ANCA

Scrabble

Rapport n°1

Debecker Bernard
Foncier Romain
Morel Arnaud

Jan 2013

A. Introduction

Dans le cadre du cours *d'ANalyse et Conception d'Application* dispensé en dernière année de bachelier en informatique, il a été proposé aux étudiants de développer une application client/serveur du célèbre jeu de Scrabble.

Le développement est à réaliser par groupe de 2 à 3 personnes. Le notre est composé des trois membres suivants : Bernard Debecker, Romain Foncier et Arnaud Morel.

Ce rapport a pour but de présenter notre travail. Il comportera quatre parties : la première décrira le travail d'analyse mis en place avant et pendant le développement. Les deux parties suivantes décriront respectivement la structure établie côté client, puis côté serveur. Enfin, une dernière partie présentera la nouvelle architecture MVC instaurée au sein du projet.

B. Architecture de l'application

Afin de démarrer notre développement sur une structure saine, nous nous sommes appuyés sur l'architecture mise en place au sein des projets *BanqueUnGab* et *BanqueUnServeur*. Ceci nous a permis de rapidement faire communiquer nos deux applications afin de n'avoir à se focaliser que sur l'essentiel. Plusieurs questions ont donc été posées :

- Quels éléments doivent être créés ? De quel côté ?
- Quelles informations doivent être échangées ? Comment ?
- Quels processus doivent être effectués uniquement côté client et réciproquement côté serveur ? Lesquels doivent être effectués des deux côtés ?

Dans un premier nous nous sommes uniquement intéressés aux éléments du jeu afin de déterminer les classes à créer et leurs interactions. Voici la liste des principales classes du Scrabble :

- *Player* : Sans lui personne ne joue
- *Tiles* : indispensables au jeu
- *Gameboard* : lui même composé des sous-éléments suivants :
    - *Grid* : matrice sur laquel le joueur place des tiles
    - *Rack* : sequence de tiles que le joueur peut placer

- *Play* : élément essentiel représentant une partie. Il regroupe les informations du joueur et du GameBoard.

Le GameBoard est présenté ici à titre informatif. Une classe GameBoard ne présentant pas d'intérêt.

Il est évident que chacun de ces éléments devront être présents à la fois côté client et côté serveur dans la mesure où ils reflètent à tout moment l'état du jeu (chaque côté devant être synchronisé).

Puis nous avons étoffé ces classes de base afin d'offrir des fonctionnalités supplémentaires :

- Création d'un compte joueur
- Connexion
- Sauvegardes des parties
- ...

Celles-ci seront abordées plus en détails dans leur partie respective.

i. Architecture Client/Serveur

Le diagramme de la **Figure 1** représente la structure générale de l'architecture client/serveur mise en place au sein de notre projet. On y retrouve des deux côtés les classes du jeu, ainsi que les classes chargées des protocoles de communication.

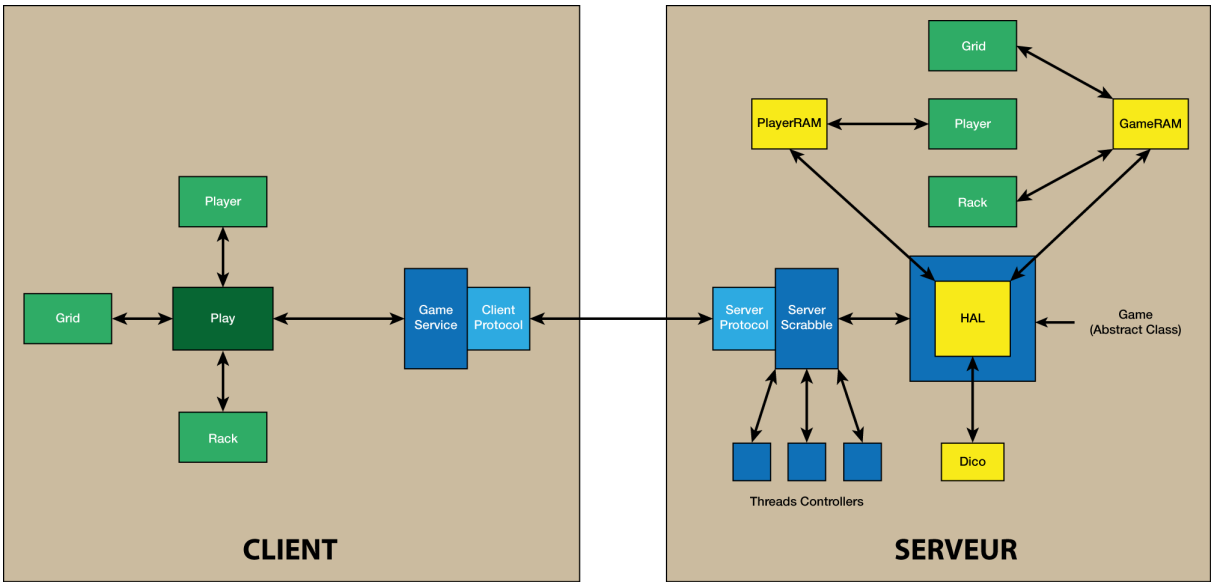

Figure 1 - Architecure Client/Serveur

Afin d'illustrer avec précisions le déroulement des échanges entre les deux parties, nous allons présenter le cas d'utilisation suivant :

- Initialisation d'une partie

***Remarque** : Nous ferons abstraction de la représentation visuelle du jeu lors de ces étapes dans l'unique but de nous concentrer sur la structure mise en place. Les vues seront abordées dans la dernière partie introduisant l'architecture MVC.*

1. Initialisation d'une partie

Lors du lancement de l'application, une instance de *Play* est créée. Celle-ci va à son tour créer une instance du *GameService* (classe lui permettant d'interagir avec le serveur), qui va lui même créer un instance de *ClientProtocol* (classe responsable des échanges avec le serveur). Toutefois jusque là, aucune instance de *Player*, *Grid*, *Rack* ou *Tile* n'est encore créée.

Etape 1 - Envoi d'une requête (côté client)

L'initialisation du jeu démarre côté client, lorsque le joueur fait appel à cette action (via un bouton ou menu console, ...).
Comme nous l'avons indiqué précédemment, notre application permet au joueur de créer un compte et de s'y connecter afin de pouvoir sauvegarder ses parties. Mais il est également possible de lancer directement une partie anonyme (celle-ci ne sera pas sauvegardée). Ce joueur sera alors considéré lui aussi comme anonyme. C'est le scénario que nous allons décrire ici.

L'initialisation va faire appel à une méthode de notre instance *Play* pour lancer l'initialisation d'une partie anonyme : *playAsGuest()*

```
public void playAsGuest() {
  String [] response = null;
  player = new Player();
  try {
    response = service.createNewPlay(player.getPlayerID(), true);
  } catch (GameException ge) {}
  initPlay(response[0], "", response[1], 0);
  ...
}
```

Cette méthode va créer un nouveau joueur, puis via la méthode *createNewPlay(String playerID, boolean anonymous)* de son instance de GameService va tenter de contacter le serveur.

```
public String [] createNewPlay(String playerID, boolean anonymous) throws
GameException {
  String [] args = null;
  Message serverResponse = (anonymous) ?
servProtocol.sendRequest(Message.NEW_GAME_ANONYM,  playerID) :
servProtocol.sendRequest(Message.NEW_GAME,  playerID);
  if (serverResponse != null) {
    switch(serverResponse.getHeader()) {
      case Message.NEW_GAME_SUCCESS:
        args = new String(serverResponse.getBody()).split("##");
        return args;
      case Message.NEW_GAME_ANONYM_SUCCESS:
        args = new String(serverResponse.getBody()).split("##");
        return args;
      default:
        exceptionTriggered(serverResponse.getHeader());
    }
  } else {
    throw new GameException(GameException.typeErr.CONN_KO);
  }
  return args;
}
```

Cett étape nous permet d'introduire deux nouvelles classes : *Messages* et *GameException*. La première va nous permettre d'identifier les actions à effectuer tout en contenant requête ou reponse. Elle est donc présente des deux côtés. La deuxième va nous permettre de gérer les erreurs. Elle est présente des deux côtés, mais elle peut contenir certaines spécificités propres à chaque partie. En effet, le client et le serveur ne vont pas toujours réagir de la même manière face au même type d'erreur.

Le GameService fait appel à la méthode *sendRequest(int header, String args)* de son instance du *ClientProtocol*.

Dans le cas d'une partie anonyme, le header sera égale à *Message.NEW_GAME_ANONYM*, une valeur entière permettant d'identifier l'opération à effectuer. Le deuxième paramètre *args* prendra la valeur de l'*IDPlayer* (créé lors de la construction de l'objet *Player* dans l'intention de pouvoir l'identifier facilement).

Nous avons donc décrit ici le démarrage du processus d'initialisation du jeu côté client. La deuxième partie se déroule côté serveur.

Etape 2 - Réception de la requête et envoi de la réponse (côté serveur)

Avant de présenter le déroulement de ce processus, une description plus précise de la partie serveur s'impose.

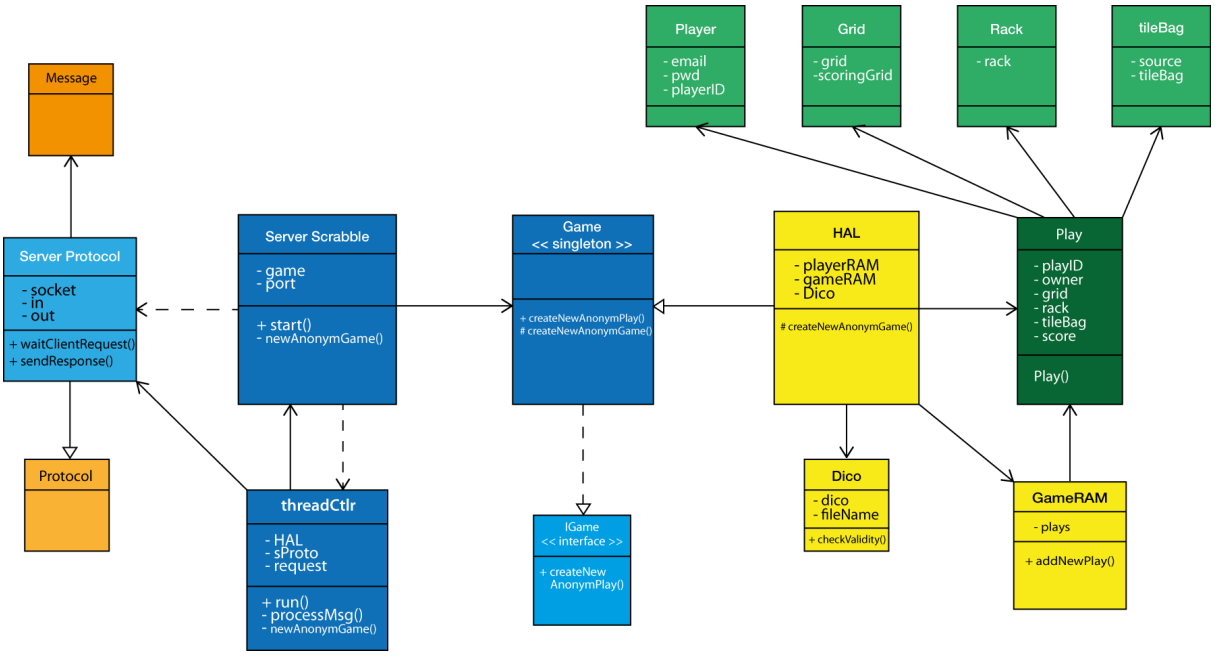

Figure 2 - Diagramme de classes logicielles (côté serveur)

Au lancement de l'application serveur, une instance de *ScrabbleServer* est créée. Le constructeur de cette classe va faire appel à la méthode statique *getGame()* de la classe *GameFactory* afin d'obtenir une instance de *Game* (Singleton). Cette dernière est à la base de toutes les opérations inhérentes au jeu et à l'application. Une fois l'instance de *Game* obtenu, le *ScrabbleServer (ScS)* va rester à l'écoute d'une potentielle connexion cliente en créant un ServerSocket (port d'écoute).

Poursuivons le scénario envisagé précédemment. Le *ScS* reçoit la demande de connexion envoyée par le *ClientProtocol* de l'application cliente. Il va alors créer une instance de *ServerProtocol* afin d'être en mesure de communiquer correctement avec celle-ci. Puis il va créer un nouveau thread *ThreadCtlr* qui aura à sa charge le traitement de la requête du client. Cette étape est importante car elle va permettre à l'application serveur de pouvoir gérer plusieurs clients en même temps. A chaque nouvelle demande de connexion, un nouveau *ThreadCtlr (TC)* est créé. Chacun partageant une même référence du *ScS*.

Pourquoi? Pour limité la redondance de références vers les classes du modèle. Seul le *ScS* a une référence de la classe *Game*. De cette manière, si deux *TC* traitent une même requête au même moment (i.e initialiser le jeu),

ils seront en concurence lorsque chacun fera appelle à la même méthode du ScS. Les méthodes du *ScS* sont donc toutes "*synchronized*".

Une fois le *TC* créé, se dernier va récupérer la requête envoyée par le client via son instance de *ServerProtocol*. Il va séparer les informations reçues , puis appellera la méthode adéquate du *ScS* pour le traitement. Ce dernier fera appel à son tour à la méthode adéquate de *Game*. Pour le scénario envisagé précédemment, il s'agira de la méthode *createNewAnonymPlay()*.

```
@Override
public Message createNewAnonymPlay(String pl_id) throws GameException {
  Message response = createNewAnonymGame(pl_id);
  switch (response.getHeader()) {
    case Message.NEW_GAME_ANONYM_SUCCESS:
      return response;
    case Message.NEW_GAME_ANONYM_ERROR:
      throw new GameException(GameException.typeErr.NEW_GAME_ANONYM_ERROR);
  }
  return null;
}

protected abstract Message createNewAnonymGame(String pl_id);
```

Game regroupe toutes les opérations concernant le jeu : placer un mot, échanger les tiles, ... Mais elle a également la charge des opérations propres à l'application : Gestion des joueurs, des parties et des connexions. Il s'agit d'une classe abstraite qui va être étendue et spécialisée pour notre jeu de Scrabble par la classe *HAL*. C'est ensuite la méthode abstraite *createNewAnonymGame()* (implémentée au sein de la classe *HAL*) qui se chargera de traiter la requête.

```
@Override
protected Message createNewAnonymGame(String pl_id) {
  if (!plays.playerIsLogged(pl_id)) {
    plays.addPlayer(pl_id);.
    Play newPlay = new Play(pl_id);
    plays.addNewPlay(pl_id, newPlay);
    System.out.println("Rack : "+newPlay.getFormatRack());
    return new Message(Message.NEW_GAME_ANONYM_SUCCESS,
newPlay.getPlayID()+"##"+newPlay.getFormatRack());
  }
  return new Message(Message.NEW_GAME_ANONYM_ERROR, "");
}
```

Deux nouvelles classes sont à présenter à ce niveau : *playerRAM* et *GameRAM*. Le première est chargée de gérer les joueurs en cours. Ainsi un joueur déjà connecté ne pourra disputer de parties (sauf si jeu anonyme) sur un autre client. La seconde classe aura la responsabilité de gérer toutes les parties en cours : anonymes et loguées.

Le joueur étant anonyme dans notre cas, on l'ajoute à notre liste de joueurs (*Cette étape n'est pas nécessaire et sera supprimée ultérieurement*), puis on créer une nouvelle partie. C'est à ce niveau que grid, rack, tileBag et score sont initialisés côté serveur. Un *UUID* (Universally Unique IDentifier) est généré pour cette nouvelle partie et l'ID du joueur y est associé afin de pouvoir aisément identifier ce dernier et la partie qu'il dispute. L'instance de *Play* nouvellement créée est ensuite ajoutée à la liste de parties du *GameRAM*. Si tout s'est déroulé sans erreurs, le header *NEW_GAME_ANONYM_SUCCESS* ainsi que la liste des tiles contenu dans le rack seront envoyés sous la forme d'un *Message*. Le cas échéant, seul un header correspondant à une erreur sera retourné.

Retour à la méthode *createAnonymePlay()* de la classe *Game*. On va dès lors vérifier la valeur du header retourné par la classe *HAL*. En cas de succès, le Message (contenant header et data) est renvoyé par le *TC* au client. En revanche, si une erreur c'est produite, une exception *GameException* sera lancée et traitée par le *ScrabbleServer*. Ce dernier pourra afficher l'erreur à la console ou l'inscrire dans un fichier de logs. Puis il renverra un Message avec une valeur de header appropriée, afin que la réponse soit traitée convenablement par le client.

Comment sont renvoyées les données ? Pour permettre au client d'identifier et d'initialiser la nouvelle partie créée, l'IDPlay ainsi que le contenu du Rack (serveur) seront formatés de la manière suivante :

IDPlay##L:V=L:V= ...

Etape 3 - Réception de la réponse (côté client)

Nous sommes à nouveau côté client au sein de la méthode *createNewPlay()* de la classe *GameService*. La réponse fournit par le serveur sera retournée à cette méthode sous la forme d'un *Message*. Encore une fois le header sera contrôlé afin de déterminer si une erreur a eu lieu ou non. Si la valeur de ce dernier est égale à celle de la variable *NEW_GAME_ANONYM_SUCCESS*, l'IDPlay ainsi que le rack formaté seront retournés. Dans le cas contraire, une *GameException* sera lancé et le client se chargera d'informer le joueur qu'un problème est survenu.

Remarque : une fois la communication terminée, le ThreadCtlr sera interrompu.

```
public void playAsGuest() {
  ...
  try {
    response = service.createNewPlay(player.getPlayerID(), true);
  } catch (GameException ge) {}
  initPlay(response[0], "", response[1], 0);
  ...
}
```

Retour à la classe *Play*. Nous allons initialiser le jeu côté client via la méthode *initPlay()* :

```
public void initPlay(String playID, String formatedGrid, String formatedRack, int
score) {
  this.id = UUID.fromString(playID);
  this.score = score;
  grid = (formatedGrid.equals("")) ? new Grid() : new Grid(formatedGrid);
  rack = new Rack(formatedRack);
}
```

Nous allons donc initialiser les éléments suivants : *IDPlay*, *score*, *grid* et *rack*. Ici le grid est vide puisqu'il n'y a aucune tile à charger (à la différence d'une partie sauvegardée). Il est donc initialisé avec une matrice 15x15 vide. Le rack en revanche sera chargé à partir des données envoyées par le serveur. C'est à ce niveau que la classe *Tile* entre en jeu. Le Rack contenant désormais une séquence de 7 tiles, mises à disposition du joueur.

On constate que chaque côté possède les mêmes données :

- Player
- Play (Player, Grid, Rack)

Client et *Serveur* sont donc synchronisés.

Diagramme d'interraction de ce scénario

Afin d'illustrer de manière plus précise ce processus tout en incluant la dimension temporelle. Nous allons présenter le diagramme d'interraction des éléments participant à l'initialisation du jeu. Il permet de mettre en avant le rôle joué par chaque partie et de faciliter la visualisation du déroulement des différentes étapes.

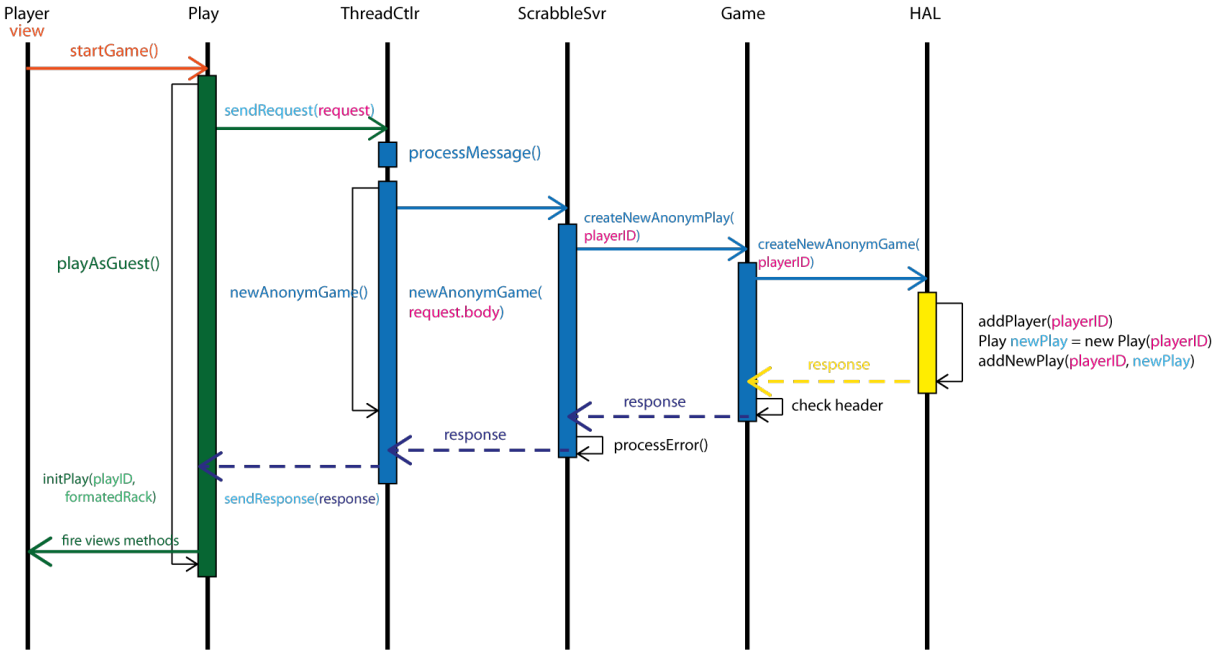

Figure 3 - Diagramme d'interraction de l'initialisation d'une partie anonyme

C. Application Cliente

D. Application Serveur

E. Architecture MVC