Projet Java : Network game

TP3 - Projet Java

Ajout de la couche réseau

Jérôme Buisine
Florian Leprêtre

jerome.buisine@univ-littoral.fr
florian.leprêtre@univ-littoral.fr

12 avril 2021

Durée : 6 heures

La finalité de la suite des TPs proposés, est de concevoir un système permettant à des utiliseurs de jouer à des jeux cartes en réseau. Les premiers jeux de cartes développés seront le jeu de Bataille classique ainsi qu'une version simplifiée du jeu de Poker.

1 Travail

L'objectif de ce TP est de mettre en place l'interaction réseau de notre système de jeu. Un serveur hébergera un jeu et des clients (joueurs) pourront s'y connecter pour y jouer.

1.1 Objectifs

L'objectif global est de mettre en place un système de jeu où chaque joueur peut se connecter à un serveur de jeu. Un serveur de jeu sera donc lié à un jeu qu'il héberge. Une fois le nombre de joueurs connectés attendu pour le jeu, le jeu peut démarrer.

Précédemment, nous avons développé notre framework de jeu permettant de simuler le jeu de la Bataille et du Poker. La limite est telle que le jeu se joue depuis le même programme uniquement. Difficile donc quand on joue au Poker de ne pas être tenter de tricher ! La mise en place de la séparation sous forme client / serveur va donc être développée dans le cadre de ce TP.

La difficulté ici résidera dans la manière de communiquer du serveur aux différents joueurs (clients) ayant rejoint le jeu et inversement, qu'un joueur puisse communiquer son intention de jeu au serveur quand c'est à lui de jouer. On rajoutera ici le fait que chaque joueur doit prendre connaissance du fait qu'un autre joueur ait réalisé une action de jeu.

1.2 Environnement de travail

L'environnement de travail sera le même que les TPs précédents :

— 1. IntelliJ/Idea (version Ultimate) comme environnement de développement (ou Eclipse en fonction de votre préférence) ;

— 2. Le langage Java avec la version 8 de Java SE ;

— 3. Git comme système de versionning du projet ;

— 4. Gitlab comme serveur d'hébergement de votre projet **Git**.

1.3 Récupération de la correction et d'exemples

En suivant le tutoriel suivant, il vous sera possible de récupérer la correction du TP précédent. Il vous faudra pointer sur la branche TP2.

En plus de la correction, deux autres classes vous seront fournis à titre d'exemple d'interaction client / serveur. Les classes `ulco.cardGame.client.Client` et `ulco.cardGame.server.SocketServer` qui montrent un exemple de communication.

Si vous lancez le serveur puis le client, et que vous continuez les instructions demandées au client, vous risquez de rencontrer cette erreur :

```
java.io.NotSerializableException:*
```

Nous verrons comment régler ce problème par la suite !

2 interaction réseau

Avant d'aller plus loin, faisons un petit récapitulatif de l'attendu au niveau de la connexion client / serveur dans notre contexte de jeu de plateau.

2.1 Communication du serveur et des joueurs

Les différentes étapes du déroulement d'un jeu en ligne sont détaillées dans la Figure 1. Voici quelques explications supplémentaires :

— Un serveur lancé, devra attendre la connexion d'un nombre de joueurs attendus (comme cela était déjà le cas) ;

— À chaque nouvelle connexion, le joueur est ajouté. On informera le nouveau joueur du nom de l'instance du jeu actuel hébergé par le serveur. Un message du serveur aux joueurs déjà connectés, les informe d'un nouveau joueur connecté ;

— Une fois le nombre de joueurs présents atteint, le serveur va envoyer un message contenant l'état du jeu ;

— Le serveur précisera au travers d'un message spécifique que l'action d'un joueur est attendu (l'action de jouer) ;

— À chaque action de joueur récupérée, il faudra fournir à nouveau un nouvel état du jeu à tous les joueurs (répercution de l'action du joueur sur le jeu) ;

— La boucle d'interaction de jeu sera ensuite faites classiquement mais en exploitant ce principe d'échanges d'information entre les joueurs et le serveur.

Il faut savoir que dans notre cas, nous allons utiliser un système d'échange de données Serveur / Client, à base de **Socket**. Cela permet de créer une connexion entre un client et un serveur distant afin de réaliser des échanges d'informations via des *messages*. Une API pour les sockets est disponible dans Java. Nous verrons dans la suite du TP comment l'exploiter.

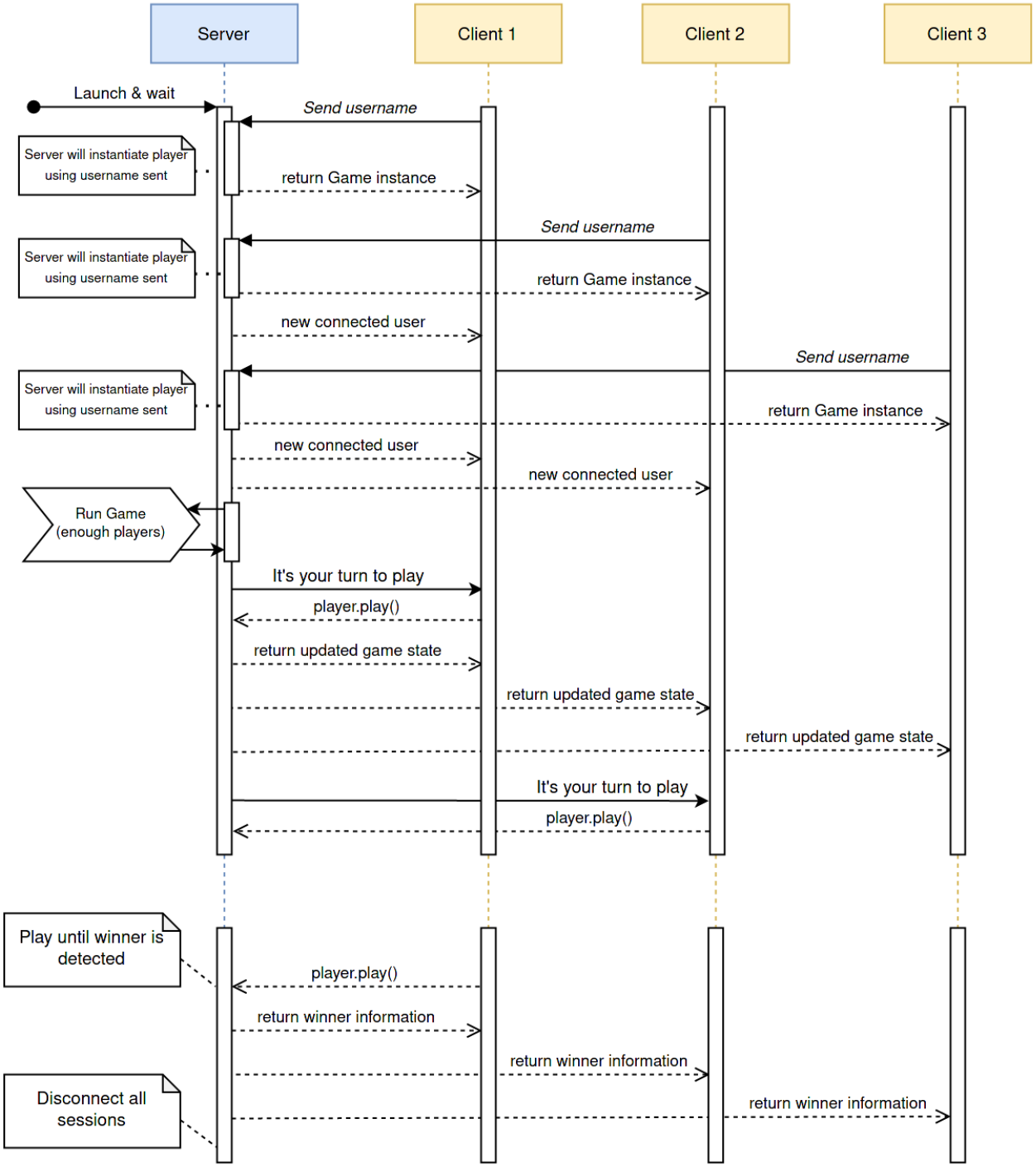


FIGURE 1 – Diagramme de séquence attendu entre clients et serveur

2.2 Compréhension d'un exemple de communication

Comme déjà précisé, un exemple de communication Socket via l'API Java, est fourni au travers des classes **`ulco.cardGame.client.Client`** et **`ulco.cardGame.server.SocketServer`** (voir branche TP2 du projet).

Pour faire intéragir les deux classes, il faut dans un premier temps lancer le programme principal du serveur, soit la classe **`SocketServer`**. Cela permet de lancer le serveur qui attendra l'arrivée de joueurs. Puis de lancer une ou plusieurs instances de **`Client`**, afin que le serveur initialise les connexions entrantes des joueurs.

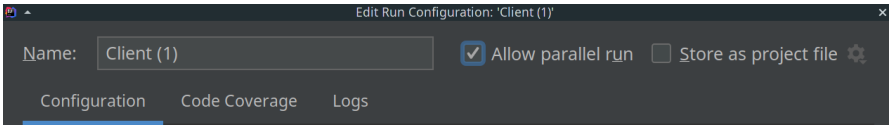
2.2.1 Résolution de l'encodage de données

Ce code n'est pour le moment pas fonctionnel. Lors de l'exécution du programme principal, une exception de type **`java.io.NotSerializableException:*`** est levée. Comme précisé, l'échange d'information se fait via un message. L'exemple proposé, souhaite échanger des objets directement et la Java Virtual Machine (JVM) semble lever cette erreur.

Pour qu'une instance de classe soit passée en tant que message Socket, il faut qu'elle soit *serializable*. C'est-à-dire qu'elle soit encodée dans un format particulier qui permettra sa **reconstruction d'instance** à la réception de son message. Pour permettre cela, nous allons ajouter l'implémentation de l'interface Java **`Serializable`** disponible dans l'API Java. Cette interface peut être soit implémentée directement via une classe, soit être héritée d'une interface.

Travail : à partir des indications précédentes, faire hériter les interfaces ou implémenter les classes de l'interface **`Serializable`** afin que les échanges d'objets spécifiques via une connexion Socket soit possibles. Vérifier que le lancement des applications **`SocketServer`** et **`Client`** soient maintenant fonctionnelles.

Attention : il vous faudra sûrement configurer le fait que votre programme Client puisse être instancié plusieurs fois. Cela est possible dans la configuration du programme, en cochant **`Allow parallel run`** :



2.2.2 Compréhension de l'API Socket Java

Dans les exemples disponibles, on peut distinguer deux classes de Socket utilisées. La classe orientée serveur, **`java.net.ServerSocket`**, qui permet d'instancier un serveur de Socket. Une instance de cette classe pourra accepter ou non des connexions entrantes. La classe **`java.net.Socket`**, qui permet de créer une connexion vers le serveur qui sera acceptée ou non. Cette instance a besoin de connaître l'adresse IP du serveur (par defaut on utilisera le localhost) ainsi qu'un port d'accès fixé par l'exemple à 7777.

Voici quelques exemples de code :

```
// ServerSocket instance with port 7777
ServerSocket server = new ServerSocket(7777);

// Set server in waiting connection mode
Socket socket = server.accept();

// Creation of Client socket using Server hostname and port
Socket socket = new Socket(host.getHostName(), CardGameServer.PORT);
```

Lors de la création de l'instance du Serveur, celui-ci va attendre la connexion entrante d'un client via sa méthode `accept`. Lorsqu'un client émet une connexion Socket entrante, cette méthode va permettre de récupérer l'instance Socket liée à ce Client. C'est grâce à cette instance de Socket récupérée, que le serveur va pouvoir communiquer avec le Client (et inversement).

2.2.3 Échange d'informations via Socket

Afin d'échanger des informations entre le client et le serveur, nous allons utiliser des classes permettant de gérer les **flux** de messages de types Input/Output.

Exemple d'envoie de message via une instance Socket (Server → Client ou Client → Serveur) :

```
ObjectOutputStream oos = new ObjectOutputStream(socket.getOutputStream());
oos.writeObject(game);
```

Exemple de lecture de message via une instance Socket (Server → Client ou Client → Serveur) :

```
ObjectInputStream ois = new ObjectInputStream(socket.getInputStream());
String playerUsername = (String) ois.readObject();
```

Note : il faut savoir que lorsqu'un message Socket souhaite être récupéré via le flux d'entrée (InputStream), le programme est en attente d'un tel message.

2.2.4 Gestion des messages Socket

Comme chaque message est en réalité une instance de classe. C'est-à-dire au mimimum une instance de classe `String`, voire une instance spécifique sérialisée. Il faut pouvoir traiter le message en conséquence. Il sera donc possible de réaliser une action spécifique en fonction du type de message (type d'instance).

Voici un exemple de gestion d'un message en fonction de son type :

```
// Read and display the response message sent by server application
ObjectInputStream ois = new ObjectInputStream(socket.getInputStream());
answer = ois.readObject();

// depending of object type, we can manage its data
if (answer instanceof String)
  System.out.println(answer.toString());
```

Attention : même si l'on *sérialise* un objet, **on ne récupère pas sa référence**. Il faudra donc faire attention à la comparaison d'objet notamment pour la gestion de composants de notre jeu. Pour cela, il est recommandé d'utiliser le champs unique `id` disponible pour chaque instance. Il permet d'identifier le composant en question. Un joueur sera quand à lui identifier par son nom.

3 Intégration de la couche réseau au framework

Afin d'intégrer cette notion d'échange d'informations via le réseau dans notre framework de jeu, nous allons définir quelques règles de gestion. Vous pouvez également vous référer au diagramme de classes attendu pour ce TP pour une meilleure compréhension. La classe `CardGameServer` du diagramme UML, fait ici référence au serveur de Socket.

Voici quelques indications d'intégration de Socket :

— Un joueur va tenter de se connecter à un jeu en envoyant au serveur un message comprenant son pseudo de jeu ;

— La méthode d'ajout d'un joueur d'un jeu, prendra également en paramètre l'instance Socket actuellement traitée (connexion entrante d'un joueur encore non ajoutée). Cela permettra d'informer le joueur si son ajout est possible et le type d'erreur (pseudo déjà existant ou nombre de joueurs maximum déjà atteint) ;

— Le serveur va stocker l'ensemble des joueurs (dont l'ajout est valide) et leurs connexions entrantes (instance Socket) dans un dictionnaire (Map), dont la clé sera le joueur et la valeur l'instance Socket ;

— La méthode **`run`** d'une instance de jeu, prendra maintenant en paramètre le dictionnaire liant chaque joueur à son instance socket (dictionnaire précédemment créé) ;

— La méthode **`play`** d'un joueur prendra en paramètre une instance de Socket et sera maintenant de type `void`. C'est l'instance de Socket qui transmettra un composant (qui sera sérialisé) joué par le joueur ;

— Pour qu'un joueur puisse identifier que c'est à lui de jouer, le serveur lui transmettra un message de type `String`, qui commencera par **[Link]**, où **Link** ici, et le pseudo du joueur qui doit jouer. Le serveur attendra ensuite la réponse du joueur ;

Exemple de messages lors du jeu entre le client et le serveur :

```
Please select your username:
Hubert
CardGame{name='Battle'}
----------------------------------------
--------------- Game State ---------------
----------------------------------------
BoardPlayer{name='Jean', score=18}
BoardPlayer{name='Michel', score=17}
BoardPlayer{name='Hubert', score=17}
----------------------------------------
Waiting for Jean to play...
Player Jean has played CQ
Waiting for Michel to play...
Player Michel has played CJ
[Hubert] you have to play...
Player Hubert has played DQ
-------------- Board state -------------
CQ played by Jean
CJ played by Michel
DQ played by Hubert
----------------------------------------
Player: CardPlayer{name='Jean', score=17} won the round with Card {name='CQ'}
```

Travail : à partir de l'ensemble des indications précédentes, intégrer la couche réseau au jeu de la Bataille. Il vous faudra tout de même adapter les signatures de méthodes du jeu de Poker mais le développement sera priorisé sur le jeu de la Bataille.

4 Bonus

Si vous avez correctement défini la couche réseau de votre de jeu de Bataille, il vous est proposé de :

— Intégrer la couche réseau également pour le jeu du Poker ;

— Faire en sorte que le jeu (Bataille ou Poker), puisse être sélectionné au démarrage du Serveur.