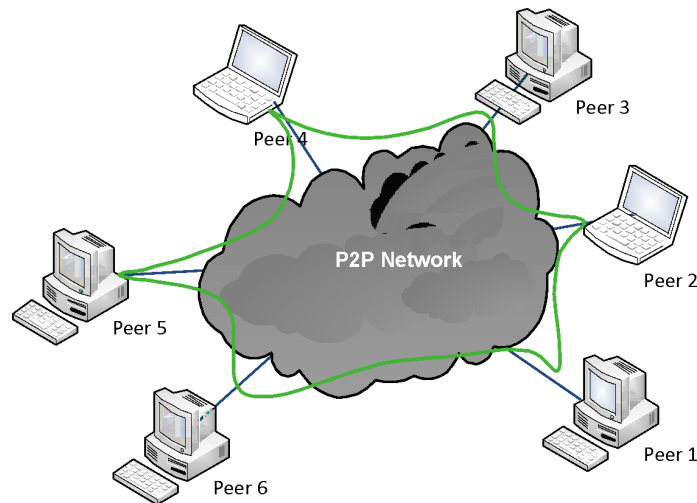


# Application d'échange de Fichiers en Pair à Pair (FileShare)



## Description du projet :

Le but du projet est de développer une application pour le partage de fichiers en mode pair à pair. Un réseau pair-à-pair (ou P2P : *Peer-to-Peer*) est constitué d'un ensemble d'ordinateurs (processus) connectés entre eux. Pour le partage de fichier en P2P, un pair est à la fois serveur et client des autres pairs. Dans un tel réseau on peut identifier deux types de pairs :

- (1) Les fournisseurs (ou « *seeders* ») : sont des pairs qui ont la totalité du fichier et le partagent avec les autres pairs.
- (2) Les consommateurs (ou « *leechers* ») : sont des pairs qui sont en train de télécharger le fichier ou des parties du fichier.

Ainsi, les deux types de pairs peuvent contribuer : les *seeders* par la totalité du fichier et les *leechers* par la(les) partie(s) qu'ils détiennent (ce comportement est voulu dans certaines applications P2P comme BitTorrent afin de décharger les *seeders*, peu nombreux, et faire contribuer les *leechers* dès le début du téléchargement).

## Travail demandé :

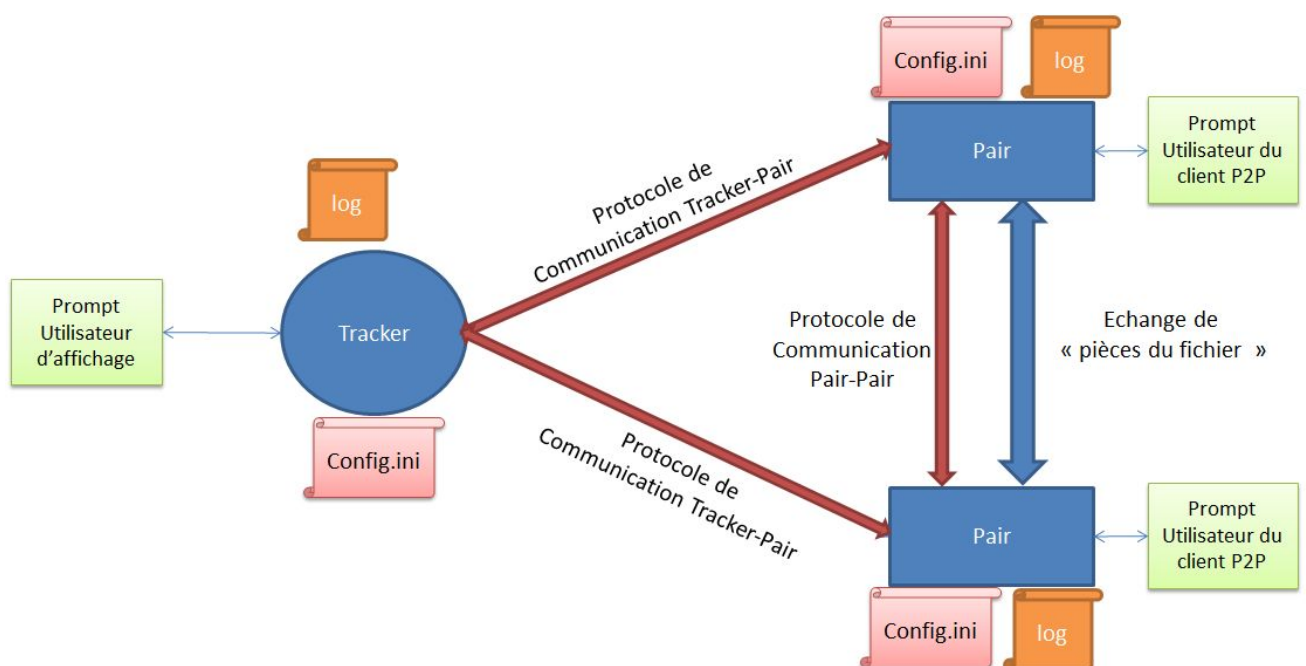
Ce sujet décrit uniquement le protocole que votre programme devra implémenter. Le but final est que tous vos programmes puissent communiquer entre eux :

- Les communications entre pairs se feront en TCP.
- Les messages échangés entre les différents pairs seront sous la forme de texte là où c'est applicable (tous sera codé en texte sauf les données binaires).
- Le caractère « blanc » servira comme séparateur entre les différents champs d'un message. Une liste de champs sera mise entre "[" et "]" .
- Pour l'échange des données, les pairs s'échangent des « pièces » du fichier (i.e. des parties de fichier). Un fichier est décomposé en pièces de données de taille égale. Cette taille, fait partie des informations décrivant le fichier.

## 1. Version Centralisée :

Dans la version dite centralisée, une entité centrale (Tracker) sera implémentée conjointement avec les programmes clients. Le Tracker assiste les différents pairs dans la recherche des fichiers à télécharger, mais aussi dans la découverte des autres pairs. Le Tracker est connu de tous les pairs.

L'interaction entre les programmes est illustrée dans la figure suivante dans la version centralisée :



L'utilisateur **DOIT** pouvoir :

- Spécifier le port d'écoute du Tracker à son lancement.
- Spécifier les informations de connexion du Tracker au Client P2P en ligne de commande ou dans un fichier de configuration.
- Le symbole "<" indique la commande envoyée et le symbole ">" la réponse reçue.
- Le Hash est calculé par MD5 (voir md5sum)

### Protocole de Communications Pair- Tracker :

- 1) Au début, chaque pair commence par annoncer sa présence au Tracker ainsi que la liste des fichiers qu'il possède. De cette manière, le Tracker a une vue globale des différents fichiers présents dans le réseau.

```
< announce listen $Port seed [$Filename1 $Length1  
$PieceSize1 $Key1 $Filename2 $Length2 $PieceSize2 $Key2  
...] leech [$Key3 $Key4 ...]  
> ok
```

L'utilisateur **DOIT** pouvoir spécifier le port d'écoute utilisé par le pair au lancement de l'application ou dans un fichier de configuration.

Une option serait de choisir un port disponible d'une manière automatique au démarrage de l'application.

Exemple:

```
< announce listen 2222 seed [file_a.dat 2097152 1024  
8905e92afeb80fc7722ec89eb0bf0966 file_b.dat 3145728 1536  
330a57722ec8b0bf09669a2b35f88e9e]  
> ok
```

Ici le pair annonce au Tracker, qu'il est en écoute sur le port 2222 et qu'il possède deux fichiers :

- « file\_a.dat » dont la taille est de 2097152 octets (2Mo), découpé en pièce de 1024 octets (1Ko), et ayant comme clé: 8905e92afeb80fc7722ec89eb0bf0966.
- « file\_b.dat » dont la taille est de 3145728 octets (3Mo), découpé en pièce de 1536 octets (1,5Ko), et ayant comme clé:  
330a57722ec8b0bf09669a2b35f88e9e.

- 2) Un pair peut à tout moment demander au Tracker la liste des fichiers présents dans le réseau vérifiant un certain nombre de critères eux-mêmes transmis en paramètre.

```
< look [$Criterion1 $Criterion2 ...]
```

```
> list [$Filename1 $Length1 $PieceSize1 $Key1 $Filename2  
$Length2 $PieceSize2 $Key2 ...]
```

*Le critère d'égalité de nom de fichier **DOIT** être implémenté, d'autres critères peuvent être implémentés en option.*

Exemple:

```
< look [filename="file_a.dat" filesize>"1048576"]
```

```
> list [file_a.dat 2097152 1024  
8905e92afeb80fc7722ec89eb0bf0966]
```

Le pair demande au Tracker la liste de fichiers dont le nom est «file\_a.dat» et dont la taille est supérieure à 1048576 octets (1Mo).

Le Tracker répond avec une liste qui ne contient qu'un seul fichier satisfaisant les critères.

- 3) Si le pair est intéressé par un certain fichier, il pourra demander son téléchargement au Tracker. Le Tracker devra alors fournir au client la liste des pairs possédant la totalité ou une partie du fichier en question ainsi que les informations nécessaires pour se connecter à ces pairs ci.

```
< getfile $Key
```

```
> peers $Key [$IP1:$Port1 $IP2:$Port2 ...]
```

Exemple:

```
< getfile 8905e92afeb80fc7722ec89eb0bf0966
```

```
> peers 8905e92afeb80fc7722ec89eb0bf0966 [1.1.1.2:2222  
1.1.1.3:3333]
```

Le pair demande au Tracker la liste des pairs partageant le fichier dont la clé est 8905e92afeb80fc7722ec89eb0bf0966.

Le Tracker répond avec une liste des pairs ainsi que leurs adresses IP et numéros de Port. (1.1.1.2 :2222 et 1.1.1.3 :3333)

## Protocole de Communications Pair- Pair :

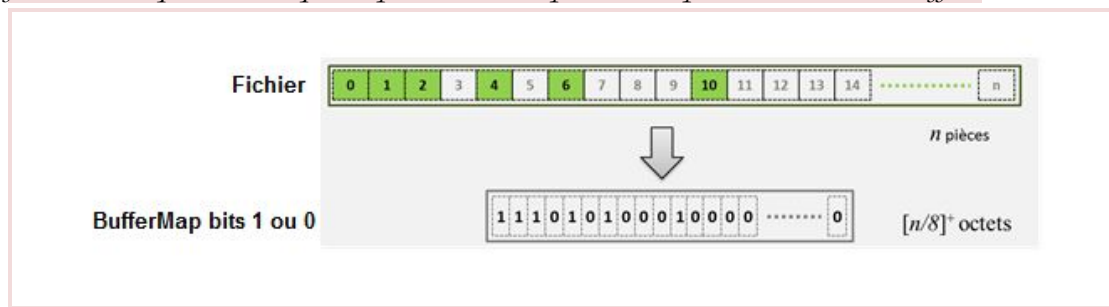
- 4) Après avoir obtenu la liste des pairs auxquels se connecter, le pair initiera une connexion avec chacun de ces pairs leur indiquant son intérêt pour le fichier désiré. Les pairs devront alors répondre avec des informations (BufferMap) concernant les parties du fichier qu'ils ont en leur possession.

*Le nombre de pairs maximum auxquels se connecter **DOIT** être paramétrable et stocké dans un fichier de configuration (par exemple, 5 pairs par défaut)*

```
< interested $Key
```

```
> have $Key $BufferMap
```

*Le buffermap est une séquence de bits dont le nombre correspond au nombre des pièces du fichier. Chaque bit indique la présence de la pièce correspondante dans le buffer.*



Exemple:

```
< interested 8905e92afeb80fc7722ec89eb0bf0966
```

```
> have 8905e92afeb80fc7722ec89eb0bf0966 %buffermap%
```

Le pair informe son voisin qu'il est intéressé par le fichier dont la clé est 8905e92afeb80fc7722ec89eb0bf0966.

Le voisin lui répond avec son buffermap (une séquence d'octets en binaire). Comme le fichier fait 2Mo et est découpé en pièces de 1Ko, on aura 2048 pièces. Pour représenter la présence de ces pièces dans le buffermap, on aura besoin de 2048bits et donc  $[2048/8] = 256$  octets.

- 5) Le pair demande ensuite les différentes pièces du fichier selon leur disponibilité au niveau des autres pairs.

```
< getpieces $Key [$Index1 $Index2 $Index3 ...]
```

```
> data $Key [$Index1:$Piece1 $Index2:$Piece2  
$Index3:$Piece3 ...]
```

L'utilisateur **DOIT** pouvoir configurer (dans un fichier de configuration) la taille des messages qu'il veut recevoir en limitant le nombre de pièces demandées.

Par exemple, si un fichier est découpé en pièces de 2Ko, et la taille maximum des messages est de 16Ko, le pair va demander au plus 8 pièces à la fois.

Exemple:

```
< getpieces 8905e92afeb80fc7722ec89eb0bf0966 [3 5 7 8 9]  
> data 8905e92afeb80fc7722ec89eb0bf0966 [3:%piece3%  
5:%piece5% 7:%piece7% 8:%piece8% 9:%piece9%]
```

Le pair demande à son voisin les pièces 3, 5, 7, 8, 9 du fichier dont la clé est 8905e92afeb80fc7722ec89eb0bf0966.

Le voisin lui répond avec les pièces demandées. Si le fichier est découpé en pièces de 1024 octets, chaque **%piece%** dans le message précédent est une séquence de 1024 octets binaires.

### *Considération dans la mise en place des deux protocoles*

- 6) Périodiquement, le pair informe ses voisins de l'état d'avancement du téléchargement en envoyant son nouveau buffermap. Ses voisins répondent par leur dernier buffermap. Ainsi un pair peut découvrir la disponibilité de nouvelles pièces dans son voisinage.

```
< have $Key $BufferMap
```

```
> have $Key $BufferMap
```

Les intervalles de mise à jour entre pairs et avec le Tracker **DOIT** être paramétrable et stocké dans un fichier de configuration.

Exemple:

```
< have 8905e92afeb80fc7722ec89eb0bf0966 %new_buffermap%  
> have 8905e92afeb80fc7722ec89eb0bf0966 %new_buffermap%
```

Les pairs échangent périodiquement leurs buffermap afin de faire connaître les pièces téléchargées depuis le dernier échange.

Le pair informe périodiquement le Tracker des différents fichiers disponibles ou en cours de téléchargement.

```
< update seed [$Key1 $Key2 $Key3 ...] leech [$Key10 $Key11  
$Key12 ...]  
> ok
```

Exemple:

```
< update seed [] leech [8905e92afeb80fc7722ec89eb0bf0966]  
> ok
```

Le pair informe le Tracker qu'il n'a pas de fichiers à fournir, mais qu'il est en train de télécharger le fichier dont la clé est 8905e92afeb80fc7722ec89eb0bf0966.

## 2. Version distribuée (optionnelle):

*Il est **obligatoire** d'avoir totalement implémenté la version centralisée avant de passer à la version distribuée.*

Dans la version distribuée, il n'y aura pas de Tracker. Un pair découvre et intègre un réseau existant en établissant des connexions avec les pairs qui font déjà partie du réseau P2P.

Les recherches sont simplifiées et ne concernent que le nom exacte d'un fichier (ex. *criterion: filename="setup\_maj.exe"*).

Un pair doit faire une découverte réseau afin d'identifier les autres pairs. Pour ce faire, on suppose que l'application finale va s'exécuter dans un réseau local. Le pair peut se permettre alors de diffuser son message de découverte du voisinage (neighbourhood) en broadcast ou en multicast UDP (classe d'adresse D). Dans ce message (UDP) il indique le nom de l'application et sa version.

*Le port UDP utilisé est le même pour tous les pairs. L'utilisateur **DOIT** pouvoir spécifier ce numéro au lancement de l'application ou dans un fichier de configuration.*

Un pair exécutant la même application, et supportant la même version va répondre avec un message UDP (sur le même port) en confirmant le nom de l'application, la version, ainsi que les informations pour établir une connexion en TCP.

*Le nombre de réponse à considérer après un message de découverte **DOIT** être paramétrable.*

```
< neighbourhood "FileShare" $Version  
> neighbour "FileShare" $Version $Port
```

Exemple:

```
< neighbourhood "FileShare" 1  
> neighbour "FileShare" 1 3333
```

Le pair diffuse un message de découverte du voisinage en indiquant le nom de l'application « FileShare » et le numéro de version supporté 1. Un pair qui a reçu ce message répond par le message « neighbour » en indiquant le nom de l'application, la version ainsi que le port d'écoute TCP (ici = 3333).

Pour la suite, la même logique que dans la version centralisée sera utilisée (moyennant quelques modifications) avec en plus quelques commandes supplémentaires :

- 1) On n'annonce plus les fichiers qu'on a mais plutôt les informations de connexions. Les pairs voisins répondent par « ok ».

```
<announce listen $Port  
> ok
```

Exemple:

```
< announce listen 4444  
> ok
```

Le pair établit une connexion TCP avec ce voisin fraîchement découvert et lui indique son port d'écoute TCP.

- 2) La requête de recherche (look) est transmise saut-par-saut à travers les pairs. Le nombre de sauts est limité à l'aide d'un paramètre TTL (Time-To-Live) de la requête.

*Le TTL **DOIT** être paramétrable et stocké dans le fichier de configuration.*

À la réception d'une requête, le pair commence par décrémenter le champ TTL de 1, et :

- Si le pair a le fichier, il répond directement au pair émetteur de la requête, éventuellement non voisin direct, après avoir établi une connexion avec lui si elle n'existe pas.

```
< look $Criterion $TTL $IP:$Port
```



*(si les deux pairs ne sont pas connectés directement, une connexion est établie)*

```
> file at $IP:$Port have $FileName $Length $PieceSize  
$Key seeders [$IP:$Port1]
```

- Si le pair est en train de télécharger le fichier, il répond avec 2 listes de pairs. La première concerne les pairs qui ont la totalité du fichier (seeders), la deuxième liste contient les pairs qui sont en train de télécharger le fichier (leechers).

```
< look $Criterion $TTL $IP:$Port
```

*(si les deux pairs ne sont pas connectés directement, une connexion est établie)*

```
> file at $IP:$Port have $FileName $Length $PieceSize  
$Key seeders [$IP1:$Port1 $IP2:$Port2 ...] leechers  
[$IP:$Port2 $IP4:$Port4 ...]
```

- Si le pair n'a aucune information sur le fichier recherché, il fait suivre la requête à ses voisins excepté le voisin émetteur.

```
< look $Criterion $TTL-1 $IP:$Port3
```

- Si le TTL est nul, le pair va notifier l'émetteur original de la requête par un message d'erreur (TTL=0).

```
< look $Criterion $TTL $IP:$Port
```

*(si les deux pairs ne sont pas connectés directement, une connexion est établie)*

```
> file at $IP:$Port
```

En général, une réponse à un message de recherché respecte la structure suivante:

```
> file at $IP:$Port have $FileName $Length $PieceSize  
$Key seeders [$IP1:Port1 $IP2:Port2 ...] leechers  
[$IP3:Port3 $IP4:Port4 ...]
```

L'absence d'un des champs have, seeders, leechers différencie les types de réponses comme décrit précédemment.

- 3) Le pair contacte les pairs qui ont la totalité ou une partie du fichier. Les pairs répondent par un BufferMap.

---

<sup>1</sup> Remarquer la même adresse spécifiée dans la liste des seeders, pour indiquer qu'on est soi-même seeder.

<sup>2</sup> Remarquer la même adresse spécifiée dans la liste des leechers, pour indiquer qu'on est aussi leecher.

<sup>3</sup> Ici, on fait suivre la requête avec l'adresse de l'émetteur original.

```
< interested $Key  
> have $Key $BufferMap
```

- 4) Le transfert des données se fait comme dans la version centralisée.

```
< getpieces $Key $Index1 $Index2 $Index3 ...  
> data $Key $Index1:$Piece1 $Index2:$Piece2  
$Index3:$Piece3 ...
```

- 5) Périodiquement, on informe les voisins sur l'état du téléchargement pour découvrir de nouvelles pièces

```
< have $Key $BufferMap  
> have $Key $BufferMap
```

### 3. Extension Blockchain (optionnelle):

*Il est obligatoire d'avoir totalement implémenté la version centralisée avant de réaliser l'extension blockchain.*

Le principe de base de la blockchain repose sur une base de donnée distribuée qui maintient une liste croissante d'enregistrement ordonné. Cette liste est censée être non falsifiable.

La plupart des projets populaire basés sur une technologie de blockchain existant étant particulièrement complexe. Notre approche se concentre sur l'essence de la blockchain en tant que simple registre distribué sous forme d'une chaîne de bloc dont le dernier bloc vérifie l'authenticité.

Dans le cadre de notre système de fichier, l'objectif est d'implémenter un mécanisme conservatif proche d'un tracker de torrent privé. Cet objectif devra être réalisé de manière distribué à l'aide d'une blockchain.

Chaque peer du réseau recevra un ratio basé sur la quantité de block envoyé et la quantité de block reçu. Ces informations seront stockées dans la blockchain.

Lorsque le ratio d'un peer diminue sous un seuil prédéfini 0.5, par exemple, selon une formule Upload/Download, les autres peers devront le limiter en téléchargement jusqu'à que son ratio redépasse le seuil.

Afin d'éviter qu'un peer s'auto-valide des blocs envoyés, il est également nécessaire que seul les peers dont l'identification est présente dans la chaîne puissent y écrire.

- 1) Imaginer une solution afin d'implémenter cette extension et apporter les modifications nécessaires aux différents protocoles.

## Fichier de « Log »

Tous les programmes doivent contrôler le degré de verbosité en affichant des informations dans un fichier log ou une console de log.

## Configuration des programmes (config.ini)

Chaque programme devra s'appuyer sur un fichier de configuration disponible dans le même dossier que l'exécutable avant le lancement du programme.

Exemple du format du fichier de configuration (config.ini):

```
# Adresse IP du tracker
tracker-address = a.b.c.d

# Numéro de port TCP d'écoute du tracker
tracker-port = 12345
```

## Organisation du projet

- Les élèves doivent s'organiser en groupes de **5** élèves au maximum.
- La programmation se fera en langages C (tracker) **ET** Java (peer)
- Une pré-démo et un rapport intermédiaire sont à prévoir pour la **6<sup>ème</sup>** séance.
- Une méthodologie de gestion de projet doit être mise en place par le groupe d'élèves. A tout moment, un audit sera demandé (définition de l'architecture générale du projet, découpage du projet en tâches, planification des tâches, affectation des tâches à des responsables, indicateurs de suivi de réalisation des tâches, mise en œuvre de tests unitaires / intégration, ...)

- La soutenance finale est organisée sous forme d'une démonstration et de discussions autour du projet. Le livrable final doit être envoyé à la fin de la 9<sup>ème</sup> séance. La soutenance aura lieu à la dernière séance du projet (10<sup>ème</sup> séance). L'ordre du passage sera indiqué par l'encadrant du projet.

### Remarques :

- Utilisation de thor pour la gestion des versions (git ou svn).
- précision : le nom du fichier annoncé ne doit pas comporter le caractère espace “ “
- précision : les commandes du protocoles se terminent toutes par un “\r\n” ou “\n”. Cela permet d'identifier implicitement la fin d'une commande du protocole.
- Le but de la clé est d'identifier de manière unique un fichier. Pour ce faire, on utilisera la fonction de hachage MD5 (voir md5sum)
- Le programme doit supporter le téléchargement de plusieurs fichiers en parallèle.
- L'utilisateur doit avoir connaissance de ce qui se passe : Progression des différents téléchargements, liste des pairs connectés, taux de transfert avec les différents pairs, le débit de téléchargement global (*download rate*), le débit de contribution globale (*upload rate*).
- Préférer une interface utilisateur simple en ligne de commande
- Le programme doit tolérer le départ d'un voisin (panne, déconnexion, ...)
- Le programme doit ignorer les commandes non implémentées d'une manière silencieuse sans altérer le fonctionnement de base.
- La taille des pièces doit être paramétrable et stockée dans le fichier de configuration. **On utilisera une valeur par défaut, connue de tous les pairs (2048 octets)**
- « **Message Framing** » : Comme TCP est orienté flux : une lecture sur la socket ne retourne pas forcément un message complet tel qu'il est transmis par le machine distante. Cependant, comme les messages échangés sont bien définis, il faut lire et identifier les messages au fur et à mesure de leur réception.

Si on considère le 2<sup>ème</sup> message dans l'échange suivant :

```
< interested 8905e92afeb80fc7722ec89eb0bf0966  
> have 8905e92afeb80fc7722ec89eb0bf0966 %buffermap%
```

À la lecture du mot “have”, on sait que le prochain token doit forcément être une chaîne représentant la clé du fichier et ceci jusqu’au prochain « espace ». Ensuite, connaissant la taille du fichier en question et la taille des pièces, on peut déterminer la longueur de la séquence du buffermap à lire (ex : fichier de 3Mo, découpé en pièces de 2Ko => 1536 pièces => 1538/8=192 octets à lire pour le buffermap). Fin du message.

## Conseil d'organisation: utilisation des mock

Afin de pouvoir répartir le travail efficacement au sein du groupe et également de pouvoir tester les modules de votre code, il peut être intéressant de mettre en place la technique des faussaires. Ceci pourra être fait à minima avec la partie réseau afin de ne pas être dépendant de cette partie pour tester le reste de votre projet. En java, cela consiste à utiliser les interfaces pour spécifier l'API entre le module et le reste du code et ensuite à fournir deux implémentations: une faussaire et une réelle. Le choix de l'implémentation à utiliser peut reposer par exemple sur: une option fournie au programme, une variable d'environnement, etc. L'utilisation du pattern Singleton simplifie normalement cela.

En C, le remplacement d'un module par son faussaire peut reposer tout simplement par la substitution de .o au moment de la compilation ou encore en utilisant les bibliothèques dynamiques et en jouant sur la variable LD\_LIBRARY\_PATH.

## Pool de Threads:

Le création de thread coûte cher aussi il est important de maîtriser cet aspect dans votre projet. A titre d'exemple, il n'est pas judicieux de créer un thread par requête entrante. Une technique très connue consiste à lancer à l'avance un nombre de thread qui vont ensuite aller chercher du travail dans une file, ainsi: on maîtrise le nombre de thread créé et on recycle ceux-ci une fois qu'un travail est fait. Un autre avantage est pour le debuggage: en créant un groupe de thread de taille 1, on

simplifie le debuggage du programme. Pour plus d'information, vous pouvez commencer :

- [https://en.wikipedia.org/wiki/Thread\\_pool](https://en.wikipedia.org/wiki/Thread_pool)
- <https://docs.oracle.com/javase/tutorial/essential/concurrency/pools.html>
- <https://github.com/Pithikos/C-Thread-Pool>

## **Mises à jour du protocole**

Consulter régulièrement cette page, des modifications seront publiées si nécessaire :

- **[26/02/2020] Publication du sujet.**