

8INF957 - Programmation objet avancée - Automne 2022



Université du Québec
à Chicoutimi

Projet de fin de session :

**Conception d'une application basée sur
l'architecture client-serveur**

Rédigé par : NGABMEN NJAWAT Benjamin – NGAB27020100

Superviseur : Hamid Mcheick

Table des matières

Introduction	3
Description du problème	3
Approche existante	4
Qu'est-ce que l'architecture client-serveur ?	4
Les types d'architecture client-serveur	5
Avantages et inconvénients de l'architecture client-serveur 2 tiers	5
C'est quoi les sockets ?	6
Solution proposée	6
Structure client-serveur	6
Autres ajustements	8
Démonstration	9
Conclusion	11
Références	11

Introduction

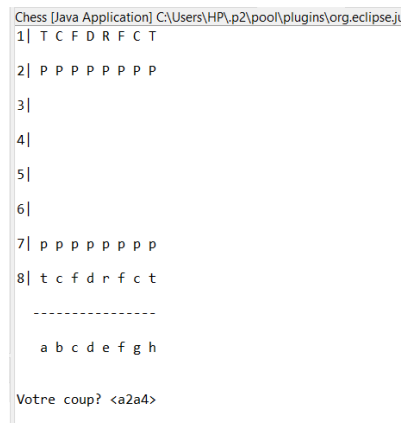
Les applications informatiques sont en constante évolution. Au fur et à mesure de leur existence, de nouvelles fonctionnalités se voient être implémentées dans l'optique de répondre au mieux aux besoins des utilisateurs. Cette évolutivité du code doit donc être prévue à la base dès la création de l'application afin de faciliter les différentes mises à jour. Tout au long de ce document, nous implémenterons une architecture client-serveur au sein d'une application, afin de la faire évoluer. Nous aurons à appliquer quelques principes de conception afin d'améliorer notre application sans pour autant altérer son fonctionnement déjà en vigueur.

Description du problème

La mission que l'on va accomplir est de modifier l'architecture d'une application afin d'y ajouter de nouvelles fonctionnalités. L'application sur laquelle on travaillera est un **jeu d'échecs** codé en Java. Il a été développé par l'Université du Québec à Chicoutimi dans le cadre de travaux pratiques. Le lien de téléchargement du fichier source est le suivant :

<https://moodle.uqac.ca/mod/resource/view.php?id=654110>

Une fois exécuté, ce code permet de jouer une partie d'échecs contre une IA rudimentaire.



```
Chess [Java Application] C:\Users\HP\p2\pool\plugins\org.eclipse.j
1| T C F D R F C T
2| P P P P P P P P
3|
4|
5|
6|
7| P P P P P P P P
8| t c f d r f c t
-----
a b c d e f g h
Votre coup? <a2a4>
```

Image 1- Capture d'une partie d'échecs

Notre objectif est d'implémenter un nouveau mode de jeu dans l'application : **le mode multi-joueur**. Concrètement, un joueur sera à mesure de :

- Créer une nouvelle partie multijoueur et partager le numéro de son salon de jeu
- Rejoindre une partie à partir d'un numéro de salon
- Affronter un adversaire en ligne en jouant à tour de rôle

L'application devra bien sûr être rétroactive, c'est-à-dire, le mode de fonctionnement existant devra toujours être fonctionnel. Nous ne devrons **modifier aucun bout de code** de l'application et juste étendre les fonctionnalités, conformément au principe **Open-Close** des principes SOLID.

Approche existante

Afin de développer le mode de jeu multijoueur de l'application, il faudrait adopter une nouvelle architecture logicielle. Etant donné que le mode multijoueur inclut l'interaction de plusieurs clients, une architecture **client-serveur** est tout à fait indiquée ; et pour assurer la communication entre clients et serveur, l'utilisation des **sockets** est plus que primordiale.

Qu'est-ce que l'architecture client-serveur ?

L'**architecture client/serveur** désigne un mode de communication entre plusieurs ordinateurs d'un réseau qui distingue un ou plusieurs postes clients du serveur : chaque logiciel client peut envoyer des requêtes à un serveur. Un serveur peut être spécialisé en serveur d'applications, de fichiers, de terminaux, ou encore de messagerie électronique.

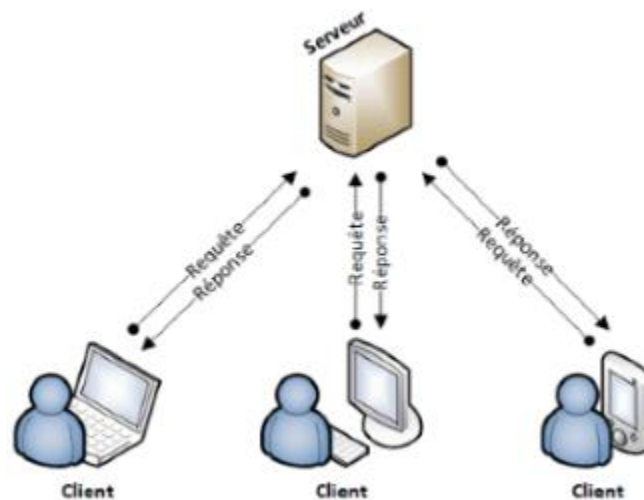


Image 2- Schéma illustratif de l'architecture client-serveur

Les caractéristiques d'un serveur sont :

- Il est passif (ou maître) ;
- Il est à l'écoute, prêt à répondre aux requêtes envoyées par des clients ;
- Dès qu'une requête lui parvient, il la traite et envoie une réponse.

Les caractéristiques d'un client :

- Il est actif (ou esclave) ;
- Il envoie des requêtes au serveur ;
- Il attend et reçoit les réponses du serveur.

Le client et le serveur doivent bien sûr utiliser le même protocole de communication. Un serveur est généralement capable de servir plusieurs clients simultanément.

Les types d'architecture client-serveur

On distingue différents types d'architecture client-serveur :

- Si toutes les ressources nécessaires sont présentes sur un seul serveur, on parle **d'architecture à deux niveaux ou 2 tiers** (1 client + 1 serveur).

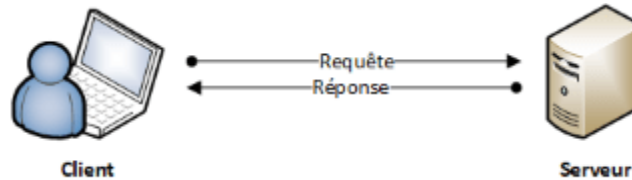


Image 3- Schéma d'architecture 2 tiers

- Si certaines ressources sont présentes sur un deuxième serveur (par exemple des bases de données), on parle **d'architecture à trois niveaux ou 3 tiers** (1 client interroge le premier serveur qui lui-même interroge le deuxième serveur).



Image 4- Schéma d'architecture 3 tiers

- Au delà de 3 acteurs, on parle **d'architecture à n tiers**.

Dans le cadre de ce travail, nous nous intéresserons à une architecture client-serveur 2 tiers. Elle présente quelques avantages et inconvénients

Avantages et inconvénients de l'architecture client-serveur 2 tiers

Les avantages sont :

- Toutes les données sont centralisées sur un seul serveur, on a donc « un contrôle de sécurité simplifié ».
- Les technologies supportant l'architecture client/serveur sont plus matures que les autres (et plus anciennes).
- L'administration se porte au niveau serveur. Toute la complexité/puissance peut être déportée sur le(s) serveur(s), les utilisateurs utilisant simplement un client léger.
- Les serveurs étant centralisés, cette architecture est particulièrement adaptée et véloce pour retrouver et comparer de vastes quantités d'information (moteur de recherche sur le web).

Les inconvénients sont :

- Si trop de client veulent communiquer sur le serveur en même temps, ce dernier risque de ne pas supporter la charge.
- Si le serveur n'est plus disponible, plus aucun des clients ne fonctionne.

- Les coûts de mise en place et de maintenance sont élevés.
- En aucun cas les clients ne peuvent communiquer entre eux, entraînant une asymétrie de l'information au profit des serveurs.

C'est quoi les sockets ?

Les sockets servent à communiquer entre deux hôtes Client - Serveur à l'aide d'une adresse IP et d'un port ; ces sockets permettront de gérer des flux entrants et sortants afin d'assurer une communication entre les deux (le client et le serveur), soit de manière fiable à l'aide du protocole TCP/IP, soit non fiable mais plus rapide avec le protocole UDP.

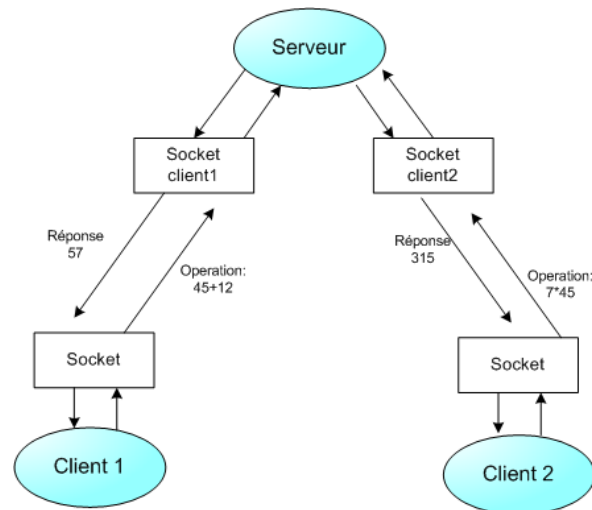


Figure 1- Communication client-serveur à l'aide de sockets

Solution proposée

Structure client-serveur

Nous souhaitons créer notre mode multi-joueur basé sur l'architecture **client-serveur 2 tiers**. En effet, le 2 tiers, bien que limité sera suffisant pour cette première version de notre mise à jour. Afin d'implémenter cette architecture, nous créons un nouveau package nommé **network**, contenant toutes les classes y relatives. Notre solution peut se décrire par le diagramme suivant :

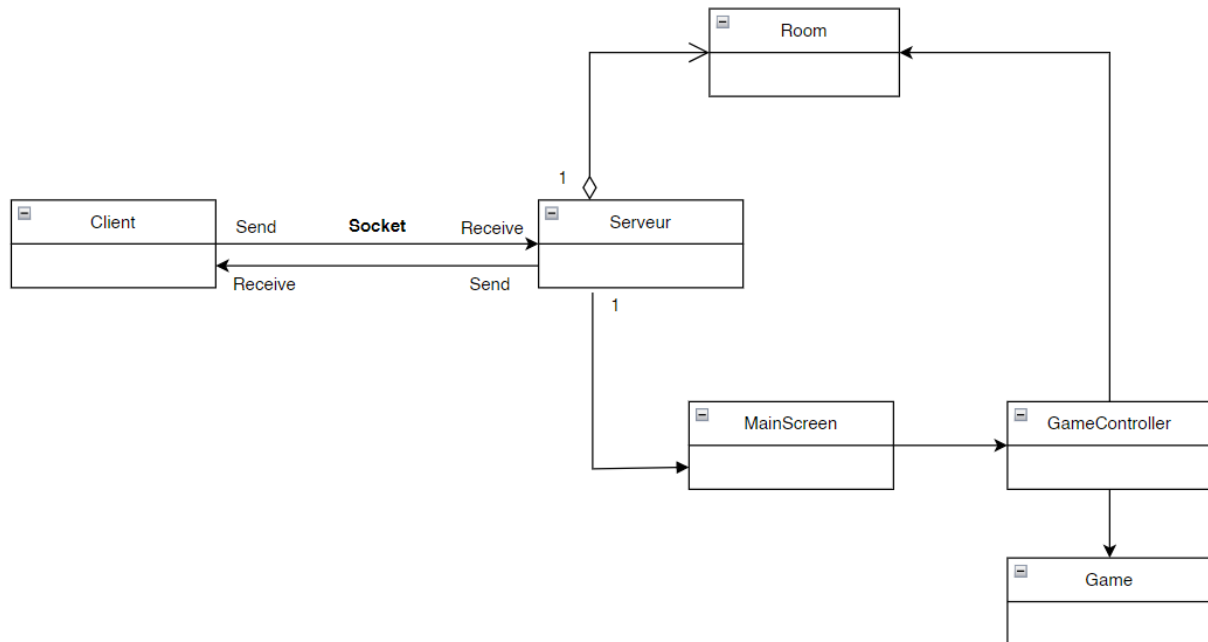


Figure 2- Diagramme de classes de l'architecture client-serveur

Ce diagramme met en exergue les différentes classes suivantes :

Classe	Fonction	Détails supplémentaires
Client	C'est la classe responsable de la connexion au serveur. Elle collecte les informations saisies par le client et les transmet au serveur à travers les sockets	
Serveur	Elle expose la connexion clients à une adresse IP et un port spécifique. A chaque connexion d'un nouveau client, elle le redirige vers un nouveau Thread de la classe MainScreen afin d'afficher à l'utilisateur le menu principal	Le serveur est multithreading, donc il gère la connexion de plusieurs clients au même moment. Ses attributs sont : <ul style="list-style-type: none"> - rooms : La liste des salons de jeu créés dans le serveur - nbRooms : Le nombre de salons de jeu créés dans le serveur
Room	Cette classe contient toutes les informations relatives à une partie de jeu	Ses attributs sont : <ul style="list-style-type: none"> - number : le numéro du salon de jeu - board : la table de jeu de la partie

		<ul style="list-style-type: none"> - turn : une variable indiquant le joueur à qui c'est le tour de jouer - nbPlayers : le nombre de joueurs dans la partie - isGameFinished : un booléen indiquant la fin de la partie - winner : une variable indiquant le joueur vainqueur de la partie
MainScreen	Cette classe est créée par le serveur. Elle affiche le menu principal de l'application et donne la possibilité au joueur de créer une nouvelle partie ou d'en rejoindre une déjà existante. Par la suite elle appelle la classe GameController qui se charge de gérer une partie.	Si le joueur décide de créer une nouvelle partie, elle crée une nouvelle classe room et l'ajoute dans la variable rooms du serveur. Si le joueur choisit de rejoindre une partie existante, cette classe fait la recherche dans la liste des rooms du salon à rejoindre
GameController	C'est la classe chargée de réguler une partie. Elle prend en paramètre dans son constructeur la classe room correspondant à la partie. Elle vérifie ensuite si la partie a suffisamment de joueurs pour commencer. Si c'est le cas elle démarre pour chaque joueur un thread Game	
Game	C'est le thread qui régit un tour de jeu de l'utilisateur. Il récupère l'entrée de l'utilisateur, effectue le mouvement de la pièce et se met en pause jusqu'à ce que le tour de l'adversaire se termine	

Tableau 1- Fonctionnalités des classes de la structure client-serveur

Autres ajustements

En plus de définir la structure client-serveur de l'application, nous avons dû procéder à des modifications supplémentaires. En effet, la classe HumanPlayer, chargée de définir le mouvement d'une pièce en fonction de l'input de l'utilisateur a présenté de fortes limitations quant à la structure client-serveur. Elle ne prend en compte que les inputs utilisateur provenant du serveur et non du client à travers le socket. Dès lors, deux alternatives s'offrent à nous : modifier la classe pour l'adapter ou recourir à l'héritage. Etant donné que nous sommes dans une optique d'extension sans modification, la seconde option semble la plus adéquate. C'est ainsi que nous avons créé la classe

HumanPlayerMulti, qui hérite de la classe **HumanPlayer** mais possède comme attribut supplémentaire l'attribut **socket** de type **Socket**, permettant la récupération de l'input du client pour le traiter. Le principe **Open-Close** reste respecté.

Par ailleurs, une fois la partie terminée, il est nécessaire d'afficher le vainqueur et clôturer la salle. Toutes fois, la question se pose de savoir quelle est la classe qui va se charger de détecter la fin de la partie une fois que la pièce Roi d'un des joueurs est bouffée. La meilleure solution que nous avons trouvée à ce problème est l'utilisation d'un **aspect**. Ainsi nous avons créé un aspect nommé **GameMultiControl**, qui détecte la fin de la partie.

Démonstration

Pour exécuter l'application, on commence par :

- Le lancement de la classe **Serveur**
- Le lancement de multiples instances de la classe **Client** en fonction du nombre d'utilisateurs à simuler

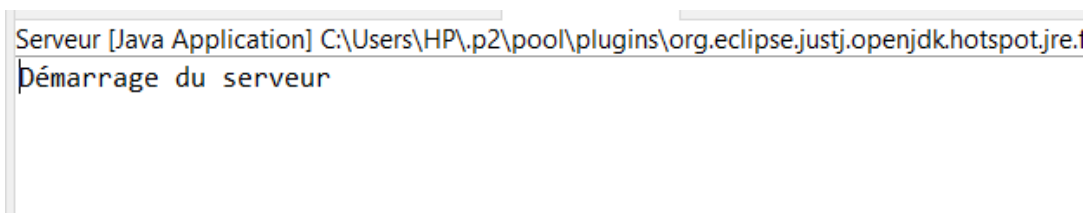


Image 5- Lancement du serveur

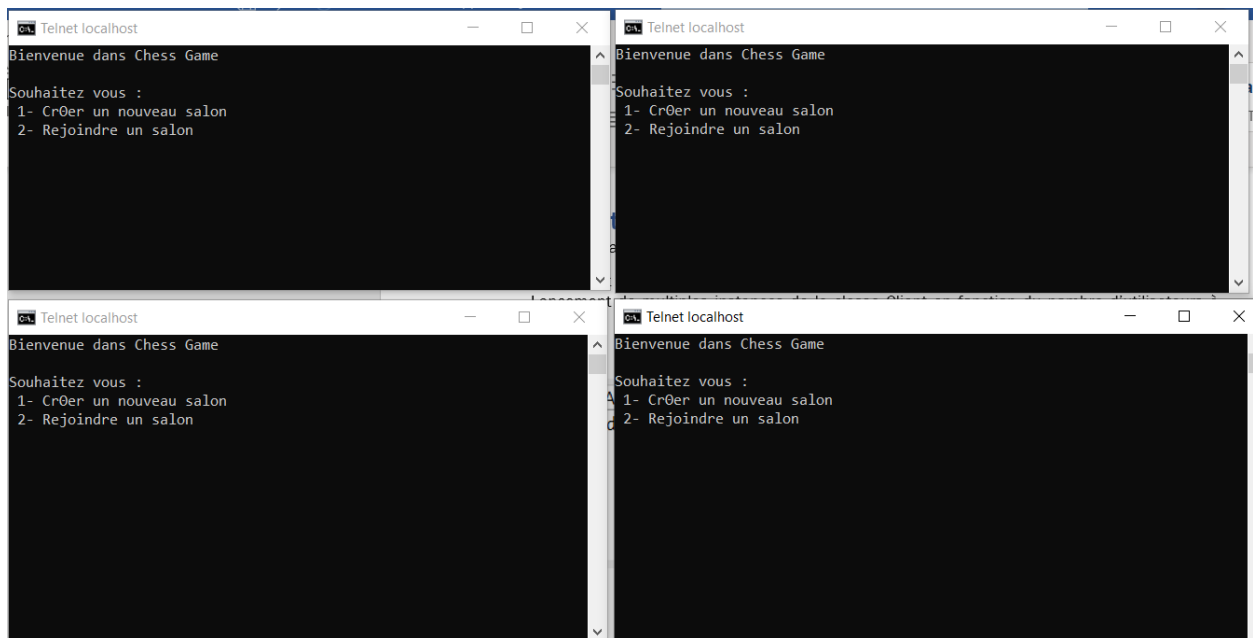


Image 6- Lancement de 4 instances de clients

Un utilisateur peut dès lors créer un salon et un autre peut le rejoindre.

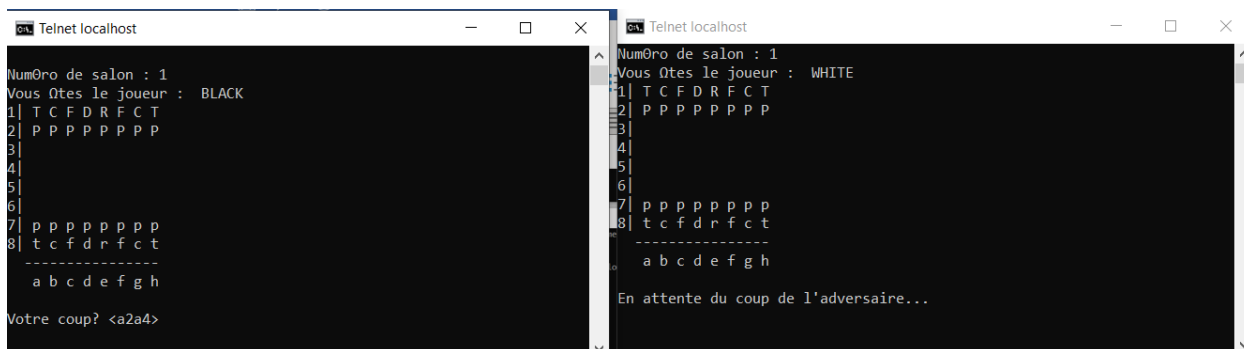


Image 7- Deux joueurs connectés à la même partie

Il est même possible que 2 ou plusieurs parties se déroulent simultanément

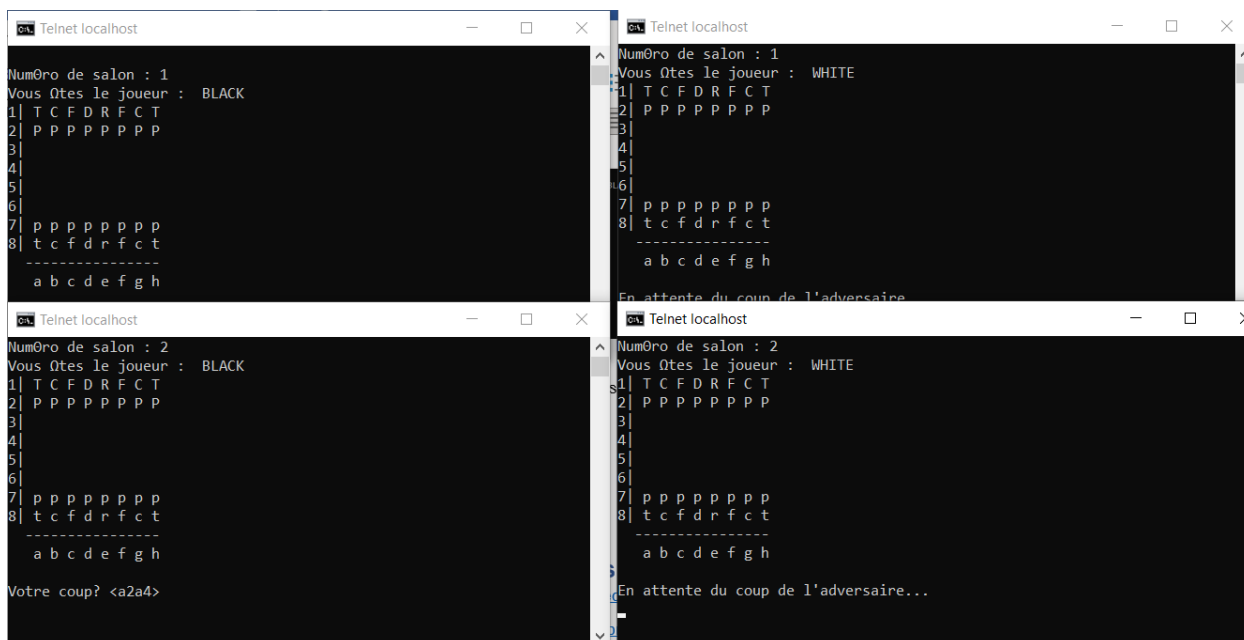


Image 8- Deux parties se déroulant simultanément

Les clients jouent chacun leur tour. Lorsqu'un joueur joue, l'autre est mis en attente jusqu'à la fin de son tour et vice versa. Une fois que la pièce Roi d'un des joueurs est bouffée, la partie s'achève et affiche le vainqueur.

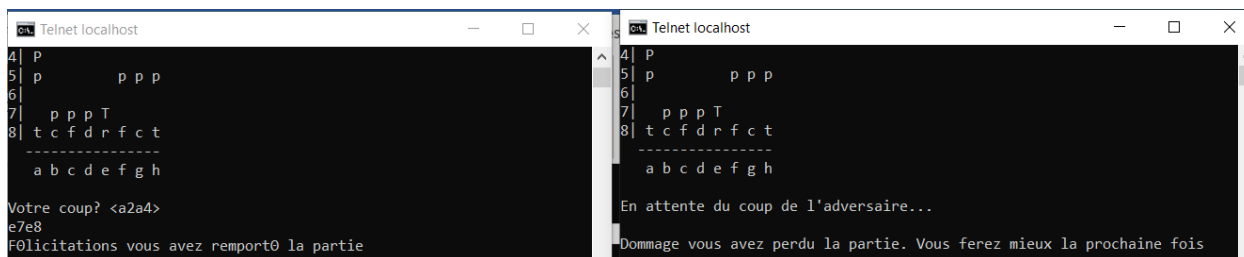


Image 9- La partie est terminée. Le jeu indique le vainqueur et le perdant

Conclusion

En définitive, il était question pour nous tout au long de ce travail, d'ajouter une fonctionnalité basée sur l'architecture client-serveur à une application, tout en veillant à ne pas modifier le code déjà existant. Nous avons ainsi pu mieux comprendre l'architecture client-serveur, son principe de fonctionnement, puis nous avons proposé un modèle basé sur cette architecture nous permettant de développer notre fonctionnalité. Finalement le mode de jeu multi-joueur a bien été implémenté, et nous n'avons pas eu à modifier le code déjà existant. Il faut reconnaître qu'étendre sans modifier fut un véritable défi, mais l'utilisation de l'héritage et de la programmation orientée aspects ont été des solutions adéquates pour réussir cette tâche. En termes de difficultés, la compréhension de la programmation multithreading en fut une aussi ; elle a une logique différente de la programmation non concurrente et il faut pouvoir s'adapter. Néanmoins, le multithreading est efficace et donne accès à de multiples possibilités. Notre application peut être davantage améliorée en incluant par exemple une interface graphique tout en suivant l'architecture MVC (Modèle Vue Controller).

Références

<https://fr.wikipedia.org/wiki/Client-serveur>

<https://www.geonov.fr/architecture-client-serveur/>

http://projet.eu.org/pedago/sin/ISN/8-client_serveur.pdf

<http://sdz.tdct.org/sdz/introduction-aux-sockets-1.html>