Projet Client-Serveur PC2R

-BOGGLE-

LACHACHI Mohammed CharafEddine
YAGCI Kaan

Etudiants M1-STL
UPMC - Mai 2018

Contents

I. Introduction

Dans le cadre de l'unité d'enseignement *PC2R*, nous a été demandé de réaliser une application client-serveur du jeu de lettre *BOGGLE*.

L'objectif du projet est d'implémenter le client et le serveur dans deux langages différents, en se basant sur des sockets et un protocole de message donné au préalable.

II. Serveur

1. Réalisation du serveur

Le serveur a été réalisé en *OCaml* (version 4.02.1 installée sur les machine de la *PPTI*). Il utilise les modules *Thread*, *Unix* et *Str* (pour les regex). Les extensions réalisées coté serveur sont la Messagerie instantanée (public et privé), la vérification immédiate des réponses (utilisée par défaut) et la journalisation des scores. L'intégralité du code du serveur est sur le fichier « server.ml ».

2. Traitement des commandes

Lors de la réception d'une commande, cette dernière est transformée en une couple grâce à la méthode « *received_request* » qui effectue un split des requêtes reçues et retourne un couple composé d'un numéro et une liste d'arguments, exemple pour la requête (*CONNEXION/USER*), le couple (1, [USER]) est retourné.

3. Connexion

La phase de connexion commence à la réception de la commande « *CONNEXION* », à ce moment un thread est créé afin d'invoquer les deux méthodes suivantes :

- Traitement connecte : permettant de diffuser le message « *CONNECTE/USER* » aux autres joueurs déjà connectés, d'envoyer le message « *BIENVENUE/USER* » au joueur venant de se connecter et aussi de lancer la partie en invoquant la méthode (début_session) si le joueur est le premier à se connecter dans la partie sinon en envoyant la commande « *TOUR/tirage* ».
- Boucle joueur : la méthode contenant la boucle infinie nécessaire pour l'attente de réception et redirection des commandes venant du client vers les méthodes de traitements adéquates.

4. Structures de données

Pour l'implémentation nous avons choisi de représenter le joueur par un type « *joueur* » contenant les différentes informations relatives au joueur. A chaque connexion d'un nouveau joueur la structure ci-dessous est initialisée.

```
type joueur = {
  nom: string;
  id : int;
  socket: Unix.file_descr;
  mutable score: int;
  inchan: in_channel;
  outchan: out_channel;
  mutable playing :bool;
  mutable  reponses : string list
}
```

Les joueurs crées sont stockés dans la liste « *joueur_liste* », ce qui nous a permis d'utiliser les nombreuses fonctionnalités du module *List*.

Les cellules de la grille du jeu sont eux aussi représentées par un type « *cell* » contenant l'identifiant de la cellule et ses cellules voisines selon les huit principaux points cardinaux. Au moment du tirage la matrice « *tirage_matrix* » de dimensions 4x4 est initialisée avec 16 cell selon le tirage généré aléatoirement par la méthode (des).

```
type cell = {
  id : string;
  n : string;
  e : string;
  s : string;
  o : string;
  ne : string;
  se : string;
  so : string;
  nou : string
}
```

Enfin lors du lancement du serveur une liste « *dictionnaire* » est initialisée à partir du fichier (dictionnaire.dat) représentant le dictionnaire utilisé pour la résolution des mots proposés par les joueurs. Le choix d'utilisation d'une liste chargée une fois au lancement du serveur est pour éviter les entrées/sorties disque à chaque vérification de mot.

5. Extensions réalisées

- CHAT

  Afin de permettre la communication entre les joueurs connectés, deux commandes ont été utilisées selon le protocole donné. La commande « *ENVOI/message* » permettant la diffusion du message à l'ensemble des joueurs connectés, et la commande « *PENVOI/user/message* » pour envoyer un message privé à l'utilisateur (user), l'implémentation à nécessité deux méthodes (traitement_message et traitement_messagei).

- VERIFICATION IMMEDIATE

  Pour ne pas alourdir l'implémentation, cette méthode de vérification a été utilisée par défaut., à l'envoi de la commande « *TROUVE/mot/trajectoire* » le joueur reçoit la validation ou non du mot, cette dernière consiste en quatre étapes de vérifications

  - Vérifier si le mot a déjà été proposé ou non par un autre joueur (méthode *find_in_enchere*).
  - Vérifier la disponibilité ou non du mot dans le dictionnaire (méthode *find_diction*).
  - Vérifier la trajectoire du mot, si toutes les cases utilisées sont voisines (méthode *verify_trajectoire*).
  - Vérifier si le mot proposé correspond bien à la trajectoire reçu (méthode *find_word_of_trajectoir*).

- JOURNALISATION

  Afin de publier régulièrement les score des parties sur une page, au début nous avons envisagé de créer une *Restful Api* en PHP qui sera interrogée par le serveur OCAML via les requêtes http usuelles. Malheureusement nous n'avons pas réussi à avoir les droits d'installer les outils nécessaires dans les salles de la PPTI. Pour remédier à ce problème nous avons opté pour la solution suivante.

  Ajouter dans le serveur un fichier « *journal.json* » qui est alimenté par les score des joueurs à chaque tour, ajouté à ce fichier une page web « journal.html » écrite en HTML/JS qui permet d'analyser le fichier « *journal.json* » via des requêtes *AJAX* et afficher le résultat .

  Pour plus d'information sur l'utilisation cf. section Utilisation.

- CLIENT AUTONOME

  Pour ce qui est d'un client tricheur, nous avons introduit dans l'application deux mécanisme de triche le premier permet de proposer des mots corrects pour la grille générée et cela en utilisant le bouton « Propositions » de l'interface graphique, et le second est un client à part entière qui est automatique et permet de jouer le rôle d'un joueur usuel mais en proposant toujours les meilleures solutions.

6. Concurrence

Lors de l'implémentation du serveur nous avons choisi d'affecter un thread à chaque connexion d'un nouveau client, ce qui est susceptible de créer des accès concurrents aux variables (toplevel) comme « *joueur_liste, tirage, phase …* », pour cela chacune de ces variables possède deux verrous « *Mutex.lock, Mutex.unlock* » utilisés pour assurer des accès sûrs et efficaces.

7. Remarques

Au cours de l'implémentation du serveur, nous avons effectué les tests via « *telnet* » ce qui ne posait aucun problème, mais avec le client java si le processus est arrêté (kill) sans envoyé un message de déconnexion au serveur via l'interface graphique (LogOut ou en fermant la fenêtre) le serveur s'arrête. Aussi la gestion des pseudos similaires des joueurs n'a pas été gérée, donc le serveur accepte deus joueurs avec le même pseudo ce qui risque de créer des conflits.

Par contrainte de temps, nous n'avons pas pu corriger ces erreurs minimes

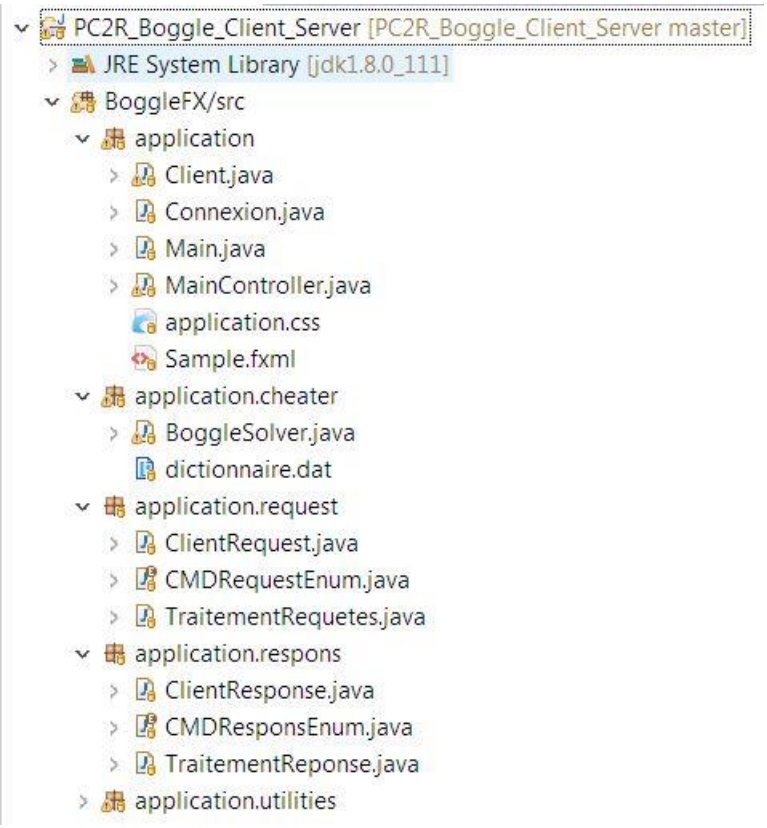
III. Clients

Pour ce qui est des clients, nous avons choisi d'implémenter deux clients, un client sous forme d'application desktop et un autre sous forme d'application web, tous cela par défis pour comprendre le comportement du serveur avec des applications basées sur des architectures différentes.

1. BoggleFX

La première application est codée en *Java* en se basant sur la librairie *JavaFX* pour l'interface graphique.

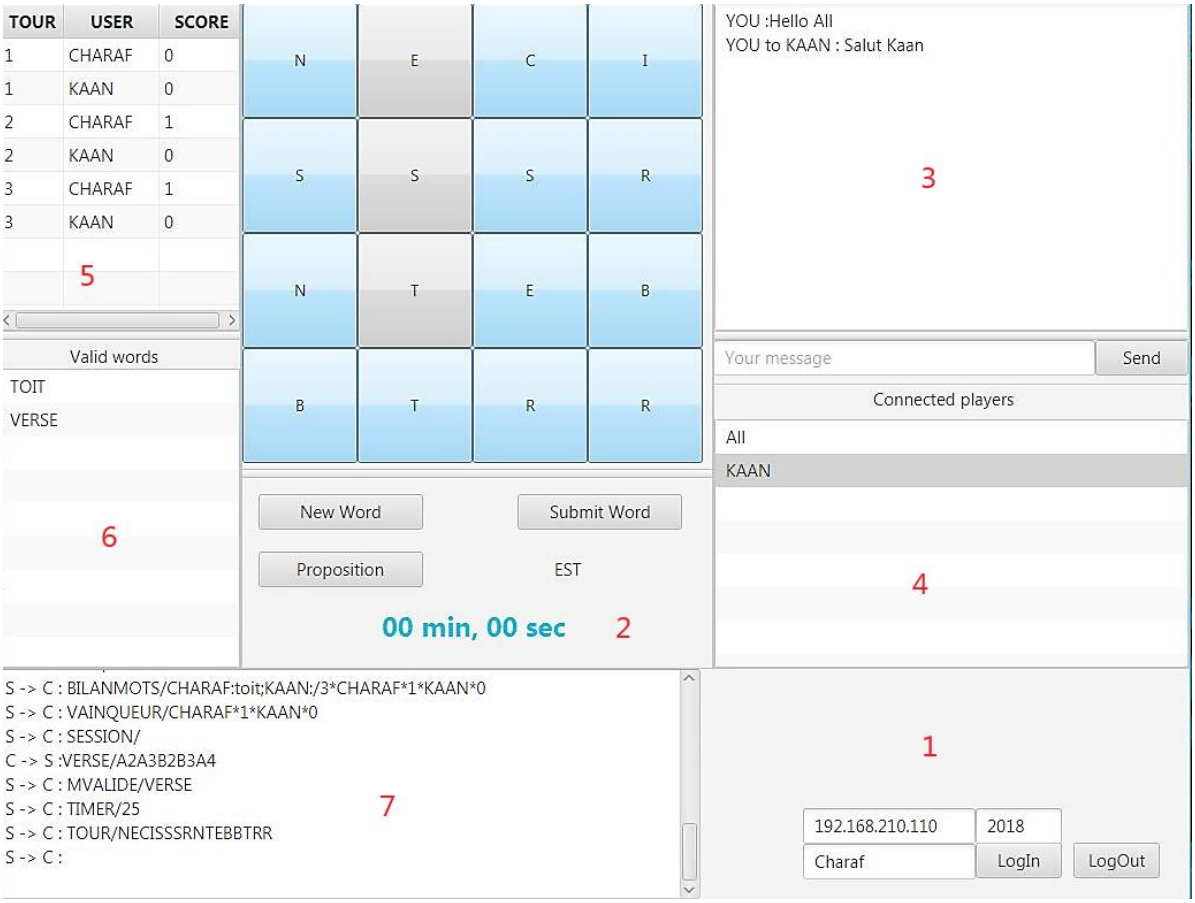
a) Classes principales



- Client : contient l'ensemble des informations concernant le joueur similaire au type joueur utilisé dans le serveur, chaque instance du client est exécutée dans un thread.
- Connexion : permettant de gérer la connexion au serveur, utilisée dans la classe Client.
- Main : est la classe propre à JavaFx pour le lancement de la scène de l'interface et le contrôleur
- MainController : la classe principale de gestion de l'interface, contient toutes les méthodes invoquées par les contrôles de l'interface
- BoggleSolver : cette classe est utiliser pour trouver toutes les solutions possibles pour un tirage de 16 lettres donné. Sur l'interface le bouton « proposition » permet d'afficher des solutions grâce à cette classe.

- TraitementRequetes : toutes les requêtes envoyées du client vers le serveur sont traitées dans cette classe.
- TraitementReponse : lancée dans un thread, contient une boucle infinie pour l'attente des requêtes venant du serveur, à la réception d'une requêtes cette classe invoque MainController pour l'afficher sur l'interface.

b) Présentation de l'interface (manuel)



- **<u>ZONE 1</u>** : lors de l'ouverture de l'application, la première chose à faire est d'introduire l'adresse IP du serveur et le port d'écoute, ces derniers par défaut sont à 127.0.0.1 :2018, et aussi le pseudo du joueur. Une fois le bouton « *LogIn* » appuyé l'application se connecte au serveur. Pour se déconnecter le bouton « *LogOut* » permet d'envoyer le message « *SORT/user* ».
- **<u>ZONE 2</u>** : une fois le joueur connecté, le serveur envoie les 16 lettres du tirage qui sont affichées dans la grille. Le joueur peut choisir l'ensemble des lettres du

mot souhaité en sélectionnant chaque lettre, le bouton « *Submit word* » permet d'envoyer la sélection au serveur pour vérification. Le joueur peut aussi décider un nouveau mot pendant la réflexion en cliquant sur le bouton « New Word » cela réinitialise les lettres sélectionnées.

Une possibilité de triche a été introduite dans l'application, en cliquant sur le bouton « Proposition », le client reçoit des propositions de mots correcte selon le tirage et cela en se basant sur un dictionnaire et un algorithme de résolution de la grille.

Cette zone contient aussi un « *Timer* » initialisé selon la durée d'un tour paramétrée dans le serveur. Vu que le protocole ne spécifié aucune commande pour le timer nous avons ajouté la commande « TIMER/durée ».

- **ZONE 3** : Les joueurs connectés peuvent communiquer entre eux via le chat en mode privé ou public, Une liste dans la « *zone 4* » permet de sélectionner « All » pour un message public ou le pseudo d'un joueur pour un message privé. Par défaut les messages sont en mode public.

- **ZONE 4** : Contient la liste des joueurs connectés pendant la session, et sert aussi à la sélection d'un joueur ou All pour l'envoie des messages de chat. Pour connaitre les joueurs déjà connectés nous avons ajouté au protocole la commande « *CONNEDTEDPLAYERS/user1/userN* » qui est envoie au nouveau joueur juste après la connexion.

- **ZONE 5** : A la fin de chaque le score envoyé par le serveur est affiché dans cette liste.

- **ZONE 6** : Après la vérification du mot proposé par le joueur, si le mot est validé par le serveur, ce dernier est ajouté à la liste « *Valid word*s »

- **ZONE 7** : Cette zone à ajouté spécialement pour le *debug* afin de voir et comprendre les messages échangés entre le client et le serveur et vice versa.

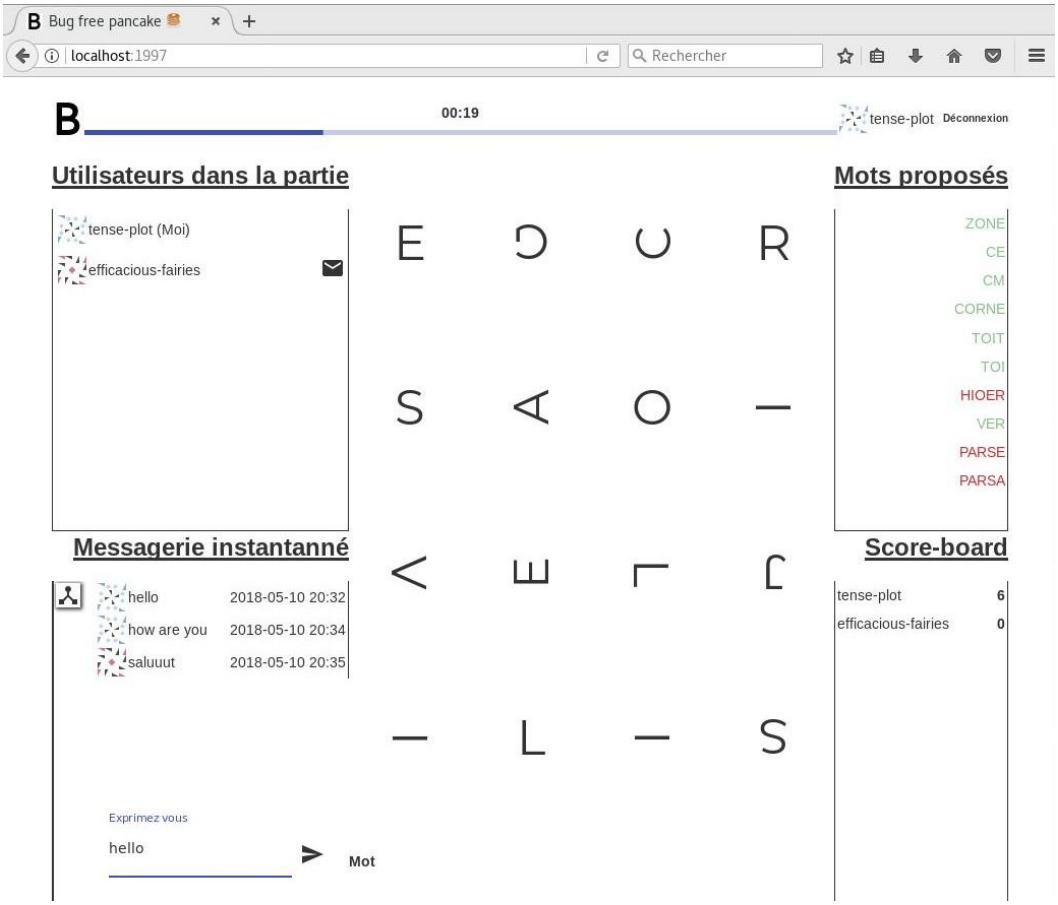
2. Client Web

La réalisation du client web comprenait plus de difficulté en comparaison au client réalisé en Java. Afin d'utiliser des sockets comme base de communication entre le serveur et le client, notre architecture a été divisé en trois couches.

- Interface graphique (UI) : est la partie qui permet à l'utilisateur d'interagir avec le serveur pour jouer au jeu, cette interface a été écrite en *Angular 6*.
- Intergiciel. (Middleware) : écrit en Typescript permettant à l'interface graphique de communiquer avec le serveur via des sockets TCP,
- Server Ocaml : permettant de gérer les différentes requêtes envoyées par les clients.

Dans le but de facilité le déploiement du client, nous avons opté pour l'utilisation de l'outil « ***Docker »*** qui permet de déployer le tous en quelques ligne de commandes (consultable sur le répertoire git du projet).

Pour une meilleure ergonomie et une facilité de familiarisation avec l'utilisation de l'interface nous avons conservé une même présentation pour les deux clients.



IV. Utilisation

Pour l'expérimentation et l'utilisation des différent module de notre projet, vous avez accès à l'ensemble des projets via les répertoires git suivants.

- https://github.com/CharafLachachi/PC2R_Boggle_Client_Server : ce lien permet d'accéder au server (dossier server), client Java (dossier BoggleFx), client tricheur Java (dossier ClientCheater) et le client web (dossier ClientWeb) . Dans chacun des projets vous trouverez un readme vous guidant pour l'utilisation.
- https://github.com/Misteryagci/bug-free-pancake : ce lien est dans le cas ou vous voulez accéder seulement au serveur et le client web.

Pour ce qui de la journalisation des scores sur une page web, vous trouverez sur le dossier (Server) une page « journal.html » à lancer sur un navigateur autre que « Google Chrome » qui interdit l'utilisation d'un fichier local qui n'est pas dans un server web.

V. Conclusion

Avant d'être un devoir à faire la réalisation de ce projet était un défis, nous avons choisi d'implémenter le serveur avec un langage et un paradigme de programmation jamais utilisés auparavant, nous n'avons pas regretté ce choix, OCAML nous a facilités de multiples taches avec les différentes libraires qu'il offre, pour ce qui est des deux clients, c'était des langages auxquels nous sommes plutôt familier, mais n'empêche, nous avons pu pratiquer la programmation concurrente et répartie, chose que nous faisions pas souvent.