Compte rendu projet INFO4B

Corentin DIEHL
Yoan DUSOEIL

Avril 2022

Sommaire

1 Analyse fonctionnelle

L'objectif est de traiter une base de données recensant toutes les parties d'échec du site https://lichess.org/. Chaque fichier contient toutes les parties d'un mois dans un format spécifique (*.pgn* qui est un format de codage pour les parties d'échecs). Le but est d'extraire les données de chaque fichier et d'y effectuer des recherches simples.

Un des objectifs principaux qu'on s'était fixé était de traiter l'entièreté de la base de donnée fournie par https://database.lichess.org/ ce qui posait donc notre premier problème auquel on devait faire face qui est l'énorme volume de donnée présent (environ 8 To de données). Dû à une limitation matérielle (disque dur), on faisait face dans un premier temps à la quasi impossibilité de multi-thread la lecture d'un fichier (la tête de lecture du disque dur ne pouvant être qu'à un endroit à la fois). De plus dans une lecture séquentielle du fichier, la vitesse d'extraction des données est de facto limitée par la vitesse de lecture étant donné que le programme lirait une partie puis en extrairait les informations avant de passer à la partie suivante.

Second problème et non pas des moindres, dû au très gros volume de données, peu importe le type de recherche souhaitée, le temps avant la production d'un quelconque résultat serait extrêmement long si on faisait nos recherches directement sur les fichiers *.pgn*.

2 Description structures de données

2.1 Un fichier dat par joueur (non retenu)

L'idée initiale était de créer un fichier par joueur contenant les noms des fichiers dans lesquels il apparaît ainsi que les octets de départ de chaque partie. Ainsi, il nous était possible de récupérer telle ou telle partie directement en ouvrant le fichier du joueur en récupérant l'octet de départ avec le nom du fichier associé.

2.2 Une classe Player contenant une ArrayList de toutes ses parties (non retenu)

Une des premières idées qu'on avait eu était de créer une classe Player contenant des références à toutes ses parties comme l'octet de départ dans le fichier pgn et d'enregistrer les données correspondantes. Cela aurait permis de charger les données de chaque joueur à partir d'un fichier .dat car cette solution allait avec la solution précédente. Cette solution n'aura finalement pas été retenue car trop coûteuse en ressources.

2.3 Une classe Game contenant les informations d'une partie

La classe Game constitue la structure des parties. Elle contient les informations suivantes : le type de la partie, son URL, le pseudo du joueur blanc et du joueur noir, le résultat, l'ouverture, la date, les mouvements et l'octet de départ. Elle contient plusieurs fonctions : une première permettant de créer et renvoyer un tableau de Game à partir d'un nom et fichier et d'un tableau d'octets de départ. Une seconde permettant de créer une Game à partir d'un nom de fichier et d'un octet de départ. Enfin, la fonction toString() permet, comme son nom l'indique, de créer et renvoyer une variable de type String contenant l'affichage d'une partie.

2.4 Une classe CommonRessources et une classe GameBuffer

La classe CommonRessources sert de ressource commune partagée entre tous les Threads consommateurs. A chaque partie traitée par un des Thread un compteur interne de la classe CommonRessources est incrémenté. A la fin de la lecture et une fois que le buffer est vide, toutes les Hashtable internes des Threads consommateurs vont vider leur contenu dans les Hashtables de la classe CommonRessources. La classe GameBuffer sert juste de buffer entre les threads producteurs et les threads consommateurs afin d'harmoniser le fonctionnement du programme.

2.5 Des fichiers .dat pour chaque fichier .pgn

Pour accélérer les requêtes clients, on a créé plusieurs fichiers pour chaque fichier source (les fichiers .pgn). Il y a tout d'abord un fichier .dat contenant toutes les URL de toutes les parties du fichier. Ensuite, il y a un fichier .dat contenant le nom des joueurs avec les octets de départ de toute leurs parties dans le fichier, il y a également leur nombre de parties. Enfin, un fichier .dat contenant le nom de l'ouverture ainsi que le nombre de fois que celle-ci apparaît dans le fichier. Tous ces fichiers permettent de réduire considérablement le temps de traitement des requêtes. Si la donnée que le client cherche existe, seulement à ce moment là le serveur ira chercher dans les fichiers .pgn la donnée que le client souhaite.

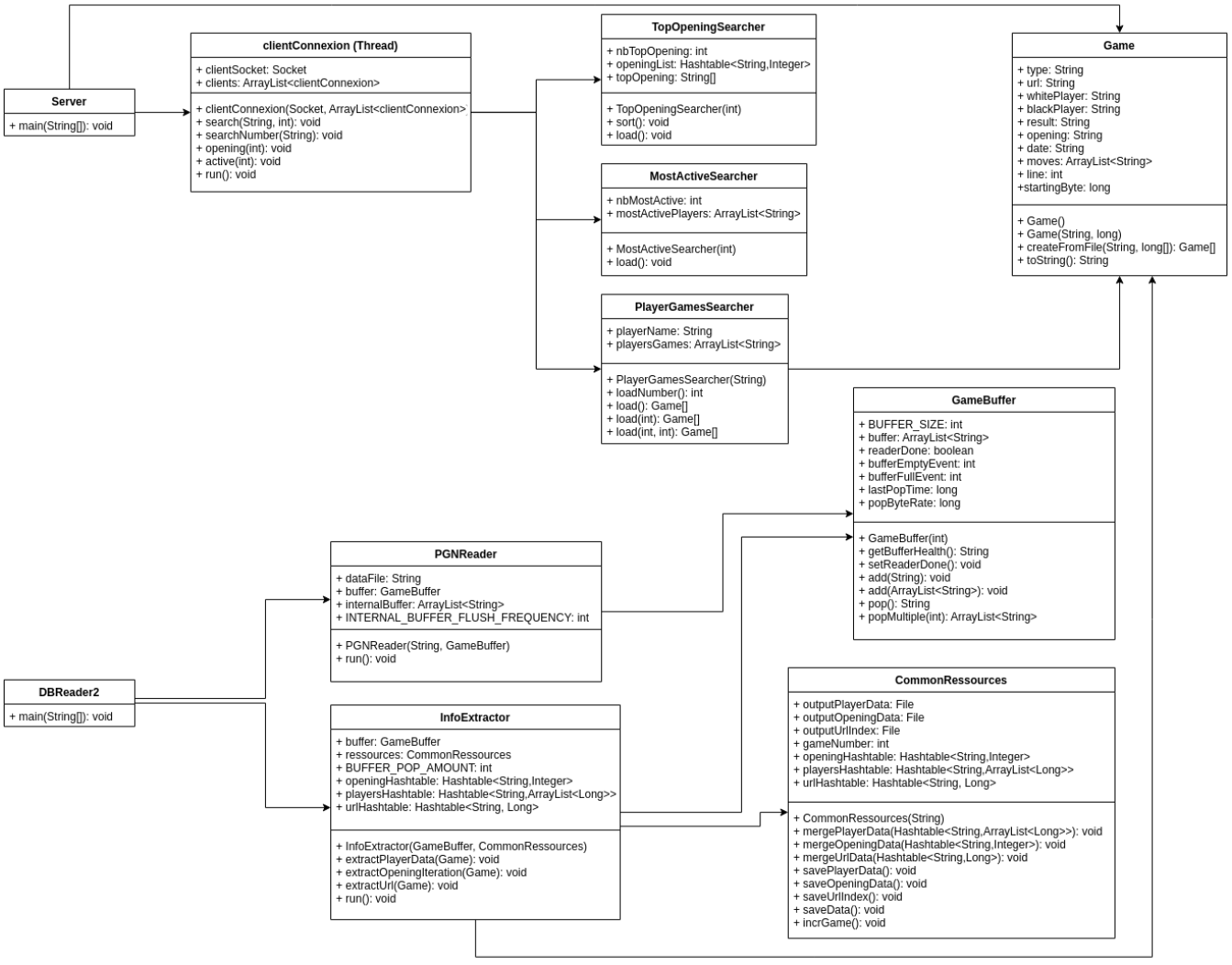

Figure 1: Diagramme des classes.

3 Spécificités des classes principales

3.1 Client

La classe client sert uniquement à envoyer les requêtes du client au serveur, et à ensuite recevoir les données renvoyées pour les afficher. La classe peut-être lancée avec pour argument une IP correspondant à l'IP du serveur. Si l'argument est vide l'IP par défaut est 127.0.0.1.

3.2 Server

La classe serveur s'occupe de toutes les requêtes clients. Le serveur écoute le client, puis effectue telle ou telle action en fonction. Elle est composée de plusieurs fonctions :

3.2.1 clientConnexion

- search()
La fonction search() prend 2 paramètres : le nom du joueur recherché de type String ainsi que le nombre de partie souhaité de type int. Une instance de PlayerGamesSearcher est créée en prenant en paramètre le pseudo du joueur recherché. La méthode load(String pseudo) est appliquée à l'instance de PlayerGamesSearcher créée, puis renvoie le nombre de parties souhaité pour le joueur en paramètre.
- searchNumber()
La méthode searchNumber() prend en paramètre le nom du joueur recherché de type String. Une instance de PlayerGamesSearcher est créée, et le paramètre nbGame qui représente le nombre de partie du joueur est modifié en appelant la méthode loadNumber(). Ensuite, si ce paramètre est positif (= que le joueur a des parties) et que le client souhaite afficher les parties, la fonction va appeler la méthode search() décrite précédemment.
- opening()
La fonction opening() crée une instance de TopOpeningSearcher. La méthode load() est appelée à cette instance. Ensuite, les ouvertures les plus actives demandées par le client sont renvoyées une par une au même client qui pourra les afficher.
- active()
La fonction active() crée une instance de MostActiveSearcher. Comme pour la fonction opening() la méthode load() est appelée à cette instance, les joueurs les plus actifs sont ensuite renvoyés au client qui pourra évidemment les afficher.
- run()
La fonction run() s'occupe de récupérer la demande du client via un Scanner, puis de lancer une des fonctions précédemment décrites :
- si le client tape "search" suivi du nom du jour, la fonction searchNumber() est appelé, et ce qui est décrit dans cette méthode s'en suit
- si le client tape "active" suivi d'un nombre, la fonction active() est appelée
- si le client tape "opening" suivi d'un nombre, la fonction opening() est appelée
- si le client tape "help", un fichier help.txt va s'afficher dans la console, et va, logiquement, aider le client en lui affichant les commandes disponibles
- si le client tape "exit", le client se ferme.

3.2.2 server

- main()
La fonction main() écoute continuellement pour une nouvelle connexion avec un client et, lorsqu'une connexion est accepté, créé une instance de clientConnexion et lui délègue le socket résultant de cette nouvelle connexion. Chaque nouveau client représente un nouveau thread.

3.3 DBReader2

Le programme DBReader2 s'occupe de créer tout les fichiers .dat. Pour se faire, il va utiliser un bon nombre de classes telles que PGNReader ou InfoExtractor. Cette classe permet de lire les fichiers .pgn en multithread. Ainsi, la vitesse globale de lecture des fichiers est d'environ 200Mo/s. Pour une base de donnée de 8To, cela représente environ 11h de traitement. Grâce à cette opération, le traitement des requêtes est quasiment instantané.

3.4 MostActiveSearcher

3.4.1 load()

Permet d'ajouter dans une ArrayList de Game les joueurs les plus actifs, le nombre souhaité est définie précédemment. Ces joueurs sont trouvés, dans l'odre, dans le fichier playersData.dat.

3.5 PlayerGameSearcher

3.5.1 loadNumber()

Charge le nombre de parties totale d'un joueur depuis le fichier PlayerData.dat afin d'avoir une indication du nombre de parties que le joueur peut avoir sans avoir à traiter toutes les parties du joueur.

3.5.2 load()

Charge toutes les parties d'un joueur et la retourne dans un tableau d'instances de Game.

3.5.3 load(int nbGame)

Charge un certain nombre de parties d'un joueur spécifié en paramètre de la méthode et le retourne dans un tableau d'instances de Game.

3.5.4 load(int fromGame, int toGame)

Charge d'une partie à une autre partie d'un joueur, spécifiées en paramètre de la méthode, et les retourne dans un tableau d'instances de Game.

3.6 TopOpeningSearcher

3.6.1 load()

De la même manière que la méthode load() de la classe MostActiveSearcher, celle de la classe TopOpeningSearcher modifie aussi un tableau de String, ces String étant les ouvertures les plus communes. Le nombre d'ouverture voulu est défini précédemment. Elles sont trouvées en parcourant tout les fichiers _opening_data.dat, le nom d'une ouverture est ajoutée à une

HashTable, si ce nom est déjà présent, la valeur est modifiée. Enfin, seulement le nombre souhaité est ajouté dans le tableau de String.

3.6.2 sort()

Permet de modifier l'ordre des ouvertures en triant le nombre d'ouverture dans l'ordre croissant. Cette fonction est appelée directement à la fin de la fonction load() définie précédemment. Ainsi, le tableau est trié.

4 Architecture globale détaillée

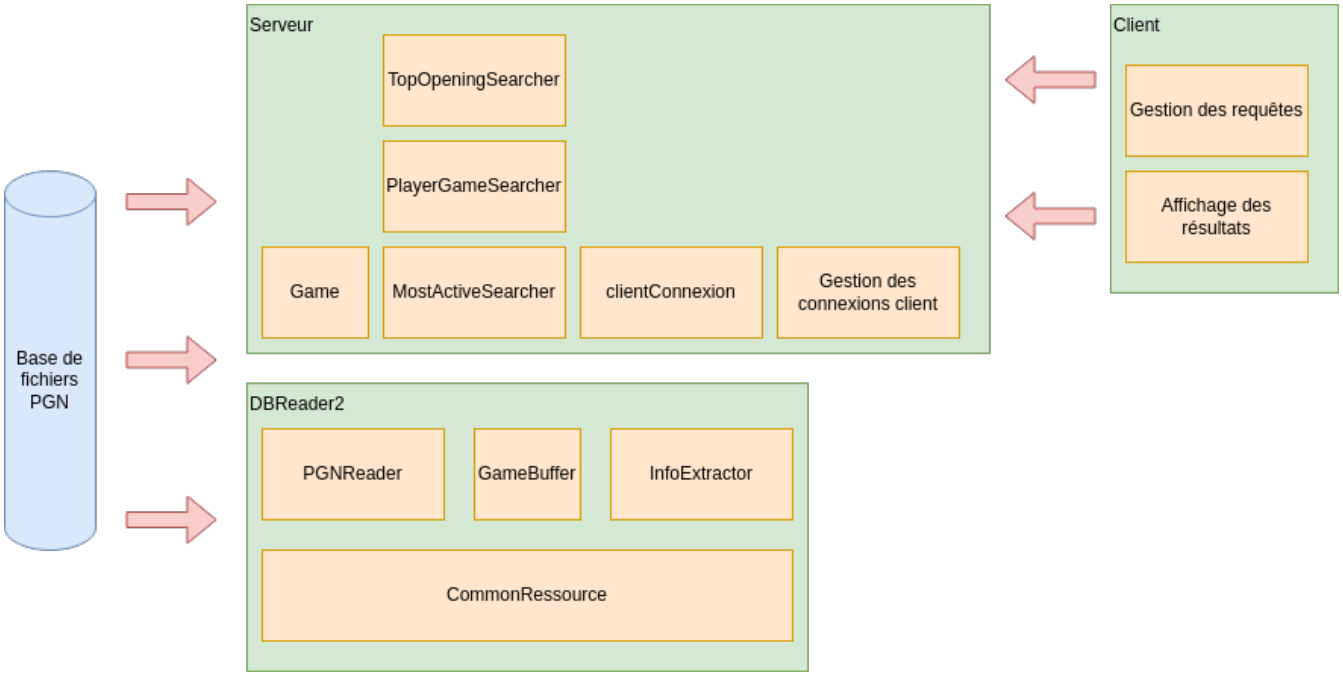

Figure 2: Conception en couches.

4.1 Client

4.1.1 Gestion des requêtes

La gestion des requêtes se fait par une écoute continue des commandes du client tant que le client n'entre pas *exit*. Si la commande entrée correspond à une requête le code correspondant à cette requête est exécuté. Parmi les requêtes possible on peut afficher un certain nombre ou toutes les parties d'un joueur, afficher les joueurs les plus actifs ou afficher les ouvertures les plus jouées.

4.1.2 Affichage des résultats

Après l'envoie d'une requête, le client se met en écoute d'une réponse du serveur et affiche la réponse du serveur. Le format de la réponse dépend donc du serveur.

4.2 Server

4.2.1 Gestion des connexions clients

Lors d'une tentative de connexion de la part d'un client, le socket résultant est délégué à une instance de clientConnexion.

4.2.2 clientConnexion

Thread qui gère la connexion avec un client et appelle la fonction correspondante avec la requête désirée. Ferme la connexion une fois la requête complétée.

4.2.3 TopOpeningSearcher

Classe dédiée à la recherche de l'ouverture la plus fréquente à partir des fichiers .dat créés par le DBReader2.

4.2.4 PlayerGameSearcher

Classe dédiée à la recherche des parties d'un joueur spécifique à partir des fichiers .dat créés par le DBReader2.

4.2.5 MostActiveSearcher

Classe dédiée à la recherche des joueurs les plus actifs à partir des fichiers .dat créés par le DBReader2.

4.2.6 Game

Classe servant de structure pour représenter les parties. Idéal pour le transport des informations concernant les parties entre les différentes classes.

4.3 DBReader2

4.3.1 PGNReader

Thread producteur. Cette classe lit tout un fichier .pgn, extrait les informations de chaque partie et envoie à une fréquence définie dans une constante interne (par défaut 10 000) au lancement du programme DBReader2 des parties dans le GameBuffer.

4.3.2 GameBuffer

Buffer servant d'intermédiaire entre le Thread producteur qu'est le PGNReader et les Threads consommateurs représentés par l'InfoExtractor.

4.3.3 InfoExtractor

Thread consommateur. Extrait les "pages (d'une taille définie en paramètre au lancement du programme)" du buffer pour en extraire les informations. Chaque instance de InfoExtractor contient des Hashtables internes qui contiennent les informations extraites.

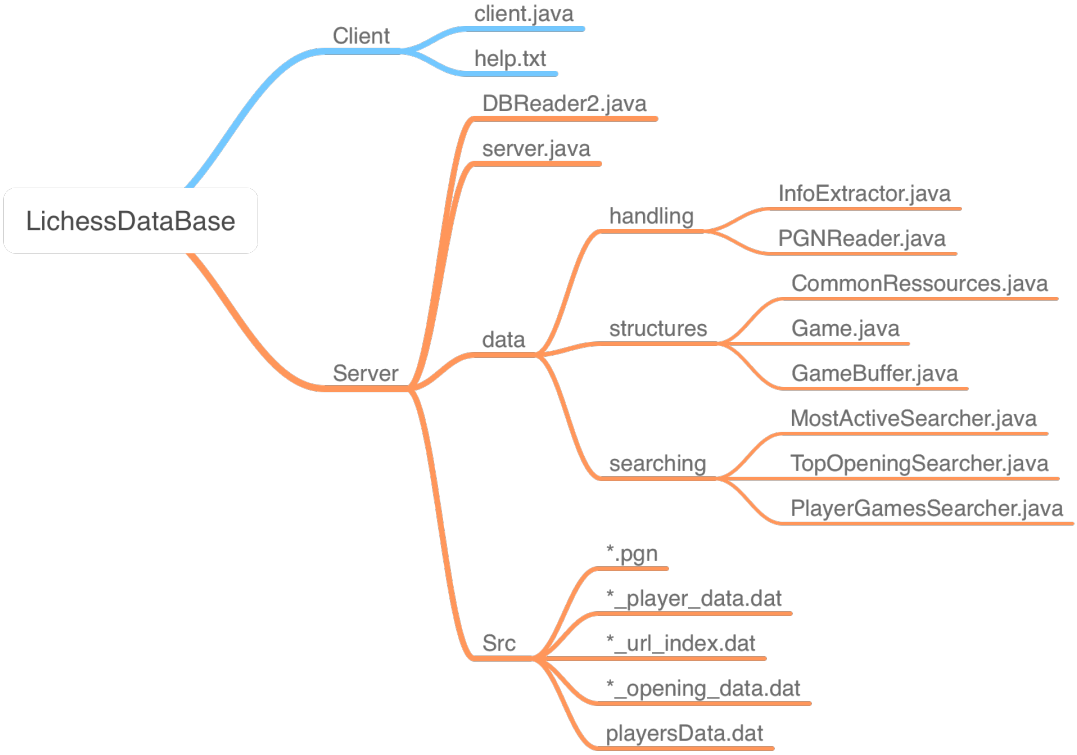

Figure 3: Arborescence du projet.

5 Description des algorithmes principaux

5.1 Extraction des parties d'un fichier .pgn

La classe PGNReader s'occupe d'extraire les parties d'un fichier .pgn. Étant donné que la longueur d'une partie peut-être variable, on a dû repérer un paterne récurrent afin de retrouver la fin d'une partie qui est le suivant :
- Une ligne blanche
- La liste des coups
- Une ligne blanche

On peut donc maintenant repérer la fin d'une partie juste en comptant le nombre de ligne blanches en lisant le fichier ligne par ligne. Si ce compteur vaut 2 alors une partie a été complètement lue et le compteur est réinitialisé. Chaque ligne lue est ajoutée à une String qui, une fois une partie lue, est envoyée dans le buffer interne de la classe PGNReader. Une fois un nombre de partie lue défini dans le constructeur, le buffer interne est vidé dans le GameBuffer. Ce système de buffer interne se révèle extrêmement utile afin de limiter les accès synchronisés au GameBuffer afin d'améliorer grandement la vitesse de lecture.

5.2 Extraction des informations d'une partie

La classe InfoExtractor est la classe qui intervient lors de l'extraction des informations d'une partie. Elle retire un nombre, défini dans le constructeur, de parties du GameBuffer afin de les stocker dans son buffer interne. Comme pour la classe PGNReader, ce système de buffer interne diminue les accès synchronisés au buffer et accélère grandement la vitesse de lecture. Chaque partie est alors lue ligne par ligne, interprétée grâce aux balises au début de chaque ligne et les informations sont placées dans une instance de Game puis stockées dans les hashtables interne de la classe InfoExtractor. Une fois le PGNReader fini, le GameBuffer vide et le buffer interne de InfoExtractor vide, les données des hashtables sont fusionnées dans une instance de CommonRessources.

5.3 Gestion des clients

Au démarrage du serveur, le Thread principal entre dans une boucle d'écoute infinie et se met en attente d'une connexion. Lorsque qu'un client tente de se connecter au serveur, la connexion est acceptée, une instance de clientConnexion est créé et le socket de la connexion lui est délégué. Le Thread clientConnexion se met alors en attente d'un message contenant la requête et traite cette requête en conséquence à l'aide de ses fonctions internes décrites précédemment.

5.4 Algorithmes de recherche

5.4.1 Recherche d'un joueur en particulier

La recherche d'un joueur spécifique est opéré par la classe PlayerGamesSearcher. La classe parcourt tous les fichiers *lichess_standard_rated_AAAA_MM_player_data.dat* à la recherche du nom du joueur recherché. Si ce nom est trouvé, la ligne suivante dans le fichier contient tous les octets de départ de ses parties dans le fichier .pgn. Ces indices sont alors stockés dans un tableau et les instances des parties correspondantes sont créées à l'aide de la fonction *createFromFile()* dans la classe Game qui prend en paramètre le nom du fichier et le tableau d'octets du début des parties dans le fichier .pgn.

5.4.2 Recherche des joueurs les plus actifs

Pour la recherche des joueurs les plus actifs, l'utilisation d'un fichier externe est requise : le fichier playersData.dat. Ce fichier est créé à la fin du DBReader2. Il parcourt tout les fichiers *lichess_standard_rated_AAAA_MM_player_data.dat* à la recherche des joueurs.
Lorsqu'un joueur est trouvé, il est ajouté à une hashtable, et son nombre de partie est mis en valeur. Si le joueur est déjà présent dans celle-ci, uniquement sa valeur est modifié en prenant la valeur actuelle et en l'additionnant avec le nombre de partie sur le fichier en cours de traitement. À la fin de ces opérations, la hashtable contient tout les joueurs avec leur nombre de partie respectifs. Un algorithme de tri permet de mettre dans une ArrayList les noms des joueurs triés avec leur nombre de parties croissant. Enfin, il est possible d'écrire dans le fichier playersData.dat le nom du joueur avec son nombre de partie.
Enfin, pour trouver les joueurs les plus actifs, il suffit de parcourir le fichier créé précédemment. Le nom du joueur sera sur une ligne et son nombre de partie sur la ligne suivante.

5.4.3 Recherche des ouvertures les plus communes

La recherche des ouvertures les plus communes parcourt juste tous les fichiers *lichess_standard_rated_AAAA_MM_opening_data.dat*, si l'ouverture lue n'existe pas dans la hashtable on l'ajoute, sinon on additionne le nombre d'ouverture lu dans le fichier au précédent. A la fin on obtient le nombre d'occurences pour chaque ouvertures. On a plus qu'à trier le résultat dans l'ordre décroissant et l'afficher. Cette requête est relativement peu pré-traitée car elle correspond a un assez faible volume de données (environ 2500 ouvertures différentes max. par mois).

6 Jeu d'essai

6.1 Lancement du client et du serveur

6.1.1 Client

```
yd271069@MI105-03:/travail2/yd271069/info4b_project/Client$ java client
Connecting to 127.0.0.1:1085
Welcome !
-------
Enter 'help' to get some help.
-------
If you want to quit, tap 'exit'
Enter a command
```

Figure 4: Lancement du client.

6.1.2 Serveur

```
yd271069@MI105-03:/travail2/yd271069/info4b_project/Server$ java server
Server listenning on port 1085
```

Figure 5: Lancement du serveur.

6.2 Affichage de l'aide

6.2.1 Client

```
Enter a command
help
search <nickname> [looking for <nickname>'s games]
        y [wish to see those games]
                <Integer> [number of games to see]
active <Integer> [looking for the <Integer> most active players]
opening <Integer> [looking for the <Integer> top openings]
help [looking for help]
exit [client exit]
Enter a command
```

Figure 6: Aide du client.

6.3 Recherche des parties d'un joueur

6.3.1 Client

```
Enter a command
search 2girls1cup
Searching for 2girls1cup
1327 games found for 2girls1cup in 1434ms
Do you want to see those games ? (Y/N)
y
How many ?
42
============== Game 1 ==============
Type : Rated Bullet game
URL : https://lichess.org/aak8624y
White : sluchaempobedyu
Black : 2girls1cup
Result : 1-0
Date : 2013.01.27
Opening : Sicilian Defense: Hyperaccelerated Fianchetto


============== Game 2 ==============
Type : Rated Bullet game
URL : https://lichess.org/rceyxu8e
White : 2girls1cup
Black : sluchaempobedyu
Result : 0-1
Date : 2013.01.27
Opening : Blackmar-Diemer Gambit


============== Game 3 ==============
Type : Rated Bullet game
URL : https://lichess.org/sgk0vgff
```

Figure 7: Recherche d'un joueur.

Ici on souhaite afficher les parties du joueur *2girls1cup*. Le client contacte donc le serveur (Voir Figure 9) qui lui renvoie le nombre totale de partie du joueur cherché. Le client demande donc si l'utilisateur souhaite afficher les parties. Si oui l'utilisateur entre un nombre communiqué au serveur et le serveur renvoie le nombre désiré de parties (Voir Figure 8).

```
-------------- Game 41 --------------
Type : Rated Bullet game
URL : https://lichess.org/jt6orxfk
White : argo
Black : 2girls1cup
Result : 1-0
Date : 2013.01.31
Opening : Sicilian Defense: Hyperaccelerated Dragon


============== Game 42 ==============
Type : Rated Bullet game
URL : https://lichess.org/l0m0ckmu
White : 2girls1cup
Black : argo
Result : 0-1
Date : 2013.01.31
Opening : Scotch Game: Goering Gambit, Double Pawn Sacrifice


Result found in 661ms
Enter a command
```

Figure 8: Affichage du résultat.

6.3.2 Serveur

```
New connexion from /127.0.0.1 (0)
42
Hashtable building ...
Hashtable done in 21ms
Closed connexion with/127.0.0.1
```

Figure 9: Processus côté serveur.

Ici on voit ce qu'il s'est passé pendant la communication avec le serveur. La requête commence au *New connexion from /127.0.0.1 (0)*. Le serveur répond alors avec le nombre de parties total pour le joueur correspondant. La ligne *42* correspond à la réponse du client pour le nombre de parties à afficher. Une Hashtable se créé alors contenant les parties lues par rapport au fichier dans lequel elle sont lues. Une fois la Hashtable complétée, toutes les valeurs contenues sont renvoyées au client.

6.4 Affichage des joueurs les plus actifs

6.4.1 Client

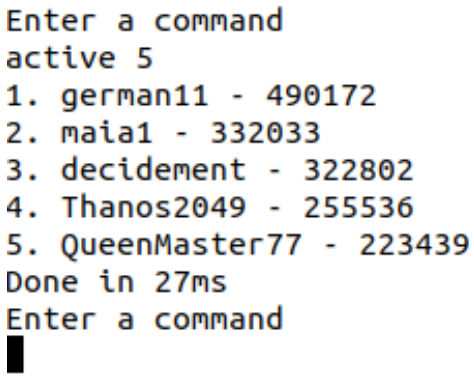

Figure 10: Affichage des 5 joueurs les plus actifs.

6.5 Affichage des ouvertures les plus fréquentes

6.5.1 Client

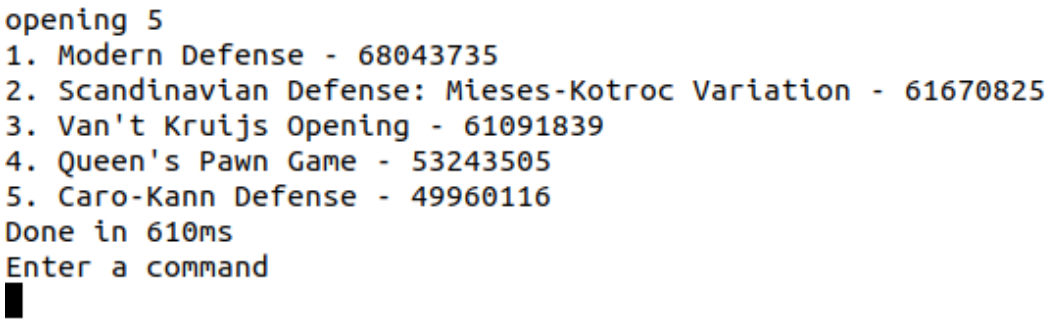

Figure 11: Recherche des 5 ouvertures les plus fréquentes.

6.5.2 Serveur

```
======= Reading lichess_db_standard_rated_2021-09_opening_data.dat=======
Reading content
Time to read file : 3ms
======= Reading lichess_db_standard_rated_2021-10_opening_data.dat=======
Reading content
Time to read file : 2ms
======= Reading lichess_db_standard_rated_2021-11_opening_data.dat=======
Reading content
Time to read file : 2ms
======= Reading lichess_db_standard_rated_2021-12_opening_data.dat=======
Reading content
Time to read file : 2ms
======= Reading lichess_db_standard_rated_2022-01_opening_data.dat=======
Reading content
Time to read file : 2ms
======= Reading lichess_db_standard_rated_2022-02_opening_data.dat=======
Reading content
Time to read file : 2ms
======= Reading lichess_db_standard_rated_2022-03_opening_data.dat=======
Reading content
Time to read file : 1ms
Closed connexion with/127.0.0.1
```

Figure 12: Recherche par fichier.

7 Liens utiles et remerciements

7.1 GitHub

Pour notre projet, nous avons utilisé GitHub.
GitHub de Corentin Diehl : https://github.com/Corentin190
GitHub de Yoan Dusoleil : https://github.com/Yaon-C2H8N2

7.2 Remerciements

Nous tenons à remercier nos professeurs Eric Leclercq et Annabelle Gillet qui nous ont particulièrement bien aidé pour la réalisation de ce projet. Particulièrement pour le prêt d'une machine sur le réseau de l'université avec une capacité de stockage et de puissance suffisante afin de réaliser notre but : traiter la totalité de la base de donnée.