Rapport

Boogle Master Project

MASTER STL – PC2R

2017-2018

Par :

William Sergeant

Vincent Havinh

Index

Client

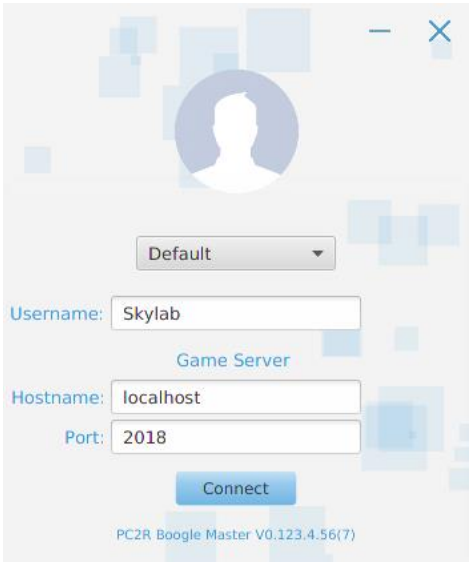
Interface

L'interface est réalisée en **Java**. Plus précisément on utilise la librairie **JavaFX** ainsi que des ressources externes (images, sons, layouts) pour les vues avec **FXML** permettant l'implémentation de scènes et de feuilles **CSS** afin de définir des styles pour les groupes d'éléments.
Java et **JavaFX** furent choisis car nous disposons d'un socle de connaissances quant à leur utilisation ce qui nous a permis d'être plus efficaces et innovants lors du développement du client. L'introduction de l'utilisation de documents **FXML** et **CSS** en adéquation avec le code fonctionnel est cependant une nouveauté pour nous. Cela nous a permis de mieux utiliser les composants graphiques de **JavaFX** ainsi que de faciliter les interactions entres les vues et les contrôleurs tout en élargissant notre cercle de compétences. Nous utilisons donc un framework simili MVC, le Modèle et la Vue étant sensiblement entrelacés.

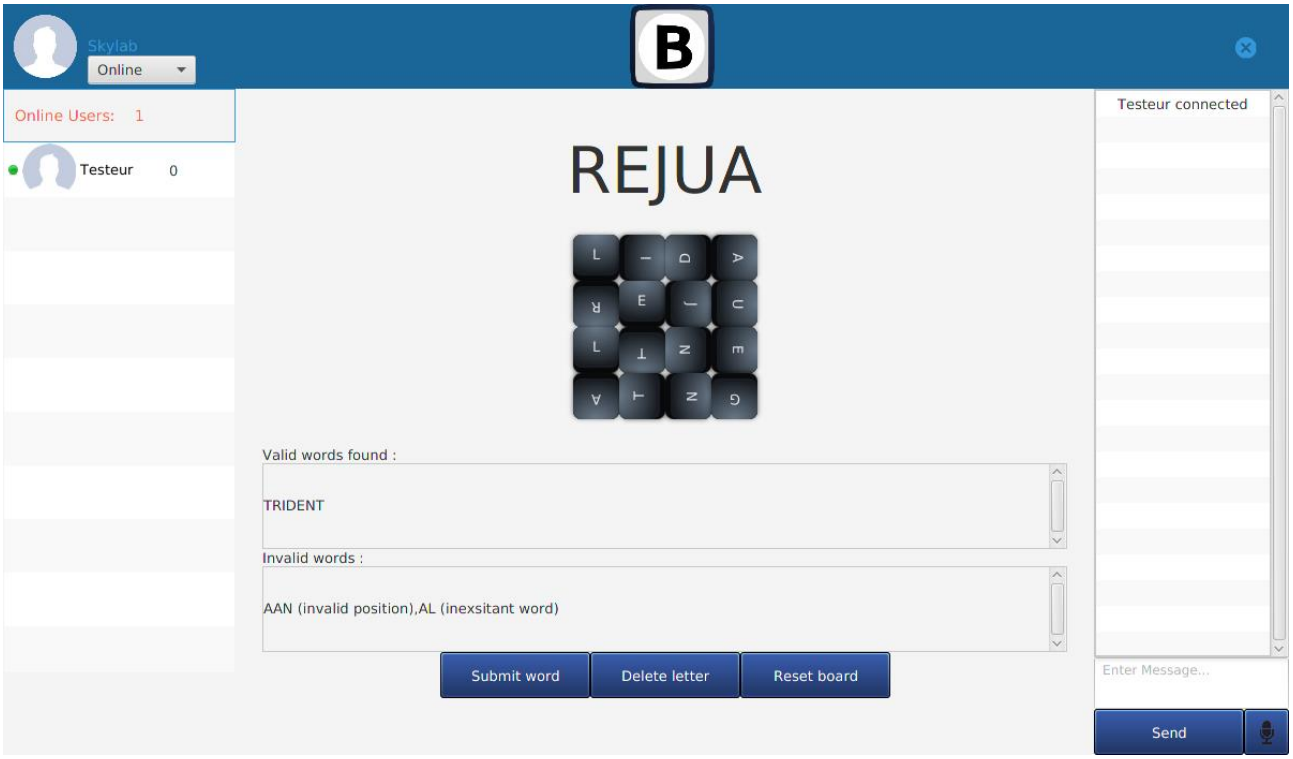
Vues

Deux vues principales ont été créées lors de la réalisation de ce projet :

- Une vue pour l'écran d'identification des joueurs, permettant de renseigner l'adresse et le port de connexion au serveur de jeu, ainsi que le pseudonyme du client lors de sa session de jeu.

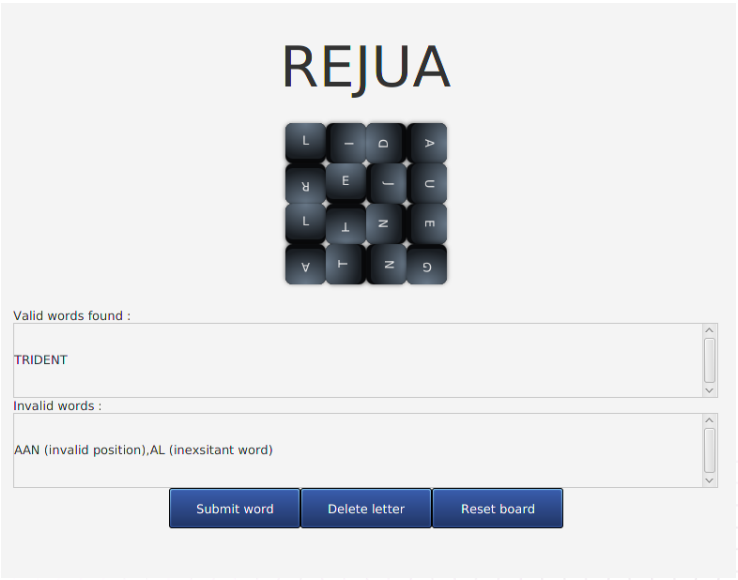


- Une vue contenant la liste des joueurs présent ainsi que leurs scores, un panneau latéral comprenant un espace de chat entre les joueurs et enfin une partie centrale représentant le plateau de jeu.

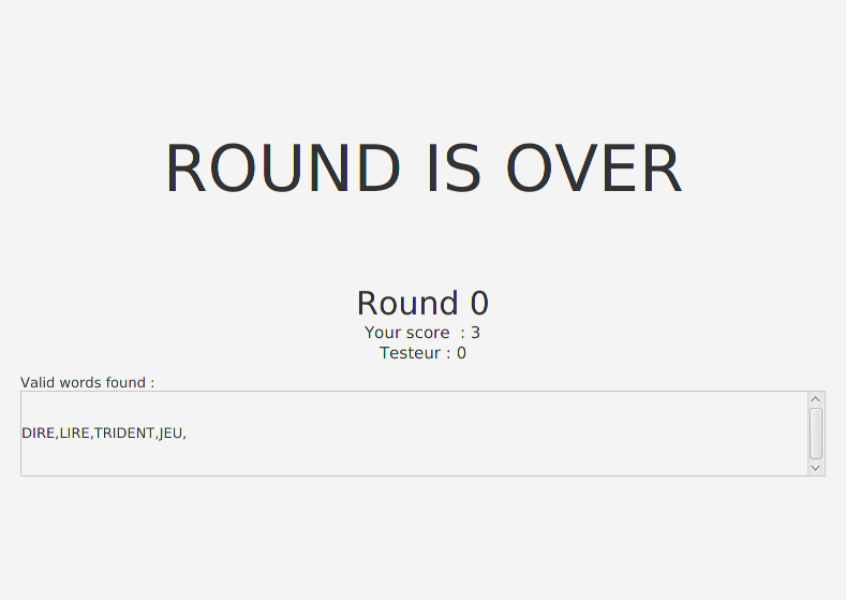


Cette partie centrale peut elle-même afficher plusieurs vues :

- Le plateau de jeu avec la matrice de dés, la liste des mots validés et invalidés ainsi que le mot en cours de confection par le joueur. De plus, l'utilisateur dispose de trois boutons afin de pouvoir effacer la dernière lettre sélectionnée, de réinitialiser le mot complet et enfin de soumettre sa proposition au serveur.

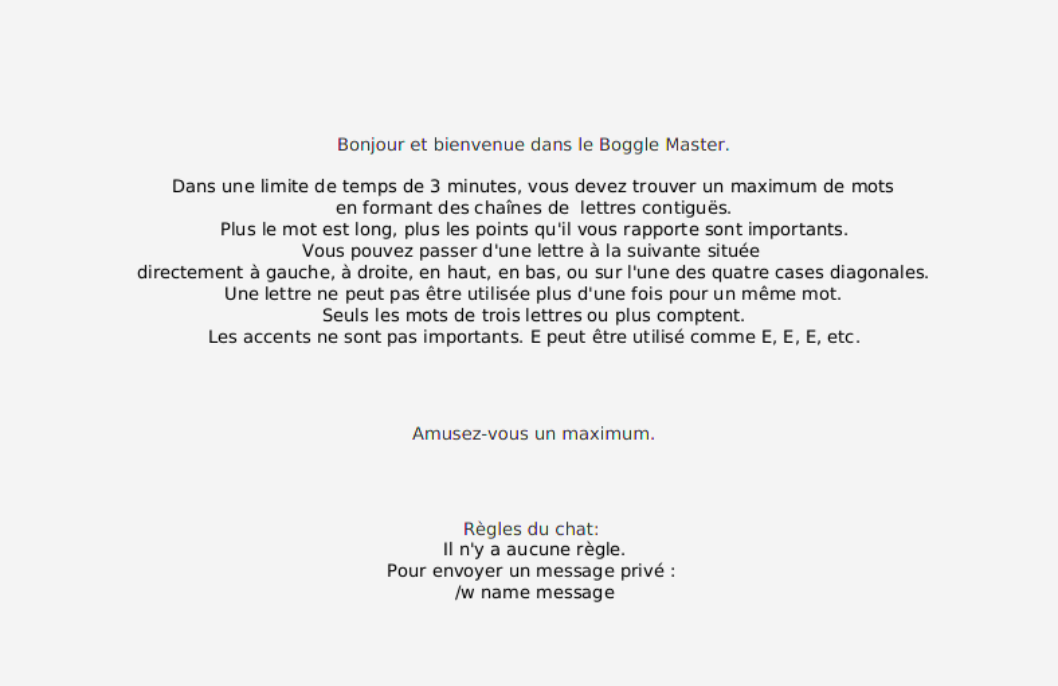


- Le panneau de résultat du tour écoulé avec le nombre de points accumulés par chaque joueur ainsi que la liste des mots validés par le serveur pour chaque client.



- La vue de fin de session avec les résultats finaux pour chaque joueur et le titre de vainqueur.

- La vue de début de session, dans l'attente de commencement du premier tour.



Note : il a été décidé de ne pas implémenter d'horloge ni de chronomètre pour le tour courant afin de laisser les joueurs libres d'esprit et d'épargner le stress de voir les secondes diminuer.

Liaisons

Les interactions entre l'interface et son contrôleur se font majoritairement par le biais de boutons, lesquels se sont vu attribuer une fonction spécifique lors de leur appel.
En outre au lieu de devoir créer, instancier, personnaliser le bouton et par la suite lui affecter un écouteur d'événement (event listener) définissant le comportement à avoir (en cas de clique du bouton par exemple) celui-ci est simplement défini par une ligne dans le fichier FXML correspondant :

<Button fx:id="**buttonSendWord**" onAction="#**sendWordAction**" text="Submit word" id="**rich-blue**"/>

Le champ ***fx:id*** est l'identité du button dans le fichier FXML, celui-ci peut être utilisé par le contrôleur java associé. L'instruction dans le code du contrôleur **@FXML Button buttonSendWord** permet de récupérer l'instance de ce bouton (après le chargement du fichier FXML, moment où les instanciations des composants sont effectuées). Ainsi celui-ci sera directement utilisable et référera au même composant dans l'interface et dans le code.

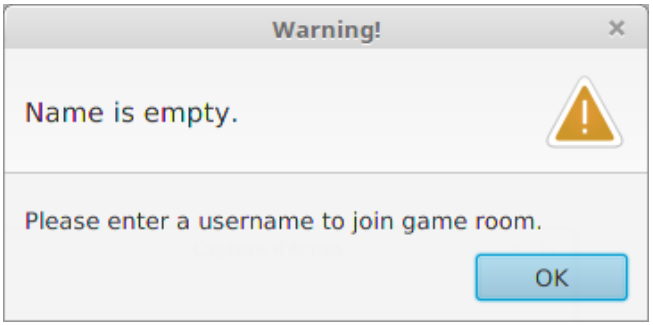
L'autre champ ***onAction*** défini la fonction associée en cas d'activation du contrôleur. Ici, lors du clique sur le bouton, la fonction **sendWordAction** sera exécutée. Ces fonctions se comportant tels les event Listeners et Handlers, il est possible de passer un argument de type **Event**. Autrement il est possible de paramétrer un champ user-data dans le composant FXML et de récupérer cette valeur lors de l'appel. Par exemple, lors de l'appel de la fonction **addLetter** (appelée lors du clique sur une lettre de la matrice de jeu) le champ user-data contient la position du bouton (par exemple **user-data= « A1 »**). En récupérant également le champ de texte du bouton (**« L »**) on ajoute au mot en cours **la lettre L et la trajectoire A1**.

Enfin le champ ***id*** est une référence interne au composant et servira notamment à définir les références aux fichiers **CSS** afin d'appliquer les styles appropriés aux composants de manière automatique.

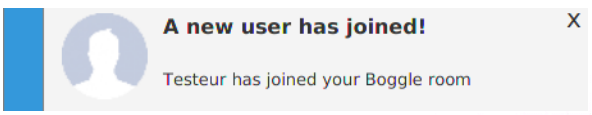
Fonctionnalités

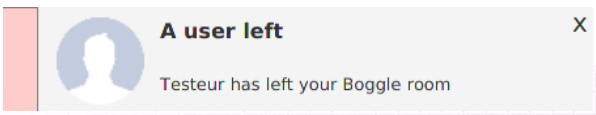
Afin de fournir un confort d'utilisation pour le joueur, divers améliorations ont été mises en place :

- La plupart des exceptions sont gérées avec l'appel d'une fenêtre modale explicitant les raisons de l'erreur au joueur : erreur de connexion au serveur, connexion perdue ou autres actions de jeu prohibée.



- Une notification visuelle et sonore en bas à droite de l'écran afin de prévenir de la connexion d'un nouveau joueur ou du départ de celui-ci. A noter qu'il est possible d'ajouter une notification pour d'autres motifs tels que le début d'une session / tour de jeu.

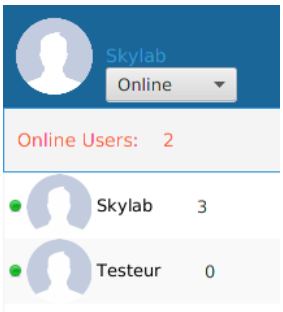




D'autres améliorations visuelles sont présentes mais non fonctionnelles, l'implémentation n'ayant pas pu être portée à son terme par manque de temps de développement :

- L'affichage d'une bulle de couleur verte / orange / rouge à côté du pseudonyme de chaque joueur définissant son statut : En Ligne / Occupé / Absent, respectivement.

- L'utilisation d'images de profil personnalisées pour chaque joueur, lui permettant une identité visuelle unique dans la liste des joueurs et le chat.



Contrôleurs

Il existe un contrôleur associé à chaque vue du programme. Ainsi la vue **LoginView**.fxml met en place les composants, **LoginStyle**.css permet d'appliquer les styles et **LoginController**.java gère les interactions entre le code de traitement des données et l'interface. De même pour **ChatView**.fxml, **ChatStyle**.css et **ChatController**.java.

Le **ChatController** contient les fonctions de gestion de l'affichage et des informations du jeu. Celui-ci gère les comportements internes tels que la sélection des lettres, l'ajout de message au chat et le basculement entre les écrans de session / plateau de jeu / résultats du tour. De plus celui-ci communique avec le serveur de jeu en envoyant des messages défini selon le protocole.

Chat

Une fenêtre de chat est présente dans un panneau latéral à gauche. Celle-ci utilise une classe tierce (Bubble) qui fournit un affichage plus agréable pour les messages. Les messages publics envoyés par le joueur s'affichent en vert (en violet pour les privés), les messages publics reçus sont en blanc (en rose pour les privés). Enfin, les messages du serveur sont en gris.

L'envoi de messages privés se fait selon le système du « ***whispering*** » : afin de contacter directement un joueur, John, pour lui dire '*bonjour*' on tapera la ligne suivante dans le chat :

```
/w John bonjour
```

Uniquement le destinataire pourra lire le message.
Une implémentation supplémentaire mais non achevée est l'envoi de messages vocaux, ceux-ci seraient enregistrés par le client, convertis en bytecode puis envoyés au serveur sous forme de chaîne. Ils seront recomposés par le client cible et le message vocal sera restitué dans sa forme originale. Ceux-ci sont malheureusement non testables pour le moment.

Threads

Ce contrôleur communique efficacement avec l'interface afin de mettre à jour celle-ci avec les informations correspondantes. Afin de passer la main au thread JavaFX pour changer les données de l'écran on utilisera la méthode **Platform.runlater(()→ {runnable}) ;**
Cela nous permettra d'effectuer des modifications mineures de l'affichage tel que le changement des scores lorsque le thread sera disponible.

Pour des traitements plus complexes tel que l'ajout d'un nouveau message à l'interface, nous utiliserons un système différent : les **Tasks.** Celles-ci implémentent l'interface Worker, héritage de Swing, permettant de faire des calculs plus lourds en parallèle et de prévenir l'interface du changement alors disponible, évitant ainsi le freeze de l'application.

Serveur

Généralités et architecture des fichiers

La partie serveur de ce projet a été implémentée en langage C.
La gestion de la concurrence se fait à l'aide de threads préemptifs, de sémaphores et d'attentes passives sur conditions. Pour ce faire, nous avons utilisé les bibliothèques standards *<pthread.h>* et *<semaphore.h>*.

Le code du serveur est structuré en 4 fichiers de la manière suivante :

- **global.h**
  Contient les déclarations des bibliothèques pour tous les fichiers c du programme, les définitions des différentes structures utilisées, les sémaphores et mutexs, ainsi que les déclarations des variables globales stockant les stockant les données métier du jeu Boggle.

- **server.c**
  Contient toutes les fonctions liées à l'aspect réseau du programme : la création du socket serveur, sa mise en écoute, et la boucle d'acceptation des connexions entrantes. Les bibliothèques *<unistd.h>*, *<sys/types.h>*, *<sys/socket.h>* et *<netinet/in.h>* sont utilisées dans ce but. De plus, ce fichier contient aussi les fonctions d'allocation dynamique et d'initialisation des structures de données et des variables globales. Enfin, il contient la fonction de parsing des arguments de la ligne de commande.

- **thread_client.c**
  Contient la fonction boucle d'écoute d'un thread qui attend et traite les commandes du client à travers le socket (CONNEXION/ SORT/ TROUVE/ ENVOI/ PENVOI).

- **thread_game.c**
  Contient toutes les fonctions en lien avec le déroulement du jeu : le cycle complet d'une session, les vérifications d'un mot proposé par l'utilisateur (vérification sur la trajectoire, le dictionnaire et si le mot est déjà pris). Ce fichier rassemble des fonctions de stockage et de suppression des mots dans les structures de données clients.

Structure du programme et Concurrence

Thread Serveur (méthode main du fichier server.c)

Le processus principal initialise les différentes structures de données et sockets.

Il lance ensuite un thread qui exécute la méthode *game_handler* et qui gérera le déroulement du jeu. (ce thread sera détaillé plus bas).

Puis le thread serveur entre dans la boucle principale d'écoute sur socket.

```
client* clients[MAX_CLIENTS];
sem_t* slots_clients;
```

La première instruction de la boucle est Prendre (sémaphore *slots_clients*).
Ce sémaphore représente le nombre de places disponibles. Il est initialisé à MAX_CLIENTS. Si un slot est disponible, le processus décrémente le sémaphore est attend une connexion sur le socket d'écoute du serveur. Quand un client extérieur se connecte, le processus serveur va parcourir le tableau de client* en cherchant un emplacement libre (*clients[i]→is_co == FALSE*).

```
typedef struct {
    boolean is_co;
    boolean is_ready;
    int sock;
    char* user;
    int score;
    propos* list_prop;
} client;
```

Cette évaluation n'est pas une section critique. En effet, comme nous le verrons plus tard, les threads clients peuvent changer les valeurs de leurs données partagées, mais le booléen *is_co* n'est modifié par eux que lorsqu'ils se déconnectent (il passe alors de TRUE à FALSE). Donc si le thread serveur lit un *is_co == TRUE*, il passera à la case suivante du tableau client*, qu'il y ait modification (due à une préemption) des données de ce client ou non entre temps. De plus, on est sûr qu'au moins une place est disponible dans le tableau grâce au sémaphore *slots_clients* et seulement le thread serveur peut ajouter des client* à ce tableau. Ainsi, il ne peut pas y avoir de corruption due à des préemptions pendant la recherche d'un slot dans le tableau de client*.

Une fois trouvé, le serveur initialise cette place avec les valeurs d'usine d'une structure client, renseigne le numéro du socket de la connexion client, puis crée un thread client qui exécute la méthode *client_handler,* avant de recommencer la boucle d'écoute.

Thread client (méthode client_handler du fichier thread_client.c)

A son lancement, ce thread se met en écoute sur le socket connecté au client, en attente de réception d'une commande.
Dans la description des actions qui va suivre, l'absence de section critique lors de l'envoi de messages aux clients se justifie par le fait que si, entre le moment ou un thread lit qu'un client→is_ready == TRUE et l'envoi du message par le thread à ce client, le client se déconnecte, alors la fonction send() retournera une erreur car le socket sur lequel elle essayera d'envoyer est fermé. Cette erreur est acceptable (perte du message pour le client déconnecté.)

- **CONNEXION/user/**
  le thread verrouille le *mutex game→mutex_clients* pour modifier les champs *clients[slot]→user = user* et *clients[slot]→is_ready = TRUE* de manière sûre.
  Puis il envoie le message **BIENVENUE/** au client et **CONNECTE/** aux autres.

- **SORT/**
  le thread envoie le message **DECONNEXION/** aux autres clients. Il n'y a pas de section critique ici. Puis le thread sort de la boucle d'écoute de commande. A sa sortie, le thread verrouille le *mutex game→mutex_clients* pour pouvoir supprimer (et dés-allouer) tous les mots proposés par le client de sa structure de données, et réinitialiser les booléens *is_co* et *is_ready* à *FALSE*.
  Le thread déverrouille le mutex, et enfin il incrémente le sémaphore *slots_clients* de 1 avant de se terminer

- **TROUVE/mot/trajectoire/**
  le thread verrouille le *mutext game→mutex_clients*, puis appelle la fonction *est_valide()* de *thread_game.c*, qui va vérifier la correspondance trajectoire/mot proposé, puis la trajectoire, le mot dans le dictionnaire et enfin si le mot n'est pas déjà pris. Cette étape consiste à comparer le mot avec les autres mots proposés dans toutes les structures de données des clients connectés. Cette section est critique car si un thread est en train de comparer les mots d'un client connecté, et que se client se déconnecte et dés-alloue ses mots proposés, le thread comparant lira une mémoire non corrompue.
  Après ces vérifications, le thread ajoute le mot (si valide) aux mots proposés de son client. C'est toujours une section critique dans le cas -immédiat activé car si deux threads ajoutent le même mot au même moment, selon les préemptions ils peuvent passer l'étape de comparaison avec l'autre client sans détecter le mot chez l'autre car pas encore ajouté (comportement incertain).
  Enfin, le thread déverrouille le *mutex game→mutex_clients*.

- **ENVOI/message/**
  Le thread envoie le message **RECEPTION/** à tous les autres clients connectés. Pas de section critique.
  A noter : la signature du message réception a été modifiée par rapport au protocole. Notre serveur envoie **RECEPTION/message/user_émetteur/** pour que les clients recevant le message puissent savoir de qui il provient. Cela ne devrait pas gêner la compatibilité dans la plupart des cas car l'argument *user_émetteur* est placé après les arguments standards du protocole. Il devrait être ignoré par la plupart des clients implémentant le protocole standard. Néanmoins, certaines façons de parser les messages serveur chez le client pourraient bloquer si le nombre d'arguments ne correspond pas exactement à celui du protocole de base.

- **PENVOI/user/message/**
  Le thread envoie le message **PRECEPTION/** à tous les autres clients connectés. Pas de section critique.

Thread de jeu (méthode game_handler du fichier thread_game.c)

```
typedef struct {
    int tour_act;
    char* grille_act;
    pthread_cond_t* event;
    pthread_mutex_t* mutex_timer;
    pthread_mutex_t* mutex_clients;
} boggle_game;
```

A son lancement, ce thread entre dans la boucle des sessions puis des tours.
Il verrouille le *mutex game→mutex_clients*, puis initialise la grille des lettres :

- si l'option -grilles est activée, il prendra dans l'ordre les grilles fournies en arguments de cette option.
- sinon, il générera aléatoirement une grille de 16 lettres à partir des 16 dés fournis dans l'énoncé et de l'algorithme suivant :
  1. crée un tableau d'entiers uniques de 0 à 15 dans un ordre aléatoire.
  2. Pour chaque entier i du tableau, utilise le i-ième dé pour choisir une lettre aléatoirement sur les 6 lettres proposées par le dé.

Le tour commence alors. Le thread envoie le message **TOUR/tirage/scores/** à tous les clients connectés. Puis il crée un thread *timer*, déverrouille le *mutex game→mutex_clients*, verrouille le *mutex game→mutex_timer* et s'endort sur la condition *game→event*.

Le thread *timer* nouvellement créé exécute la méthode *timer_handler*, qui consiste à attendre TEMPS_TOUR secondes, à verrouiller le *mutex game→mutex_timer*, à envoyer un signal sur la condition *game→event*, puis à déverrouiller le *mutex game->mutex_timer*.

Cela réveille le thread du jeu. Dans l'ordre :

1. le thread verrouille *game→mutex_clients* (entrée section critique de lecture et écriture des données métier)
2. envoie le message **RFIN/** de fin de tour
3. calcule les scores et envoie **BILANMOTS**
4. réinitialise à vide les mots proposés de tous les clients connectés.
5. Déverrouille le *mutex game→mutex_clients*
6. attend TEMPS_PAUSE secondes
7. incrément le numéro du tour actuel et passe à la prochaine itération de la boucle tour.

8. (conditionnel) si la boucle de tours se finit (*game→tour_act = nb_tours*),
   - envoie du message **VAINQUEUR/**
   - attend TEMPS_PAUSE secondes
   - verrouille le *mutex game→mutex_clients*
   - réinitialise à 0 les scores de tous les clients
   - déverrouille le *mutex game→mutex_clients*