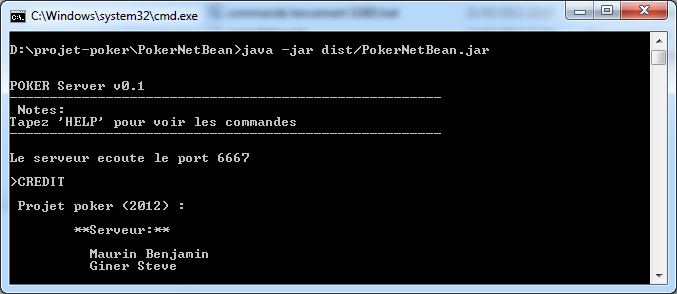
**E) Aspect fonctionnel de la partie serveur**

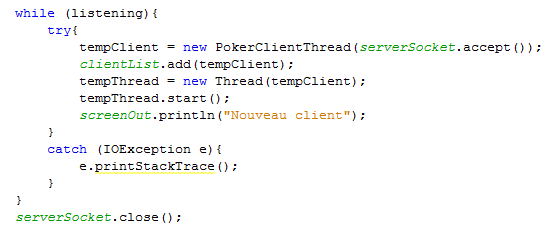
Le Serveur contient onze classes pour environ 3700 lignes de code. Il doit gérer la communication et la gestion des clients, le jeu et les parties, la base de données, et le lecteur de commandes.



1) Communication et gestion des clients

Le serveur, lorsqu’il est lancé, écoute indéfiniment les demandes de connections (via l’objet Socket) des clients via sa classe PokerServer (classe main). Lorsqu’il en réceptionne une, il instancie un nouveau *thread* (classe PokerClientThread) et conserve un pointeur vers cette classe. Lorsque la connexion s’arrête, le *thread* est terminé et le pointeur supprimé.

Voici le code de la boucle qui écoute les demandes de connexion :



Une fois le client connecté, c’est le *thread* créé qui s’occupe d’écouter tous les messages envoyé par la socket du client au serveur. La classe PokerClientThread contient une fonction « traitements » qui découpe les messages reçus et réagit en fonction.

Voici un morceau du code de cette fonction :



Les messages reçus sont traités en fonction du protocole (en annexe). Cependant, pour éviter qu’un client mal intentionné ou mal conçu fasse boguer le serveur, de nombreuses vérifications sont effectués pour que seules les actions possibles pour chaque client soient effectuées.

Par exemple, un client qui n’est pas loggé ne peut pas créer ou rejoindre une partie, un client qui n’est pas le créateur d’une partie ne peut pas la lancer, ect…

De plus, dans le cas de messages erronés ou qui ne font pas partit du protocole, des exceptions sont attendus dans de nombreux endroits critiques du code (à l’aide de try/catch).

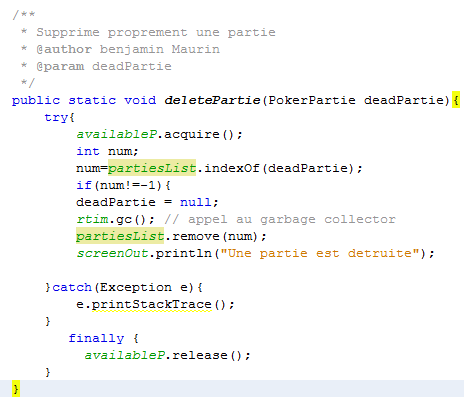
La classe PokerServer est statique. Ainsi, tous les *threads* peuvent accéder à ses fonctions pour par exemple, recevoir la liste des parties qu’un client demande.

Cette classe est aussi celle qui crée des objets de type PokerPartie lorsqu’un client décide d’en créer une. Ces parties sont conservées dans un vecteur (partiesList).

Pour faciliter la communication, les objets PokerPartie contiennent une liste de pointeur vers les classes/*thread* instanciés des clients qui sont dans la partie.

Comme plusieurs threads peuvent, par l’intermédiaire de PokerServer, créer des parties, modifier la base de données, modifier la liste des clients (déconnection), il faut gérer l’exclusion mutuelle. Ainsi des sémaphores sont utilisés pour éviter les erreurs, tel que la double suppression de parties.

Ici un exemple pour la fonction qui permet de détruire une partie :



2) La gestion des jeux

L’ensemble de classes décrites ici doit prendre en charge les évènements du jeu côté serveur comme la donne, l’attente de la réponse d’un joueur, la mise des jetons, etc…

4 Classes sont utilisées pour le déroulement d’une partie :

* PokerPartie : classe principale qui gère le déroulement de la partie (enchères, répartion des jetons, …).
* Jeu : classe permettant de gérer tout ce qui touche aux cartes (paquet, comparaison des mains, …).
* PokerClientThread : classe contenant les paramètres du client (jetons, jetons posés, état, …) et qui permet la communication.
* TimerChoixJoueur : classe permettant d’attendre le choix du joueur pendant 15 secondes.

La partie commence en lançant tournoi() qui lancera déroulement() avec en paramètre l’indice du clients qui doit poser la blind dans la liste clientList.

clientList est composée d’instances de PokerClientThread. Chaque instance représente un client. Cette classe contient différents paramètres, voici ceux qui nous intéresserons pour le déroulement d’une partie :

* ‘int jetons[2]’ : avec en 0 les jetons du joueur et en 1 les jetons posés.
* ‘int cartes[2]’ : avec les 2 cartes de la main du joueur.
* ‘int attente’ : pour savoir l’état du joueur (0 : en jeu, 1 : couché, 2 : tapis, -1 : perdu).
* ‘int joue’ : pour savoir si c’est son tour (0 : non, 1 : oui).
* ‘int pot’ : jetons que le joueur n’a pas le droit de toucher.

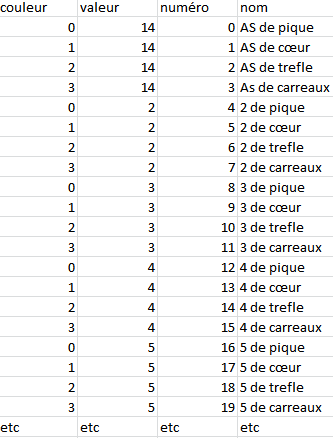
Bien sûr, ce sont des variables de classes privées et ne sont modifiables et utilisables que par des fonctions.

La classe PokerPartie est composé de différentes fonction pour permettre l’envoi des différents messages du protocole utiles au déroulement de la partie.

Au début, de tournoi(), on instancie la classe Jeu pour avoir le paquet de cartes.

Dans Jeu, le paquet de cartes est représenté par une pile d’entier contenant des entiers de 0 à 51 (‘tas’). Chaque entier représente une carte différente. Chaque carte a une valeur et une couleur qui sont des entiers allant respectivement de 2 à 14 et 0 à 3. La valeur et la couleur des cartes sont très rapide d’accès car ses données sont présente dans un tableau d’entiers à deux dimensions ‘vc’ de 52 entrées. Si on veut la couleur de la carte 12 on a donc juste à faire vc[12][1] et on obtient la couleur. De plus, ce tableau n’est initialisé qu’une seule fois par tournoi puisqu’il reste le même et a ce coût est donc négligeable. Des fonctions sont utilisées dans le code pour permettre tout changement rapide de ce système tout en conservant le code des fonctions.

Nous avons donc des cartes que l’on peut représenter comme ceci :



La pile ‘tas’ est mélangée en échangeant chaque entier avec un autre entier du tas parmi les 52 à l’aide de la fonction random() de la librairie Math ce qui nous permet d’avoir un comportement aléatoire dans la répartition des cartes.

A chaque fois que l’on veut tirer une carte, il suffit de dépiler un élément de ‘tas’ et on n’a donc aucun doublon de cartes possible. Cette action s’effectue par le biais de tireUneCarte().

Lorsque que deroulement() commence, il faut initialiser les variables des clients en jeu, les cases du tableau jetonsPose[] et les différentes variables permettant le déroulement du jeu. Ce tableau permet de conserver les jetons posés par chaque client même s’il se déconnecte car c’est nécessaire pour répartir les gains correctement.

On peut remarquer qu’il y a plusieurs fonctions permettant d’effectuer les enchères. Même si elles sont assez similaires au premier coup d’œil, elles sont en fait assez différentes. premièreEnchere() permet d’effectuer la première enchère qui n’a pas de première phase contrairement aux autres et qui doit faire un tour complet de table y compris du joueur qui a mis la surblind alors que c’est le dernier à avoir relancé. Dans une enchère classique on ne retourne pas sur la dernière personne à avoir relancé.

La premierePhase() permet de regarder si quelqu’un relance lorsque l’on vient d’engager un nouveau tour d’enchère et s’il y a relance (ouverture) seulement on lancera enchere() sinon on passera au prochain tour d’enchère.

Lors d’un tour d’enchère on va parcourir les joueurs en jeu (attente==0) et envoyer un message aux clients chacun leur tour pour leur demander de faire un choix d’action. Ces choix sont : passer, suivre et relancer (représenté respectivement par l’entier 1, 2 ou 3). La relance n’est pas toujours possible (3 relance max par tour d’enchère, ouvrir n’est pas une relance), le message contient donc un booléen (‘true’ ou ‘false’) pour indiquer aux clients s’ils doivent proposer ce choix.

Le serveur va se mettre en attente de la réponse du client pendant 15 secondes en instanciant TimerChoixJoueur et en mettant un sémaphore. Si au bout des 15 secondes il n’y a pas de réponse, l’instance lance jouage() avec les paramètres pour passer et donc le joueur se couche (attente=1), ça libère le sémaphore et le programme reprend dans choix(). Si le client envoie une réponse correcte, l’instance du client lance jouage() avec le choix du joueur, ça libère le sémaphore et le programme reprend dans choix() pour appliquer le choix du joueur changer les paramètres et envoyer les modifications aux clients.

Un mauvais choix implique que le joueur se couche. Un mauvais nombre de jetons lors d’une relance est rétabli : s’il y a plus de jetons que ce qu’il possède alors il fait tapis, s’il y en a moins ou le même nombre que le minimum pour suivre alors son choix devient suivre.

A la fin de tous les tours d’enchère, vient la répartition des gains. C’est là qu’intervient la variable ‘pot’ des clients. Pour savoir, combien a le droit de toucher chaque joueur, il suffit de faire jetons de la table moins le pot du client.

La fonction repartionDesGains() permet de répartir les gains entre les clients selon leur statut (gagnant, perdant, tapis).

Pour savoir qui sont les gagnants, l’instance Jeu de PokerPartie appelle la fonction gagnant() qui va comparer les mains de chaque client encore en jeu. Pour ce faire, on va regarder quel est la meilleure combinaison de chaque client en regardant dans cet ordre s’il a :

* Quinte flush royale.
* Quinte flush.
* Carré.
* Full.
* Couleur.
* Suite.
* Brelan.
* Deux paires.
* Une paire.

Pour chaque combinaison, on récupère en même temps la meilleure carte de la combinaison. Puis s’il y a égalité, on va comparer le reste des cartes (je rappelle qu’on ne prend en compte que 5 cartes).

Après la répartition des gains, il faut vérifier si la partie est terminée (plus qu’un seul joueur avec des jetons) et si ce n’est pas le cas relancer un tournoi() continue de boucler et relance le premier tour d’enchère en faisant avancer l’indice du client devant poser la blind.

3) La base de données

Toutes les actions sur la base de données sont gérées via les méthodes de la Classe ClientBDDNoSQL. La classe PokerServer possède un objet de ce type et utilise ses méthodes. MongoDB dispose d’une API java simple et documentée, utilisée ici.

Les méthodes de l’API permettent : de se connecter à la BDD, de faire requêtes de lecture, d’insertions et de modification dans la BDD.

La classe ClientBDDNoSQL possède de nombreuses autres méthodes répondant aux besoins précis envers la BDD qui sont :

-Créer ou effacer un compte pour un joueur.

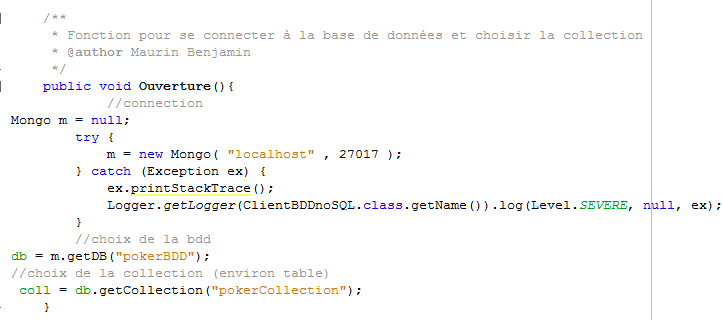
-Changer le pseudo ou le mot de passe d’un compte.

-Vérifier le mot de passe d’un joueur donné.

-Augmenter le nombre de parties gagnées ou perdues.

-retourner les infos d’un ou de tous les Joueurs.

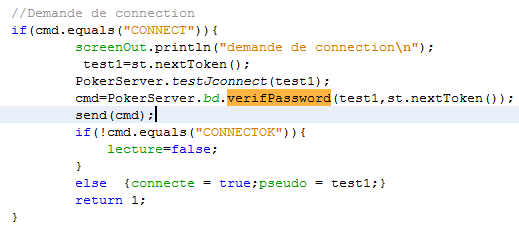
Voici le code de la fonction qui permet de se connecter à la BDD, qui est lancé au démarrage du serveur :



Contrairement aux *API* SQL, ici les requêtes se font en java via de simples fonctions « put » ou « find ». Ceci permet de maitriser plus facilement les requêtes que l’on veut faire.

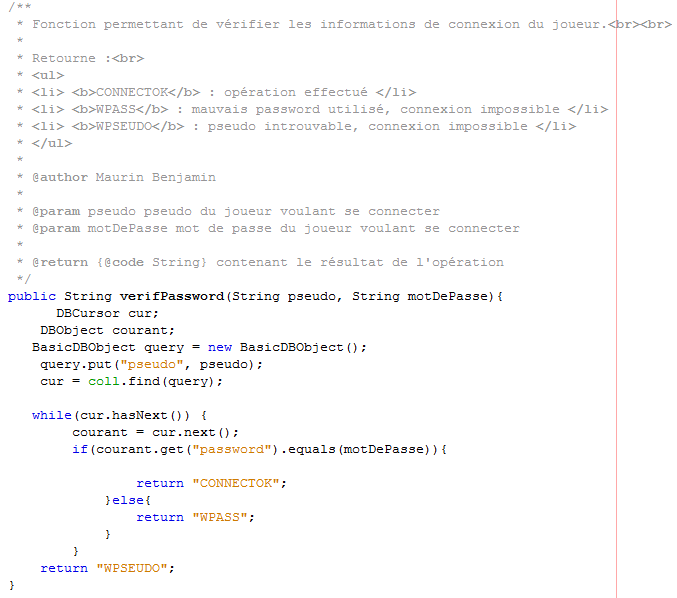
Les méthodes de ClientBDDNoSQL retournent des String correspondant au résultat des requêtes, en fonction du protocole. PokerServer n’a ainsi qu’à envoyer au client ce que retourne les methodes de ClientBDDNoSQL.

Par exemple, lorsqu’un client veut se logger, le *thread* qui réceptionne la demande a ce code :



Dans ce code, l’objet bd de PokerServer est une instance de ClientBDDNoSQL précédemment connecté au SGBD NoSQL.

Voici la méthode verifPassword de cet objet :



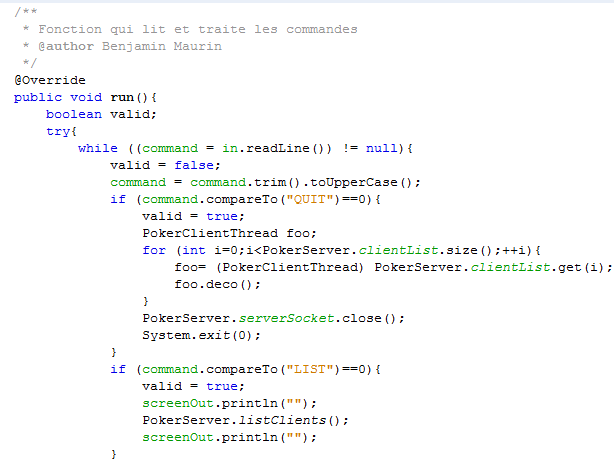
Ici, une requête est créée associant le message pseudo avec la variable pseudo passée en paramètre. Le DBCursor cur permet de parcourir les résultats de cette requête (via la fonction .next()). Chaque résultat est de la forme DBObject qui est l’équivalent d’un tuple en SQL. Il suffit alors de comparer l’attribut « password » avec celui fournit par le client.

4) Le lecteur de commandes

Le lecteur de commande doit être actif à tout moment. Pour cela, La classe CommandReader (qui est un *thread*) est lancée au lancement du serveur, et elle boucle sur l’écoute des commandes de l’utilisateur.

Comme pour le traitement des messages, cette classe découpe la commande pour effectuer la bonne action et utilise les fonctions statique de PokerServer pour les mener à bien.

Voici un morceau de la fonction qui traite les commandes :



5) Diagrammes de l’application :

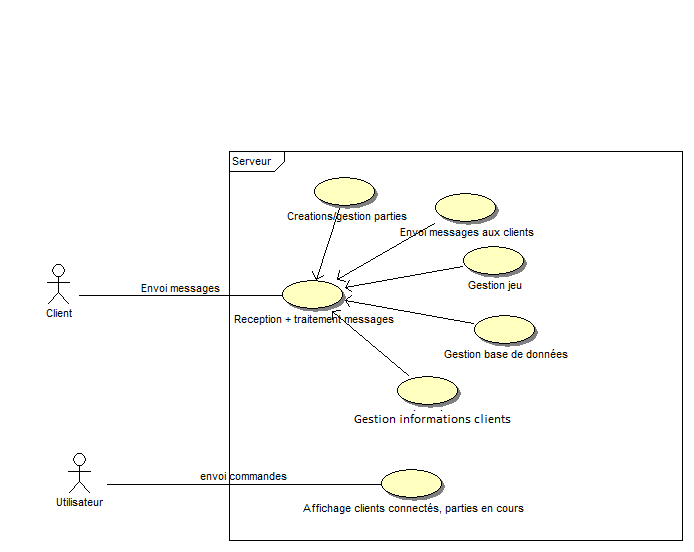
Diagramme des cas d’utilisation :

Diagramme des classes : 