Rapport de mi-projet

Projet de Réseaux

BARBARIN Paul COMITI Santinu DUCAMP Simon
MORENO CARPIO Kenzo MOREL Mathieu

S8 — 2023-2024

Table des matières

1 Le tracker

1.1 Principe général et architechture

Le tracker a la mission centrale de **fédérer** et de **coordonner** les différents pairs qui peuvent s'y connecter. Il s'agit d'un serveur à part entière : Sa mission est de traiter des **requêtes** et de renvoyer des **réponses**.

Afin de développer ce programme, nous avons fait le choix d'une architechture multi-thread en C. Ce choix paraît indispensable. En effet, une forme de parallélisme est nécéssaire : Le serveur doit pouvoir communiquer avec plusieurs pairs en parallèle, tout en continuant d'écouter les nouvelles connexions.

Nous avons donc $n + 1$ threads qui sont créés simultanément lors du démarrage de notre programme, n éxécutent la fonction `specificListener`, et un thread central éxécutant `newClientListener`.

Ces deux fonctions permettent respectivement de traiter la connexion entre un pair donné et d'écouter les connexions des nouveaux pairs. Nous reparlerons de ces deux fonctions plus en détails dans la section suivante.

1.2 Les informations stockées par le tracker

Le tracker maintient et met à jour trois principales structures de données, toutes représentées par des listes chaînées :

- **Les clients en attente de connexion** : Lorsqu'un nouveau pair se connecte au serveur, il est inséré dans cette structure de données comme pair en attente de traitement. Les threads éxécutant la fonction `specificListener` ne traitant pas déja une connexion interrogent en permanence cette structure pour traiter les clients les uns après les autres.
- **Les pairs identifiés** : Lorsqu'un pair envoie une requête `announce`, il est automatiquement ajouté à cette liste chaînée.
- **Les fichiers présents sur le réseau** : Une liste chaînée est également chargée de stocker les différents fichiers présents sur le réseau. Elle est mise à jour lors de requêtes comme `announce` ou `update`.

1.3 Le tracker : Avant tout un parseur

Le serveur tracker est avant tout un parseur de commandes reçues via le réseau. Une importante partie du code est donc consacrée au parsing. C'est le cas du code présent dans les fichiers `announce.c`, `look.c`, `getfile.c` et `upddate.c`. Tout au long du parsing, ces fichiers font appel à des fonctions dans les fichiers `peer.c` et `fileAvailableList.c`, fonctions permettant la manipulation des listes de pairs et de fichiers.

Afin de permettre une analyse lexicale et sémantique de nos requêtes, nous itérons sur celles-ci en utilisant le caractère espace comme séparateur.

Globalement, nous avons remarqué que parser des chaînes de caractères est assez difficile en C. En effet, il aurait été plus simple de réaliser cette tâche en utilisant un langage de plus haut niveau ne nécéssitant pas une allocation pour chaque chaîne. Typiquement, le langage Javascript. Nous aurions également pu utiliser des outils spécifiques à l'analyse syntaxique tels que **lex** et **yacc**.

2 Le client

Le client est la partie du projet qui implémente le comportement du pair défini dans le protocole. Nous l'avons programmé en Java et avons réussi à implémenter l'échange de fichier en soi, cependant il nous manque un certain nombre de mécanismes demandés par le sujet.

Dans cette section, nous détaillerons les difficultés auxquelles nous avons fait face et nos décisions d'ingéniérie dans le développement du client.

2.1 Architecture globale

L'architecture globale de notre application prend cette forme :

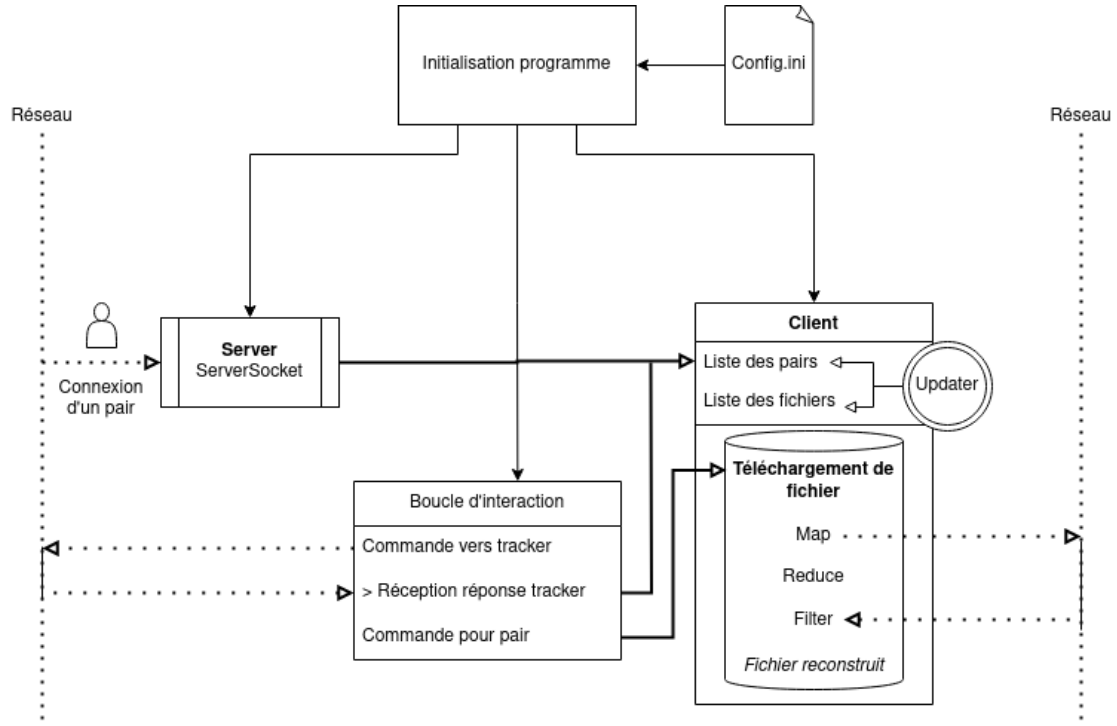

FIGURE 1 – Diagramme des composants de notre client

Chaque composant peut être détaillé et possède des fonctionnalités qui n'apparaissent pas sur le schéma. En particulier, des classes liées à la programmation réseau (voir section 2.2) ou à la programmation multithreadée (section 2.4) sont détaillées plus bas dans ce rapport.

2.2 La classe Connect et les sockets

Nous avons écrit une classe Connect pour wrapper l'utilisation des sockets. Comme une socket, on veut pouvoir écrire et recevoir un message dessus : ce sont les deux méthodes principales.

Pour simplifier l'utilisation de la Connect, on munit la classe d'un système inspiré du modèle Producer/Consumer. Le Producer génère des données de manière asynchrone, stockées dans une file, et le Consumer les consomme de manière asynchrone. Dans notre cas, on souhaite munir notre Connect de deux files :

— une destinée à l'envoi de message, dont le Consumer est la socket, et le Producer l'utilisateur (via la méthode sendMessage)
— une destinée à la réception de message, dont le Producer est la socket, et le Consumer l'utilisateur (via la méthode receiveMessage)

Pour permettre ce fonctionnement asynchrone, les Producer et Consumer seront des Runnable, donc lançable dans des threads. Connect possède donc deux threads en interne qui servent à faire la communication entre les méthodes de l'utilisateur et la socket. Une conséquence de ce fonctionnement est que l'appel receiveMessage est bloquant dans le cas où la file des messages reçus est vide ; c'est le comportement souhaité, semblable à l'opération de lecture d'une socket qui est elle aussi bloquante.

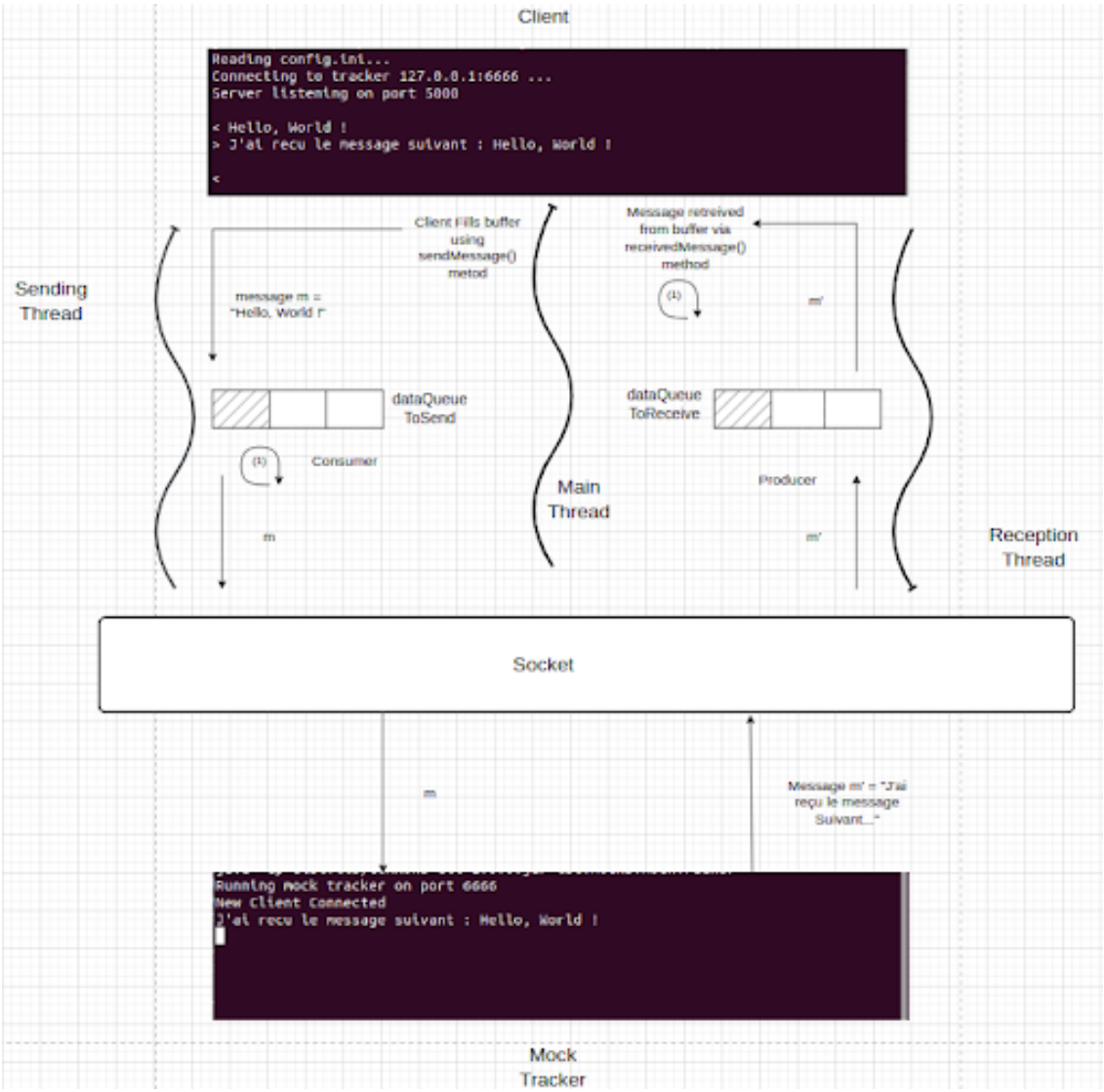

FIGURE 2 – Schéma représetant le fonctionnement de la classe Connect

2.3 Le parsing

Le parseur est la partie du code permettant d'analyser les chaînes de caractères envoyées au client afin de vérifier leur logique (analyse lexicales et syntaxique). Entre autres, le parseur va s'occuper des communications entre le client avec le tracker et les autres pairs en convertissant les messages du réseau en objets Java compréhensible par le client, et inversement.

Nous avons décidé dans notre projet de distribuer la responsabilité de parsing entre plusieurs classes : on peut transfomer un message du réseau en *CommandParsed*, qui s'occupe dans son constructeur de faire une découpe du message en token.

Ensuite, chaque *CommandParsed* peut être transformée en "objet commande". On utilise pour cela le patron de conception du **foncteur** : chaque "objet commande" aura sa propre classe (classes *Announce*, *GetFile*, *Interested*, *Have*, etc) qui implémentera l'interface *ICommand*, qui décrit une unique méthode `execute`. On appelle ces objets foncteurs car ce sont des classes qui se comportent comme des fonctions, en effet, leur utilité unique est de lancer la commande associée sur une connexion passée en paramètre via `execute`. Elles s'occupent aussi de faire les analyses lexicales et syntaxiques au sein de leur constructeur, et de parser les informations issues du paramètre *CommandeParsed* en structures de données définies pour le projet.

Nous wrappons la création de commandes à la *CommandFactory*. Les commandes étant issues d'entrées utilisateur ou entrées réseau, il nous est impossible de faire de la vérification statique (à la compilation) de leurs types. Nous déléguons donc cette vérification à des vérifications dynamiques (mot-clef `instanceof`) éparpillées un peu partout dans le code.

2.4 Programmation multithreadée

Ce projet a été l'occasion d'utiliser énormément les threads, que ce soit pour des simples coroutines ou dans des architectures plus complexes. Le projet a été très formateur sur ce point, puisque les threads étaient au centre des fonctionnalités les plus importantes du projet.

2.4.1 L'envoi de fichier via Map-Filter-Reduce

Lorsque l'on regarde les 2 premières commandes du protocole destinée à des pairs : interested et getpiece, on remarque que la seconde dépend des réponses à la première, et que chacune de ces commandes ne s'envoie non pas à un seul pair mais à l'ensemble des pairs d'un client donné.

On peut donc en déduire un pipeline logique pour l'échange des données de fichiers basé sur un modèle Map-Filter-Reduce :

- Map : on envoie "interested" à chaque pair, et on récupère leurs buffermaps
- Filter : on filtre parmis les buffermaps récupérés pour déduire les pièces de fichier à demander et à quel pair les demander
- Reduce : on envoie "getpiece" à chacun de ces pairs filtrés, et on écrit dans le fichier les pièces récupérées

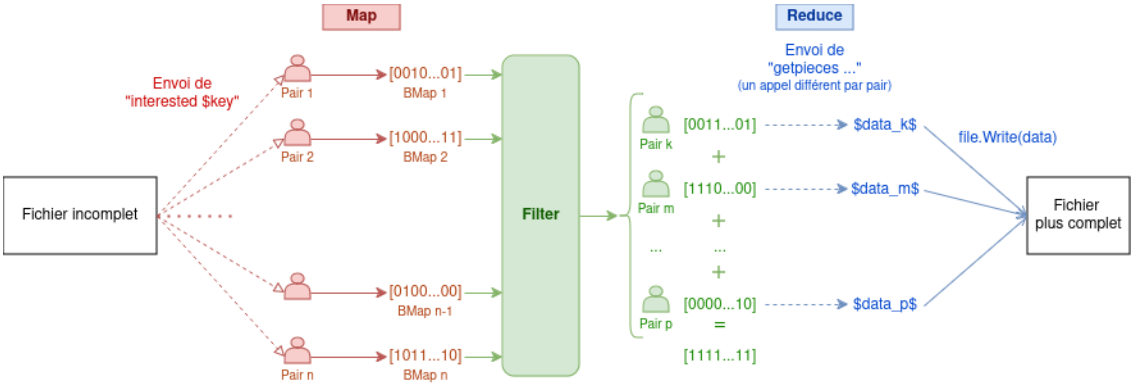

FIGURE 3 – Schéma représetant le Map-Filter-Reduce

Cette logique est stockée dans les classes implémentant l'interface Client, dont par exemple ClientStandard On remarque qu'on peut paralléliser les appels réseaux dans la partie "Map" et dans la partie "Reduce". En effet, supposons qu'un pair en particulier soit long à répondre, il est pertinent de mettre cet appel en attente pour continuer à exécuter d'autres appels. Les opérations de map et de reduce ne sont concrètement que des itérations sur des listes, que l'on peut paralléliser facilement en lançant un Thread qui fait une opération parallèle par tour de boucle. C'est pourquoi nous avons créé une classe MultiThreadOperators, qui implémente les boucles for et forEach en version multithreadée pour effectuer des appels parallels, que nous avons utilisé pour notre algorithme de Map-Filter-Reduce.

2.4.2 Opérations en arrière-plan

Un autre aspect du projet est la présence d'opérations en arrière-plan, implémentée via des threads.

Les deux exemples les plus importants de ces opérations d'arrière-plan sont :

- La récupération régulière des connexion de pairs, et l'écoute de leurs messages entrants
- L'envoi régulier des commandes *update* et *have* sur le réseau

Par pression de temps, nous n'avons pas eu le temps de bien implémenter ce dernier point.

Pour l'écoute des autres pairs, nous avons créé un système où une classe statique, *Server*, écoute en permanence sur un thread l'arrivée de nouveaux pairs grâce à l'objet Java *ServerSocket*, et dès qu'une tentative de connexion est détectée, elle est acceptée et lance un autre thread qui écoute en permanence la réception de messages de la part de ce pair. Si cet autre pair réclame un buffermap de fichier avec *interested* ou des données de fichier avec *getfile*, alors on crée les ICommand au statut spécial *OnInterested* ou *OnGetFile* qui renvoie la réponse adéquate au pair.

2.5 Génie logiciel et implémentations

La programmation orientée objet offerte par Java permet une séparation claire des interfaces et des implémentations, ainsi que la possibilité d'une extension rapide et facile du code.

Nous avons exploité cette fonctionnalité dans la création de la classe Client, qui gère la logique de l'échange pair à pair. L'interface Client définit le comportement général de cet échange pair à pair, la classe abstraite.

ClientStandard définit l'algorithme de Map-Filter-Reduce de base que nous utilisons, et la classe concrète ClientStdSmallBmaps définit les fonctions précises de Map, de Reduce, ainsi que l'algorithme de Filter. Ce dernier point est important, car la fonction Filter décrit l'algorithme de sélection des pairs à leech qui est au coeur du protocole d'échange pair à pair. La création de la classe abstraite ClientStandard permet donc de créer plusieurs variations interchengables de cet algorithme.

Point négatif de notre projet : on se retrouve avec un projet client trop gros, avec du code parfois superflu et des redondances d'informations. Cela s'explique par une architecture qui est mal pensée vis-à-vis des besoins du projet, qui est la résultante de notre manque d'expérience en programmation multithreadée et programmation réseau.

En particulier, notre projet présente certaines faiblesses d'architecture qui seraient facilement réglables et faciliteraient la lecture ou les extensions de code. C'est le cas par exemple de la classe *Server* qui aurait pu être mieux intégrée avec la classe *Client*, ou encore les dépendances circulaires avec *Client* : *Client* possède un *Updater* qui crée des hooks *OnGetFile* ou *OnInterested* qui ont besoin d'avoir accès au client. On peut finalement citer une surabondance de constructeurs dans certaines structures de données, qui auraient pu être remplacés par un patron de conception du *builder*, ou encore des violations des principes objets comme des setter/getter dans le *Client*.

3 Le protocole réseau

Le protocole proposé par le sujet présentait un certain nombre de trous ou ambigüités que nous avions la responsabilité d'interpréter et de remplir.

Nous détaillons dans cette section nos décisions d'ingéniérie quant à ces problématiques qui ont structuré l'échange de données sur le réseau et le comportement de chaque acteur sur ce même réseau.

3.1 Représentation des données

Certaines structures de données décrites par le protocole n'ont pas de représentation fixe dans le réseau : les "pieces" de fichier, et les "buffermap".

Nous avons décidé pour les "pieces" de se baser sur le langage ayant permis de développer le client : Java. En effet, Java dispose d'un type de données permettant de représenter des octets (le type byte pour un octet, le type byte[] pour une suite d'octets). Ce type de données est idéal pour lire/écrire des fichiers qui peuvent concrètement être décrits comme une suite d'octets. Quant à l'échange sur le réseau, Java dispose d'une conversion bytes[]/String bijective qui permet de transformer les données de fichiers en chaînes de caractères échangeables sur une socket (si le fichier est initialement un fichier texte, le string d'arrivée est la suite de caractères lues). Ainsi, une "piece" est une String qui provient d'un byte[] et qui pourra être efficacement être retransformé dans le même byte[] par le client d'arrivée.

Cependant, avec cette solution naïve, on se retrouve face à des erreurs d'envoi sur le réseau et au niveau du découpage des *piece*. Cette première erreur intervient si le message contient un '\n', qui termine la bufferisation du message sur la socket, et coupe donc le message en deux envois. On choisit donc d'encoder un '\n' dans chaque *piece* par un '\\n', et de décoder chaque '\\n' en '\n'. D'une manière analogue, si une *piece* contient une suite de caractères de la forme "blabla 5 :texte_quelconque 9 :suite", alors le réseau interprètera le message comme d'autres *pieces* envoyées, ce qui peut mener à des bugs. On choisit donc d'encapsuler chaque *piece* par des guillemets ("). On remplace donc chaque guillemet interne par un encodage particulier comme \'\'

Cette approche, même si plus avancée que l'approche naïve, présente un désavantage : si le contenu présentait déjà les caractères encodés (\'\' ou \\n), alors ces caractères seront remplacés par leur décodage, l'opération d'encodage puis décodage n'est donc **pas bijective**. Cela peut mener dans certains cas à la corruption de fichier, puisque les données écrites à la réception sont plus petites que les données envoyées, donc l'écriture des données mèneraient à du 0-padding dans le fichier.

Pour éviter ces problèmes, on aurait pu choisir d'encoder les informations du fichier en hexadécimal plutôt qu'en UTF-8. Cela éviterait tous les problèmes de caractères interdits ou de violation de bijection, mais chaque octet nous coûterait 2 caractères pour être encodé (un octet vaut 8 bits, et un hexadécimal vaut 4 bits), donc chaque message pour transférer des données sur le réseau serait 2 fois plus lourd.

Secondairement, et dans la même logique, nous avons décidé de représenter les buffermaps sous la forme de chaînes de caractères en hexadécimal sur le réseau. En effet, les buffermaps peuvent se retrouver longs pour des gros fichiers comme des vidéos, donc l'envoi sur le réseau en binaire semble être une mauvaise idée. Cependant, il est important de noter que les buffermaps sont équivalents non pas à des nombres binaires, mais à des **mots binaires** (011 et 11 sont des buffermaps différents). Or, la conversion de mots binaires en hexadécimal n'est pas triviale : les représentations de 011 et 11 sont ambigües : 3. C'est pourquoi nous ajoutons à la fin de la représentation hexadécimale un masque de taille sous la forme d'un slash

'/' et d'un nombre nombre hexadécimal de la taille du buffermap.

3.2 Clé de fichier

Nous nous sommes heurtés à des problématiques quant à l'identification et l'unicité des fichiers : à partir de quelle information du fichier sa clé est-elle générée, son nom ou son contenu ? Que se passe-t-il si deux clients essaient d'envoyer deux fichiers différents mais de même clé, ou deux fichiers identiques de clé différente ? Y-a-t'il un porteur originel de fichier qui ferait le rôle de propriétaire au sein du réseau ? Ensuite, si un client arrive à regénérer un fichier en entier, en devient-il un nouveau propriétaire aux yeux du réseau ?

Pour la première problématique, il semble plus judicieux de générer la clé selon le contenu, puisqu'un fichier se caractérise avant tout par son contenu, plutôt que son nom. Cependant, pour des raisons de simplicité, et en gardant à l'esprit que c'est une solution sous-optimale, nous avons décidé de générer les clés de fichier en fonction de leur nom.

Pour les autres problématiques, il semble plus difficile de trancher et d'élire une solution au dessus des autres. C'est pourquoi nous nous sommes accordés sur le comportement suivant :

- Celui qui possède le fichier initialement le pousse sur le réseau, et en devient le owner, ou "source of trust"
- Le tracker l'élit comme owner du fichier, et caractérise ce fichier par la clé unique qu'il a envoyé (issue donc de son nom)
- Tout autre pair qui essaie d'envoyer un fichier de même clé se voit rejeté.
- Tout pair qui essaie de télécharger le fichier doit le faire par clé (les criterions ne permettant que de retrouver des clés)
- Tout pair qui a reconstruit le fichier en entier peut devenir un nouvel owner du fichier (ce point là n'a pas été creusé plus en profondeur)

3.3 Logging et gestion des erreurs

Des parties importantes dans la création d'un protocole réseau et d'une application client/serveur sont

- la robustesse de chaque acteur
- la résilience du réseau
- la trace des erreurs

Le premier point est la capacité de chaque application (qu'elle soit le client ou le serveur) à générer, intercepter et interpréter des erreurs sans entrer dans un état incohérent voire pire : crash.

Dans le cadre de notre client par exemple, développé en Java, c'est le mécanisme de `try/catch` qui est prépondérant pour cette tâche. Il permet de garder une trace

d'exécution des erreurs et de continuer à exécuter le programme sans le faire planter. Nous avons essayé de créer notre propre message d'erreur pour ce projet : *BadReponseException* afin d'intercepter les erreurs liées au protocole.

Pour notre tracker, développé cette fois-ci en C, il n'existe pas de mécanisme simple à utilier comme Java. Le mécanisme en C qui y ressemble le plus, c'est les signaux. Nous avons donc implémenté un gestionnaire de signaux qui interprète `SIGINT` ou `SIGPIPE`. On aurait pu imaginer désigner un comportement par signal et créer un gestionnaire exhaustif qui gère chaque signal individuellement sans faire crash l'application.

Le second point est la capacité de chaque application ou du réseau tout entier de rester en marche dans le cas d'une déconnexion soudaine d'un ou plusieurs acteurs. Il s'agit de mettre en place des mécanismes de *timeout* pour ne pas attendre une réponse dans le vide, ou encode de ping régulièrement les autres pairs, de mettre à jour les informations sur le réseau et les fichiers disponibles en cas de déconnexion avérée, etc. Nous n'avons pas pu développer ce point comme nous le souhaitions, notamment le ping des voisins ou la mise à jour des informations sur le réseau, cependant nous avons pu mettre en place un mécanisme de *timeout* naïf pour le client, dans la classe *Connect*.

4 Gestion de projet

Dans cette section, nous nous intéresserons à certaines cméthodologies de travail mis en place au long du projet.

4.1 Tests et mocks

Pour tester les interactions client/serveur, il est très difficile de mettre en place des tests unitaires, ce que nous n'avons pas pu développer à cause de la pression en temps sur le projet. Nous avons donc reservé des tests unitaires à des fonctions critiques du code, comme la lecture/écriture dans des fichiers, ou encore la bonne conversion de structures de données en informations à envoyer sur le serveur (l'encodage/décodage de buffermaps, d'ip-ports, etc).

Afin de tester les interactions réseau, nous nous sommes concentrés sur des tests manuels des échanges client/serveur en utilisant des objets "Mock". C'est-à-dire des faux clients pour la partie serveur, ayant un comportement écrit à l'avance, ou un faux tracket pour la partie client, ayant aussi un comportement écrit à l'avance. C'est aussi de cette manière que l'on a pu tester les échanges pair à pair.

4.2 Utilisation des LLMs

Les LLMs nous ont été utiles dans toutes les étapes du développement. Nous avons principalement utilisé Chat GPT 4. Voici un détail de notre utilisation :

- Débogage et explication de code (notamment des portions de programmes trouvés sur internet)
- Création de tests unitaires, d'objets "mocks"
- Aide au développement de certaines fonctions communes comme le hachage MD5, ou des fonctions utilisant des expressions régulières
- Génération de code pour gérer les arguments en ligne de commande ainsi que les fichiers de configuration (config.ini)
- Ecriture et corrections du Makefile

Dans tous les cas d'usages, le plus fréquent est donc de se servir du LLM comme d'un moteur de recherche, plutôt que de chercher une solution sur internet. L'utilisation du LLM s'inscrit alors dans un processus de recherche documentaire au même titre que les documentations officielles de nos outils, ou encore les forums d'échange comme *StackOverflow*. Les LLMs présentent l'avantage de fournir une solution rapidement, de pouvoir modifier à volonté cette solution grâce à des prompts ou encore de demander au LLM d'expliquer son "raisonnement" pour nous éclairer ; il est aussi important de noter le point fort du LLM comparé à la documentation classique ou les forums : la capacité du LLM à donner une solution imprégnée d'un contexte.

Chacun de nos cas d'usages des LLMs ont globalement respecté plusieurs propriétés parmi les suivantes :

- Code non essentiel à l'architecture logicielle
- Portions de codes relativement courtes
- Portions de codes difficiles à réécrire
- Portions de code connues (où l'on sait donc que le LLM aura été entraîné sur un grand nombre de données)

Cependant, à mesure que le projet avance et se complexifie, les LLMs s'avèrent moins intéressants car les problématiques rencontrées impliquent un grand contexte difficile à leur soumettre.