Rapport de projet :

L2 : INFO4B

MKAVAVO Eddin

Introduction :

Le projet

Pour ce projet j'ai décidé de faire le sujet LiChess. Ce choix a été motivé par deux raisons principales :

La première est que sur ces dernières années nous avons déjà dus faire de nombreux jeux comme projet de fin d'année/semestre et que je désirais un peu de changement.

La seconde est que je voulais vraiment essayer de manipuler une grosse quantités de données avec ce projet. Je me suis heurté a de nombreuses difficultés, mais finalement je suis capable de présenter un projet fonctionnel et organisé.

Première compréhension (et erreurs)

Mon départ sur ce projet a été bancale car je suis partie sur une première compréhension des consignes complètement fausse. Finalement je me suis vue obligé de revoir le fonctionnement et la logique de mon système en plein milieu de sa conception. Le projet actuel est bien différent de celui que je voulais établir en premier lieu, mais je pense qu'il est tout de même important que j'évoque cette première fausse direction prise car elle est à elle seule une parfaite illustration de pourquoi ce projet (bien que parfaitement fonctionnel) est en deçà de ce que j'aurais pu produire avec une meilleur préparation et un temps plus raisonnable a la réflexion du système.

Dans un premier temps j'avais pour idée d'emprunter la puissance de calcul des ordinateurs connectés au serveur. Ces derniers agiraient ainsi comme des Threads en recevant une quantités de donnée a traiter et en renvoyant une réponse. Ces réponses auraient été par la suite traité à nouveau par le serveur pour fournir une réponse qui aurait été envoyé de nouveau au client concerné. J'ai rencontré de nombreux problèmes avec ce système que j'ai essayé et amélioré pendant plusieurs semaines. Cependant j'ai fini par réaliser qu'il était contreproductif et que le cœur du problème venait du fait que je ne m'étais pas posé la question de comment optimisé la vitesse d'accès aux information sur la machine. De plus, l'envoie incessant de donnée était un vrai frein a la vitesse de traitement des demandes, peu importe comment je retournais la chose. J'ai donc décidé de m'orienter vers un bashing de mes données en amonts pour optimiser l'accès à ces dernières.

Départ et correction

Après avoir gaspillé plusieurs semaines, j'ai réfléchi a comment optimisé l'accès a mon document. J'ai fini par « prétrier » mes données dans deux fichier sauvegardé dans un dossier (Nous y reviendrons en détails). Ce procédé permet de ne pas avoir à parcourir l'intégralité du document pour accéder à une information. Plus précisément, elle me permet de lancer en parallèle plusieurs Threads qui vont travailler sur une partie bien précise du document et renvoyer leur résultat. Ses dernier seront traité et organisé par le serveur pour fournir une réponse aux client.

Analyse fonctionnelle

Organisation des données :

Avant de commencer à parler de mon système, il est important de voir comment sont organisé les données qui doivent être traité. En voici un exemple :

```
[Event "Rated Bullet tournament https://lichess.org/tournament/yc1WW2Ox"]
[Site "https://lichess.org/PpwPOZMq"]
[Date "2017.04.01"]
[Round "-"]
[White "Abbot"]
[Black "Costello"]
[Result "0-1"]
[UTCDate "2017.04.01"]
[UTCTime "11:32:01"]
[WhiteElo "2100"]
[BlackElo "2000"]
[WhiteRatingDiff "-4"]
[BlackRatingDiff "+1"]
[WhiteTitle "FM"]
[ECO "B30"]
[Opening "Sicilian Defense: Old Sicilian"]
[TimeControl "300+0"]
[Termination "Time forfeit"]

1. e4 { [%eval 0.17] [%clk 0:00:30] } 1... c5 { [%eval 0.19] [%clk 0:00:30] }
2. Nf3 { [%eval 0.25] [%clk 0:00:29] } 2... Nc6 { [%eval 0.33] [%clk 0:00:30] }
3. Bc4 { [%eval -0.13] [%clk 0:00:28] } 3... e6 { [%eval -0.04] [%clk 0:00:30] }
4. c3 { [%eval -0.4] [%clk 0:00:27] } 4... b5? { [%eval 1.18] [%clk 0:00:30] }
5. Bb3?! { [%eval 0.21] [%clk 0:00:26] } 5... c4 { [%eval 0.32] [%clk 0:00:29] }
6. Bc2 { [%eval 0.2] [%clk 0:00:25] } 6... a5 { [%eval 0.6] [%clk 0:00:29] }
7. d4 { [%eval 0.29] [%clk 0:00:23] } 7... cxd3 { [%eval 0.6] [%clk 0:00:27] }
8. Qxd3 { [%eval 0.12] [%clk 0:00:22] } 8... Nf6 { [%eval 0.52] [%clk 0:00:26] }
9. e5 { [%eval 0.39] [%clk 0:00:21] } 9... Nd5 { [%eval 0.45] [%clk 0:00:25] }
10. Bg5?! { [%eval -0.44] [%clk 0:00:18] } 10... Qc7 { [%eval -0.12] [%clk
0:00:23] }
11. Nbd2?? { [%eval -3.15] [%clk 0:00:14] } 11... h6 { [%eval -2.99] [%clk
0:00:23] }
12. Bh4 { [%eval -3.0] [%clk 0:00:11] } 12... Ba6? { [%eval -0.12] [%clk 0:00:23] }
13. b3?? { [%eval -4.14] [%clk 0:00:02] } 13... Nf4? { [%eval -2.73] [%clk
0:00:21] } 0-1
```

Deux choses TRÈS importantes a noté, car elles ont été source de conflits avec moi-même. Premièrement, contrairement à ce qu'on pourrait croire avec l'exemple plus haut, TOUTE les informations ne sont en fait que sur une seule et même ligne. Même la dernière partie (Symbolisant le déroulement de la partie) est en fait une seule ligne.

Deuxièmement, les informations NE SONT PAS normalisées. Les informations « WhiteRatingDiff » et « BlackRatingDiff » ne sont pas systématiquement présente dans les parties. Ainsi, dans mes premières versions, j'essayais de récupérer les parties en fonction du nombre de ligne (non vide) passé. Evidemment, le nombre de lignes change en fonction de si ces deux informations sont présente ou non. Ce qui a inévitablement mené a des décalages inexplicable pendant un long moment. Ceci est la raison précise de pourquoi je déterminer plus l'information de la ligne selon son numéro.

Vue d'ensemble

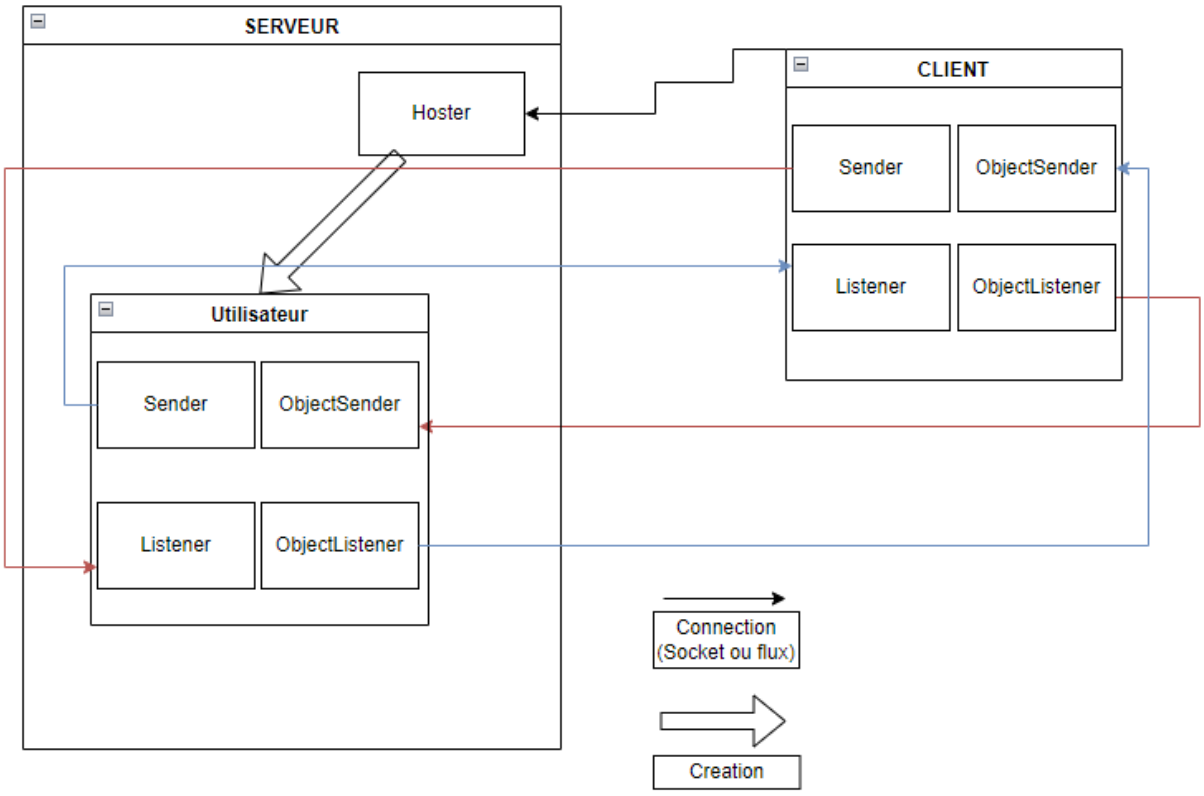
Le but du projet est de mettre en place une communication entre un serveur et un ou plusieurs clients. Cette communication doit permettre au client d'accéder à des informations présent sur le serveur. Ces informations sont fragmentées et répartie dans un ou plusieurs fichiers de taille volumineuse. Cependant, le client doit pouvoir avoir accès à l'information rapidement et pouvoir mettre en route plusieurs commandes simultanément. L'utilisation de Threads sera primordiale et la gestion de la mémoire et du processeur central dans ce projet. Il faut dans un premier temps s'assurer que le serveur est capable de parcourir les données de bout en bout sans excéder sa capacité de

mémoire. Pour se faire le serveur utilisera deux fichiers (détaillés dans la prochaine partie) pour accéder à l'information précisément demandé. Ainsi j'évite une lecture du document de bout en bout. Cette procédure est découpée et donnée à divers Threads qui vont en parallèle, traiter l'information. Le nombre de Thread pouvant tourner simultanément pour une même requête est plafonné a 10. Le multithreading n'est pas utilisé pour toute les demandes dans ce projet. Car certaines peuvent être répondu en utilisant directement la puissance brute du serveur.

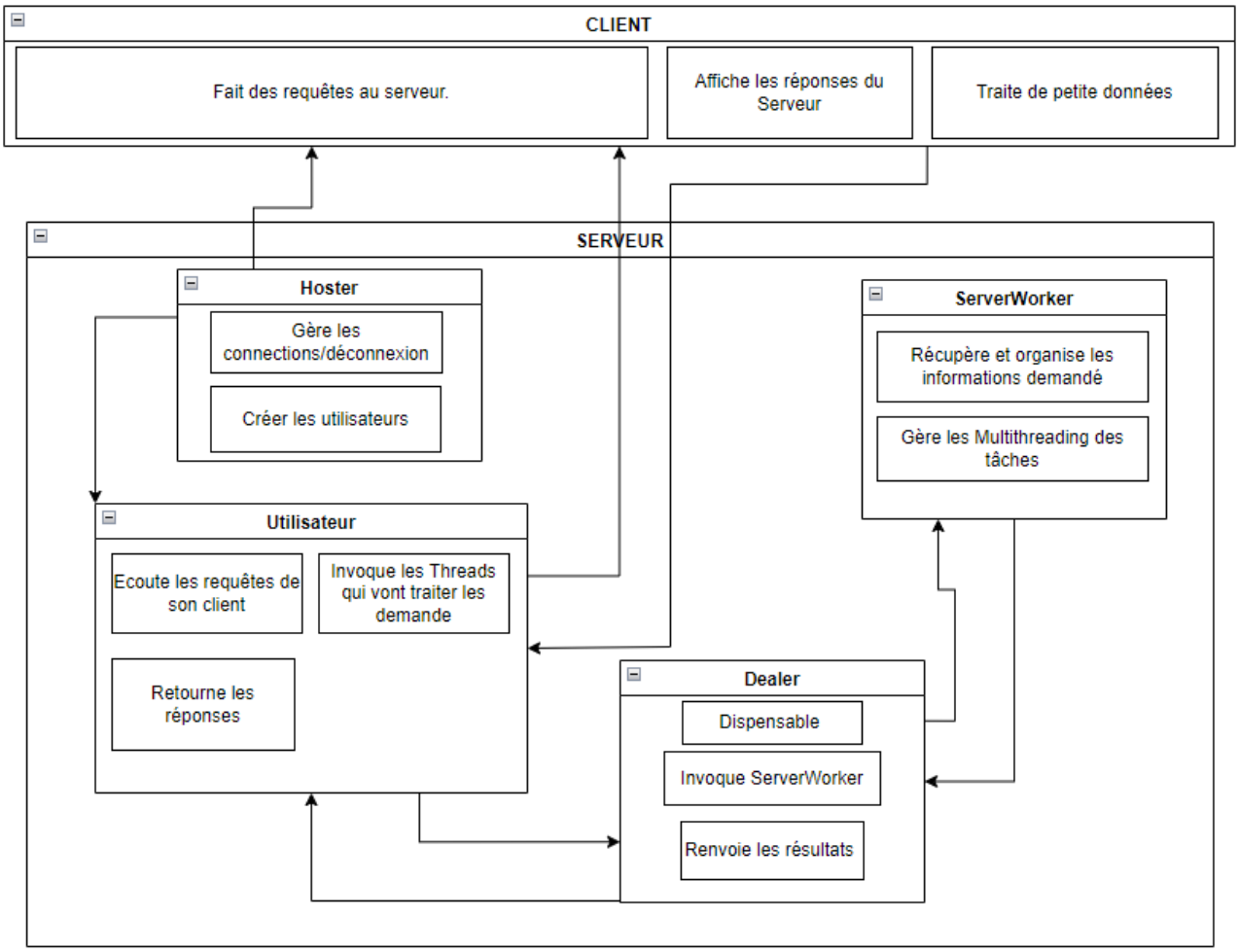
La communication serveur client est la partie simple du projet. Il faudra simplement conserver une communication tant que le client est présent et le supprimer lorsqu'il ne l'est plus. Il sera nécessaire de pouvoir envoyer des chênes de caractères et dans certains cas, des objets au client. Il sera donc utile d'avoir deux Sockets ouvert par clients un pour les chênes de caractères et l'autre pour les objets.

D'un point de vue connexion, les classes class les plus importantes sont sans doute les class « Hoster » et « Utilisateur » du serveur et le divers « Sender » et « Listener » côté client.

Le Hoster est celui qui va gérer les connexions avec chaque client, autoriser les connexions et gérer les déconnections. Il fonctionne en parallèle de la class utilisateur qui est la représentation d'un Client pour le serveur. La class Utilisateur va garder en mémoire les informations de connexions avec le client qu'il représente (Socket, Listeners, Senders). Les class « Sender », « ObjectSender », « Listener » et « ObjectListener » sont lié a la class « Utilisateur » mais sont aussi présente côté client. Elles représentent les voies de communications entre Serveur et Clients

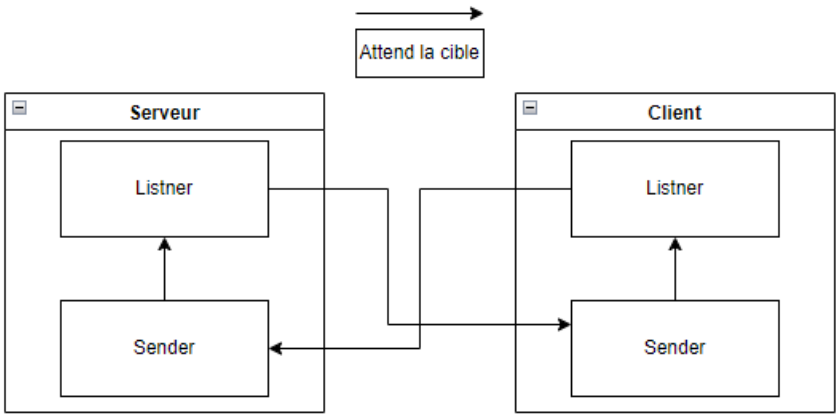


Architecture logiciel



Fonctionnement

Lorsqu'un client se connecte au serveur, de part et d'autre un « Sender », un « ObjectSender », un « Listener », et un « ObjectListener » seront créé dans cet ordre de priorité pour le client et le serveur. J'insiste sur ce fait, car je me suis retrouvé dans une situation de « Deadlock » pour cette raison. Lorsqu'un Listener est créer d'un côté, il attend le signer du Sender. Cependant, si du côté serveur, ou client le Listener est créé en premier, alors serveur et client attendront indéfiniment le Sender ou ObjectSender de l'autre.



Un fois la connexion entre serveur et client établi, le client peut commencer à envoyer des requêtes. Ces dernières peuvent être affiché en tapant la commande « help ». La casse est ignorée. Voici ce qu'affiche la commande help :

```
COMMANDS :
help :
        Affiche les commandes

getClientsNumber :
        Retourne le nombre de clients actuellement connecté

getPlayers :
        Retourne tous les joueurs ainsi que leur nombre de parties
                Peut aussi être utilisé pour afficher les joueurs trié selon leur nombre de partie avec les opération <,>,= et <> suivi d'une ou de deux valeurs

mostUsedOpening :
        Retourne les 5 ouvertures les plus utilisé de totue la partie.
                Si la commande est suivi du nom d'un joueur, elle retourne les 5 ouverture les plus utilisé du joueur

nbPartieFor :
        Affiche le nombre de parties d'un joueur

showGameFor :
        Affiche le ou les parties d'un joueurs (Faire suivre la commande du nom du joueur ainsi que du numero des parties
```

Nous avons déjà vue ce que fait la commande « help ».

- « getClientNumber » est une commande simple qui affiche le nombre de clients connecté actuellement au serveur.

- « getPlayers » est une commande très importante car elle va permettre de télécharger la liste de tous les joueurs et de pouvoir la traiter par la suite. Lors de sa première utilisation, elle ne fera que télécharger cette information. Par la suite elle permettra d'afficher tous les joueurs, classé selon la première lettre de leur pseudo et séparé par un « * ». Cette commande accepte des arguments. Elle en requière au maximum 4, tous ce qui excède ceci sera ignoré. Le premier argument après le nom de la commande, est une signe parmi les suivant : <, >, <=, >= ou <>. Ce dernier signe représente une encapsulation de valeur. Les deuxième et troisième arguments sont des valeurs chiffré positive entière. En appliquant ses arguments, il est possible de filtrer les joueurs selon leur nombre de partie. Par exemple, la commande « getPlayers <> 2000 2050 » affichera la liste des joueurs classé par la première lettre de leur pseudo et ayant fait entre 2000 et 2050 partie (inclus).

- « mostUsedOpening » est l'une des commandes qui prend le plus de temps à être traité. Cette commande va afficher la liste des 5 ouvertures les plus utilisé sur l'ensemble des données et leur occurrence. Cette commande peut être suivi d'un argument qui est le pseudo d'un joueur. Dans ce cas, elle affichera les 5 ouvertures les plus utilisé par le joueur et leur occurrence.

- « nbPartieFor » est une commande simple qui va afficher le nombre de partie qu'a fait un joueur. Elle demande un argument qui est le pseudo d'un joueur

- « showGameFor » est une commande qui va afficher toute les information de la partie d'une ou de plusieurs partie d'un joueur. Elle demande au moins deux arguments, mais peut théoriquement en accepter une infinité. Le premier argument est le pseudo d'un joueur, le second est le numéro de la partie devant être affiché (commence à 0). Il est parfaitement possible de renseigner plusieurs parties en une seule fois. Par exemple, la commande : « showGameFor ImSoFabulous 0 1 2 3 4 5 », affichera les parties de 0 à 5 du joueur « ImSoFabulous ».

- « END » est la commande déconnexion pouvant être envoyé par un client. (Cette commande est sensible à la casse

Description des structures de données

Structuration des joueurs

Structurer les données est un point crucial dans ce projet. En fonction de comment cela a été fait, les performances peuvent être drastiquement perturbé. Une de mes premières idées était de simplement prétraiter mon document dans une Hashtable<String,Integer> (pseudo joueur, nombre de partie) pour directement pouvoir récupérer le nombre de parties qu'à fait un joueur. Cependant cette façon de faire est très limité. Elle me permet d'accéder rapidement au nombre de parties d'un joueur, mais uniquement à cette information. Ainsi j'ai préféré structurer mes données dans une Hashtable<String, ArrayList<Long>> (Pseudo, ArrayList qui contient le nombre de caractères à sauter pour arriver au début de la nième partie du joueur). Cette configuration me permet d'avoir accès rapidement à chaque partie d'un joueur, mais d'aussi avoir accès à leur nombre de partie avec un .size(). L'inconvénient est que le fichier serializé est plus lourd que ma première idée. Il faut compter 6 minutes pour que le serveur se lance, uniquement pour ce fichier.
(Voir partie sur la Class Hoster pour plus de détails)

Structuration générale

Dans soucis de simplicité et de polyvalence, mon second bashing est très basique. Il va simplement fragmenter le fichier principales en différentes parties représenté par le nombre de caractère à sauter avant d'arriver à la zone à traiter. Cette information sera rangé dans une Hashtable<Integer,Long>. Initialement j'ai essayé de découper le document en fragments de 5 000 000 de parties. Puis essayé de le fragmenté en zone de 500 000 parties. Puis je suis revenue à 5 000 000. Cette configuration offre un meilleur équilibre par rapport à l'utilisation des ressources du serveur. Particulièrement dans un contexte avec plusieurs clients.
(Voir partie sur la Class Hoster pour plus de détails)

Classes principales et méthodes

Class Hoster (Thread) :

La class Hoster est la class qui va gérer (par le biais de d'autres class, la communication avec les clients). Elle tourne en parallèle de toutes les autres.

run()

```
39
40      @Override
41      public void run(){
42          int i=0;
43          try{
44              //Downloading datas
45              System.out.println("Starting Hoster "+ new Date());
46              ObjectInputStream objectReader = new ObjectInputStream(new FileInputStream(new File("savedDatas/playersList.txt")));
47              this.playersList = (ChessData) objectReader.readObject();
48              objectReader.close();
49              objectReader = new ObjectInputStream(new FileInputStream(new File("savedDatas/hashedTable500000.txt")));
50              this.hashedTable = (ChessData)objectReader.readObject();
51              objectReader.close();
52              System.out.println("Hoster running "+ new Date());
53
54              //Starting hosting
55              while(true){
56                  this.socketOfServer = listener.accept();
57                  this.socketForObject = listener.accept();
58                  System.out.println(""+socketOfServer.getInetAddress()+" is connected");
59                  Utilisateur u = new Utilisateur("user"+i, this.socketOfServer,this.socketForObject,this);
60                  users.put("user"+i,u);
61                  i++;
62              }
63          }catch(IOException e){
64              e.printStackTrace();
65          }catch(ClassNotFoundException e){
66              e.printStackTrace();
67          }
68
69      }
70
```

Cette méthode a deux utilités :
Initialiser les fichiers qui ont été prétraité (de la ligne 42 à la ligne 50 de la class). Et accepter, puis gérer les clients qui veulent se connecter (de la ligne 52 à la ligne 59).

Lors de l'initialisation, un ObjectInputStream va se charger de lire dans le dossier « savedDatas » pour récupérer les objets sauvegardé. Ses objets seront par la suite accessibles directement par la class Hoster avec des gets.

Un fois l'initialisation terminé, le Hoster attend des demande de connexion à accepter. Lorsqu'une demande arrive, il procède à la création d'un Utilisateur. Qu'il va ranger dans une Hashtable<String,Utilisateur>. Cette Hashtable sera utile pour supprimer un utilisateur lors de la déconnexion.

Class ChessData (Serializable) :

ChessData est une class de stockage. Elle n'a pas de méthodes sophistiqué et est né du besoin de pouvoir stocker et envoyer plusieurs Hashtable de paramétrés différent. Pour cette raison elle contient de nombreuse Hashtable facile d'accès avec des getters et setters.

Class Listener (Thread) :

Une class très importante et central dans le rapport Client, Serveur. Leur fonctionnement différent en fonction de s'il s'agit du client ou du serveur, mais le principe reste le même. Il va en permanence écouter ce que dit l'autre côté et agir si la situation est pertinente.

Côté Client :

```
14      public Listener(Socket sockIn){
15          this.close = false;
16          this.sBridge = sockIn;
17          try{
18              this.readerMan = new BufferedReader(new InputStreamReader(this.sBridge.getInputStream()));
19          }catch(IOException e){
20              e.printStackTrace();
21          }
22
23          this.lastRead = "";
24      }
25
```

Le constructeur est basique. Il va simplement ouvrir un BufferedReader sur le flux du socket passé en paramètre. Ce BufferedReader ne sera fermé que lorsque la méthode « shutDown » sera appelé. (Uniquement à la déconnexion).

```
26      @Override
27      public void run(){
28          String s = null;
29          try{
30              while(!this.close){
31                  s = readerMan.readLine();
32                  if(s!=null){
33                      setLastRead(s);
34                      System.out.println(s);
35                  }
36              }
37          }catch(IOException e){
38              System.out.println("Listener closed");
39          }
40      }
41
```

Une fois lancé, avec un .start(), le Listener va passer en attente active d'une information. Lorsque celle si en percera une pertinente (Différente de null) alors elle l'affichera. C'est par ce biais que les réponses aux commandes sont réceptionnées et affichées.

Coté Serveur :

Le principe reste le même. Mais il y a une différence notable :

```
30      @Override
31      public void run() {
32          String s = null;
33          ArrayList<String> sList = null;
34          try {
35              while (!this.close && (s = readerMan.readLine()) != null) {
36
37                  if (isAnOrder(s) >= 0) {
38                      System.out.println("ORDER");
39                      setLastOrder(s);
40                      sList = getArguments(s);
41                      this.myUser.launchDealer(isAnOrder(s),sList);
42                  } else {
43                      setLastInput(s);
44                      System.out.println(this.myUser.getNom()+" : "+s);
45                      if (s.equalsIgnoreCase("END")) {
46                          this.myUser.shutMeDown();
47                      }
48                  }
49
50              }
51          } catch (IOException e) {
52              e.printStackTrace();
53          }
54      }
55
```

La différence majeur se fait a partir de la ligne 37. A chaque fois qu'un serveur percevra une information venant d'un client, il va définir s'il s'agit d'une commande connue de lui-même. Ceci ce fait via la méthode isAndOrder() qui retourne une valeur entre -1 et 5. -1 correspondant à a l'absence de commande et les valeur de 0 à 5, les indices correspondant a l'ordre en question.
Les ordres/commandes sont définit dans le Hoster à partir de la ligne 8.

Class Sender :

Cette class avait pour but premier d'écouter les entrée clavier et d'envoyer le contenue au Serveur (Côté client) et de permettre au serveur d'envoyer simplement des informations de réponse au Client. Son utilisation c'est vue déformé côté client, car je l'utilise aussi pour filtrer quelque demande en amont.

```
33      @Override
34      public void run() {
35          String s;
36          boolean getIn = true;
37          try {
38              while (!this.close && (s = this.inputLook.nextLine()) != null) {
39                  if (isAnOrder(s) == 2) {
40                      System.out.println(2);
41                      if (this.myLink.getObjectListenerMan().getPlayers().getDataInt() != null) {
42                          String signe = "<>";
43                          int valeur1 = 1, valeur2 =10;
44                          if (!getFirstWord(s).equalsIgnoreCase(getWord(s, 2))) {
45                              try{
46                                  signe = getWord(s, 2);
47                                  valeur1 = Integer.parseInt(getWord(s, 3));
48                                  valeur2 = Integer.parseInt(getWord(s, 4));
49                              }catch(NumberFormatException e){
50                                  System.out.println("Valeur(s) invalide(s)");
51                                  getIn = false;
52                              }
53                          }
54                          if(signe.length()>2 || (!signe.equals("<") && !signe.equals("<=") && !signe.equals(">")
55                                                  && !signe.equals(">=") && !signe.equals("==") && !signe.equals("<>"))){
56                              System.out.println("Signe invalide");
57                              getIn = false;
58                          }
59                          if(getIn ){
60                              if (valeur2 >= valeur1)
61                              affichePlayer(triPlayers(signe, valeur1, valeur2));
62                              else
63                              System.out.println("Valeur des marges invalides");
64                          }
65                          getIn = false;
66                      }
67                  }
68                  if (s != null && !s.equals("") && getIn) {
69                      this.writerMan.write(s);
70                      this.writerMan.newLine();
71                      this.writerMan.flush();
72                      if (s.equals("END")) {
73                          myLink.shutDownAll();
74                      }
75                  }
76                  getIn = true;
77              }
78          } catch (IOException e) {
79              e.printStackTrace();
80          }
```

De la ligne 68 à la ligne 75, l'information est simplement envoyée au Serveur. Ce dernier se chargera des traitement. De la ligne 41 à 66, nous avons une série de vérifications visant à détecter les demandes inutile au serveur. Par exemple l'envoie d'information déjà obtenue et stocké sur la machine du client. Je profite aussi du moment pour faire diverses vérification pour la requête getPlayers, et plus précisément vis-à-vis de la gestion des arguments.

triPlayers()

```
109     public Hashtable<Character, String> triPlayers(String signe, int nb, int nb2) { // tri les joueurs par ordre
110                                                                                      // alphabetique et les retourne
111         Hashtable<Character, String> nomParOrdre = new Hashtable<>();
112         char firstChar;
113         String tmpStorage;
114         switch (signe) {
115             case "<":
116                 for (Map.Entry<String, Integer> player : this.myLink.getObjectListenerMan().getPlayers().getDataInt()
117                         .entrySet()) {
118                     if (player.getValue() < nb) {
119                         firstChar = player.getKey().charAt(0);
120                         firstChar = Character.toUpperCase(firstChar);
121                         if (nomParOrdre.containsKey(firstChar)) {
122                             tmpStorage = nomParOrdre.get(firstChar);
123                             nomParOrdre.replace(firstChar, tmpStorage, tmpStorage + " * " + player.getKey());
124                         } else {
125                             nomParOrdre.put(firstChar, player.getKey());
126                         }
127                     }
128                 }
129                 break;
130             case "<=": ···
145             case ">": ···
160             case ">=": ···
175             case "=": ···
190             case "<>": ···
205         }
206         return nomParOrdre;
207     }
208
```

Cette méthode est appelé à chaque fois que getPlayers() est appelé par le client. Elle va récupérer l'objet Player de type Hashtable<String, Integer> et le réorganiser pour correspondre aux demandes passé en paramètre par le client (L'objet player n'est pas modifié dans le processus).

Class ObjectSender et ObjectListener

Ces deux class fonctionne sur le même principe que Sender et Listener. En principe elles étaient destinées à avoir un plus grand rôle dans le projet initiale visant à multiplier les échanges d'objet ChessData entre Clients et Serveur pour profiter de la puissance de calcul de tout ceux présent sur le réseau. Finalement ils ne sont que très rarement utilisé dans cette version du projet. ObjectSender est notamment utilisé lors du premier appel de getPlayers().

Toute les prochaine class existent uniquement côté serveur.

Class Hasher

Cette class a pour unique rôle de procéder au Bashing des données en amont du lancement final. Elle n'est pas destinée a fonctionner en parallèle du Serveur une fois lancé. Elle contient deux méthodes très importante. (Ces méthodes étant longue, je n'afficherais que les parties importante ici)

hashPlayers()

```
37      try {
38          BufferedReader fileReader = new BufferedReader(new FileReader(this.fileForWork));
39          while ((line = fileReader.readLine()) != null && !secureStop && counterIn<100000) {
40              nbChar+=line.length()+1;
41
42              if (getFirstWord(line).equalsIgnoreCase("[White") || getFirstWord(line).equalsIgnoreCase("[Black"))
43              {
44                  nomJoueur = getWordBetwen(line);
45                  //Ajout jouer et/ou partie
46                  if(!dataForPlayers.containsKey(nomJoueur)){
47                      dataForPlayers.put(nomJoueur, new ArrayList<>());
48                  }
49                  dataForPlayers.get(nomJoueur).add(nbCharToNextPartie); // /!\ Recupère une ArrayList a chaque fois plus grosse
50                  counterIn++;
51              }
52              if(getFirstWord(line).equalsIgnoreCase("1.")){
53                  //Passe la ligne vide
54                  line = fileReader.readLine();
55                  if (line!=null)
56                      nbChar +=line.length()+1;
57                  else
58                      secureStop = true;
59                  //
60                  nbCharToNextPartie = nbChar;
61              }
62              //LoadBarre
63  >           /*if (counterIn >= 10000000) {···
67          }
```

Cette méthode a pour rôle de créer le fichier listPlayer.txt stocké dans le dossier savedDatas. Cette partie aurait pu être bien mieux optimisé en procédant au bashing via du multithreading, pareil pour la lecture du fichier au lancement du Hoster. Cette possibilité n'a malheureusement pas été exploité.

hashGame()

```
96          try {
97              // Bashing
98              BufferedReader fileReader = new BufferedReader(new FileReader(this.fileForWork));
99              hashingGame.put(0, (long) 0);
100             this.idCounter++;
101             while ((line = fileReader.readLine()) != null) {
102                 firstWord = getFirstWord(line);
103                 nbChar += line.length() + 1;
104                 if (firstWord.equalsIgnoreCase("1.")) {
105                     partieCounter++;
106                     // Passage de la ligne vide
107                     line = fileReader.readLine();
108                     nbChar += line.length() + 1;
109                     //
110                 }
111                 if (partieCounter == nbPartieParClasseur) {
112
113                     hashingGame.put(this.idCounter, nbChar);
114                     this.idCounter++;
115                     partieCounter = 0;
116                     System.out.print("I");
117                 }
118             }
119             System.out.println("] End Bashing games at " + new Date());
120             fileReader.close();
121
```

hashGame a pour but de créer le fichier hashedGame.txt. Ce fichier (Comme listPlayer) est dépendant du fichier de donnée originel. Ainsi le programme ne peut pas fonctionner sans ce dernier.

hashGame et hashPlayers vont stocker sous forme de Long, le nombre de caractère a passer avant d'arriver à un point précis du document originel. Ainsi, par l'utilisation de la méthode .skip(), il est très facile de naviguer dans le fichier. Attention à ne pas oublier de compter le caractère de retour a la ligne. Ceci a été source de nombreuses confusions.

Class ServerWorker

La Class ServerWorker est une class très importante du projet. C'est cette class qui va faire les traitements et retourner une réponse transmissible aux clients. Cette class va créer et appeler des instance d'elle-même pour traiter les données en multithreading.

```
19      public ServerWorker(Hoster hoster,ServerWorker summoner,int commande, int id, ArrayList<String> args) {
20          this.commande = commande;
21          this.dataIn = hoster.getPlayersList();
22          this.hoster = hoster;
23          this.myId = id;
24          this.args = args;
25          this.answerString = new ArrayList<>();
26          this.answerInt = new ArrayList<>();
27          this.returnAnswer="";
28          this.summoner = summoner;
29      }
30
31      @Override
32      public void run() {
33          switch(this.commande){
34  >           case 2: ···
37  >           case 3: ···
48  >           case 4: ···
53  >           case 5: ···
70          }
71      }
```

Le constructeur de la Class prend divers arguments.

- hoster permet d'accéder aux données nécessaire au traitement des demandes.
- summoner représente si oui ou non le Thread actuel est un fils du ServerWorker initial. Cette valeur est null s'il s'agit du ServerWorker initial.
- commande est le numéro de la requête obtenue avec isAnOrder()
- id servira au ServerWorker secondaire pour savoir sur quelle plage de données ils doivent travailler. Cette valeur est égale a -1 pour le ServerWorker initial.
- args représente la liste des arguments renseigné en paramètre par le client

Du côté du run() :

Un simple switch() sur la valeur de this.commande permet de faire appel a la bonne méthode selon la bonne commande renseigné a la création du Thread.

Nous allons voir l'exemple pour une méthode de traitement de la class. Toute les autres sont construite de la même façon. Tout ce qu'elles feront c'est cherché à un autre endroit.

Prenons en exemple la méthode getMostUsedOpeningFor(). Cette méthode contient deux parties distinct :

La partie « Master »

```
210     public String getMostUsedOpeningFor(String joueur){
211         if(this.hoster.getPlayersList().getDataPlayer().get(joueur)==null) return "Joueur non valide";
212         ArrayList<Long> parties = this.hoster.getPlayersList().getDataPlayer().get(joueur);
213         int i =0;
214         int tmpSwitch=0;
215         String line = null;
216         boolean run = true;
217         Hashtable<String,Integer> sorteOpening = new Hashtable<>();
218         ArrayList<ServerWorker> workers = new ArrayList<>();
219 
220         if(this.summoner==null){
221             //Master
222             while(i<parties.size()){
223                 if(getThreads()<this.MAX_THREAD){
224                     setThreads(getThreads()+1);
225                     //Chaque Worker traitera 500 parties consecutive du joueur en question (ou moins)
226                     workers.add(new ServerWorker(this.hoster, this, this.commande, i, this.args));
227                     workers.get(i/500).start();
228                     i+=500;
229                 }
230             }
231 
232             try{
233                 for(i=0;i<workers.size();i++){
234                     workers.get(i).join();
235                 }
236             }catch(InterruptedException e){···
239 
240 
241 
242             return triReponse(workers);
```

Le « Master » va créer un certain nombre de « workers » et leur attribuer une tâche a chacun. Une fois que tous les workers ont fini, le Master va collecter toutes les réponses des workers et fournir une seule réponse qui synthétise le travail des workers. Cette reponse sera retourné et traité a l'extérieur par le Dealer aillant invoqué le ServerWorker.

La partie « Worker »

Le code du Worker est très dense, mais son idée est plutôt simple. L'objectif est de parcourir le document jusqu'à la fin ou jusqu'au début du prochain marqueur marquant le début d'une autre partie du document. Pendant la lecture, si le worker trouve un élément marquant, il ajoute l'information a une ArrayList qui sera par la suite traité par le Master pour fournir une réponse utilisable. Pour comprendre un peu mieux le code, découpons-le en fragments :

- Lignes 254 – 275. La partie de lecture du document et d'identification des
    - 256 – 264. Passage de la partie actuel du joueur, a la partie suivante. Le skip doit être traité de telle sorte qu'il n'est pas nécessaire de refaire un skip depuis le début du document
    - 265 – 266 arrêt du worker s'il a travaillé sur sa plage de 100 parties
- Lignes 267. Traitement de la partie pertinente (Actuellement la partie des ouvertures)
- Lignes 280 – 286. Tri des résultats pour faciliter le traitement du Master.
- Ligne 287. Retour d'une valeur nécessaire pour le bon fonctionnement de la méthode.

```
244         //Workers
245         try{
246             i = this.myId;
247             Long nbChar=parties.get(this.myId);
248             File file = this.hoster.getDataToUse();
249             BufferedReader fileReader= new BufferedReader(new FileReader(file));
250             fileReader.skip(nbChar);
251             i++;
252             while((line=fileReader.readLine())!=null && run){
253                 nbChar+=line.length()+1;
254                 //Detection de la fin d'une partie
255                 if(getWord(line, nb: 1).equals( anObject: "1.")) {
256                     line=fileReader.readLine();
257                     nbChar+=line.length()+1;
258                     if(i<this.myId+500 && i<parties.size()){…
262 
263                     i++;
264                 }
265                 //Detection de la zone interessante
266                 if(getWord(line, nb: 1).equalsIgnoreCase( anotherString: "[Opening")){
267                     if(sorteOpening.get(getWordBetwen(line))!=null){
268                         tmpSwitch = sorteOpening.get(getWordBetwen(line));
269                         sorteOpening.replace(getWordBetwen(line), tmpSwitch, tmpSwitch+1);
270                     }else{…
273                     }
274                 }
275             fileReader.close();
276         }catch(IOException e){…
279         for(i =0;i<5;i++){
280             this.answerString.add(getBigger(sorteOpening));
281             this.answerInt.add(sorteOpening.get(answerString.get(i)));
282             sorteOpening.remove(this.answerString.get(i));
283         }
284         this.summoner.setThreads(this.summoner.getThreads()-1);
285         System.out.println("End a Thread " + this.getName() + " : " +answerString+" "+answerInt);
286         return "worker 0";
```

triReponse()

TriReponse prend en argument un ArrayList<ServerWorker> qui représente tous les workers et les réponses qu'il apportent avec eux. Cette méthode va comparer toutes les réponses et retourner sous forme de chêne de caractères une réponse unifié. Cette méthode prendra en compte le fait que parmi le classement individuel de chaque Worker, il y a peut-être des doublons. Par exemple, sur les parties 500 à 1000 il y a peut-être 120 fois une même ouverture, et sur les parties de 1000 à 1500 il y aura 70 répétition de plus. Ces informations sont présente dans plusieurs fichiers de résultats différents. Cette méthode va les trouver et les unifier.

```
76  private String triReponse(ArrayList<ServerWorker> workers){
77      System.out.println( x: "Tri les plus utilisé");
78      ArrayList<String> mostUsedS = new ArrayList<>();
79      ArrayList<Integer> mostUsedI = new ArrayList<>();
80          while (mostUsedS.size() < 5) { // Recupère les 5 ouvertures
81              String opening = "";
82              int openingNb = -1;
83              int index = -1;
84              for (int i = 0; i < workers.size(); i++) {
85                  if (opening.equals( anObject: "")) {
86                      opening = workers.get(i).getAnswerString().get( index: 0);
87                      index = i;
88                      openingNb = workers.get(i).getAnswerInt().get( index: 0);
89                  } else {
90                      if (openingNb < workers.get(i).getAnswerInt().get( index: 0)) {
91                          opening = workers.get(i).getAnswerString().get( index: 0);
92                          index = i;
93                          openingNb = workers.get(i).getAnswerInt().get( index: 0);
94                      }
95                  }
96              }
97              if (opening.equals( anObject: "") || opening == null)
98                  break;
99              /* Créer une troisième liste a partir de celle des workers */
100             for (int i = 0; i < workers.size(); i++) {
101                 if (i != index && workers.get(i).getAnswerString().get( index: 0).equalsIgnoreCase(opening)) {
102                     openingNb += workers.get(i).getAnswerInt().get( index: 0);
103                     workers.get(i).getAnswerInt().remove( index: 0);
104                     workers.get(i).getAnswerString().remove( index: 0);
105                 }
106             }
107             workers.get(index).getAnswerInt().remove( index: 0);
108             workers.get(index).getAnswerString().remove( index: 0);
109             mostUsedS.add(opening);
110             mostUsedI.add(openingNb);
111             /* Retir les doublons des listes */
112             String lastOpeningAdd = mostUsedS.get(mostUsedS.size() - 1);
113             for (int i = 0; i < workers.size(); i++) {
114                 for (int j = 0; j < workers.get(i).getAnswerString().size(); j++) {
115                     if (workers.get(i).getAnswerString().get(j).equalsIgnoreCase(lastOpeningAdd)) {
116                         workers.get(i).getAnswerString().remove(j);
117                         workers.get(i).getAnswerInt().remove(j);
118                     }
119                 }
120             }
122         //Construction de toReturn
123         String toReturn = "";
124         for (int i = 0; i < mostUsedS.size(); i++) {
125             toReturn += mostUsedS.get(i) + " " + mostUsedI.get(i) + "\n";
126         }
127         System.out.println("Final response = " + mostUsedS + mostUsedI);
128         return toReturn;
129 }
```

Table des matières