UPMC
SORBONNE UNIVERSITÉS

Mai 2018

Rapport PC2R - Boggle

Amel Arkoub & Ling-Chun SO

Table des matières

Introduction

Nous souhaitons créer notre propre jeu de lettres multijoueur en ligne, de type Boggle. Nous avons décidé d'écrire le serveur en C et le client en Java. Ces choix se justifient de la manière suivante. Maîtrisant le C et ayant déjà eu l'occasion d'écrire des serveurs dans ce langage, c'est de façon naturelle que nous options pour ce langage. Quant au client, Java propose la bibliothèque d'interface graphique JavaFX, qui est facile d'utilisation et dont l'esthétisme final est incontestable.

Le sujet

Le protocole de l'énoncé est respecté.

Quelque liberté

Modification apportée au sujet

Nous prîmes l'initiative de considérer uniquement la vérification immédiate des mots envoyés par le client au serveur. Ainsi, il est inutile de lancer le serveur avec l'option – ***immediat***, le serveur vérifie toujours que le mot envoyé par le client fait bien partie du dictionnaire dès qu'il le reçoit.

Les extensions

Toutes les extensions obligatoires, à savoir le chat et la vérification immédiate, sont implantées.

Concernant les extensions facultatives, vous pourrez trouver un **client graphique**, c'est-à-dire le cube et la représentation de la trajectoire, ainsi qu'un **client autonome**, tricheur absolu et imbattable, qui vient venter son intelligence incontestable sur le chat. De plus, si vous le souhaitez, **les grilles peuvent être générées automatiquement**.

Génération de grille automatique

Chacun des 16 dés décrits dans le sujet est modélisé par une chaîne de caractères. La grille est alors construite en choisissant au hasard un caractère pour chacun des dés et en tirant au hasard leur position dans la grille.

Description algorithmique du client autonome

Le client autonome calcule toutes les combinaisons possibles du Boggle et envoie les mots corrects au serveur après les avoir vérifiés dans un dictionnaire.
Ce client est très difficile à reconnaître mais peut néanmoins se faire trahir par son ton un peu vantard !

Le client suit cet algorithme :

1) Chargement du dictionnaire et de répliques pour le chat
2) Connexion au serveur avec un nom choisi au hasard (peut être enrichi)
3) Attente d'un tour
4) Calcul de toutes les solutions possibles de la grille
5) Envoi des solutions au serveur

A partir de l'étape 3, le client autonome devient bavard et envoie quelques messages dans le chat.

Calcul des solutions:
Le calcul de toutes les combinaisons possibles s'effectue par un algorithme récursif qui explore toutes les possibilités. Le principe de cet algorithme se repose sur celui du backtracking.
Notons que l'algorithme qui suit est lancé sur chaque case (point de départ de l'algorithme) de la grille.

L'algorithme procède comme ceci :

Entrée :

- *visit* : une matrice de booléen
- *word* : une chaîne de caractères du mot en construction
- *traj* : une chaîne de caractères de la trajectoire en construction
- *row* : position de la ligne
- *col* : position de la colonne

Si word est un mot du dictionnaire et de taille supérieure à 2 alors

On ajoute *word* et *traj* dans les solutions

visit[row][col] ← *true* (Cela permet d'éviter de repartir en arrière et de bouclé à l'infini)

On effectue un appel récursif sur les 8 chemins s'ils sont possibles (haut, bas, gauche, droite, haut-gauche, haut-droite ...), c'est-à-dire non visité et ne sortant pas de la matrice.
Dans ce cas-là, on ajoute aussi la nouvelle lettre et la nouvelle trajectoire

visit[row][col] ← *false* (cela permet de rendre visitable pour la fonction récursive appelante).

Le serveur

Structure de données, variables globales et mutex

Les structure de données

Modélisation du client

Le client est modélisé par la structure ***Info***, qui contient :

- Une chaine de caractères correspondant au *pseudo*
- Un entier correspondant au *score*
- Un pointeur sur liste Liste_mot * correspondant à ses *mots proposés et validés*

Modélisation des listes de mots

Les listes de mots sont le dictionnaire, les mots proposés propre aux clients, et la liste des mots déjà trouvé par tous les clients au cours du tour. Une liste de mots Liste_mot* est modélisée par :

- Une chaine de caractères correspond au *mot*
- Un pointeur vers l'élément suivant Liste_mot* *next*

Les variables globales

Voici la liste des variables globales utilisées dont il nous semble nécessaire de décrire :

- *grille* : chaîne de caractères générée par le processus principal, et lues par les threads qui s'occupent des clients. Les accès concurrents n'ont pas besoin ne génèrent aucune incohérence, car seul le processus principal écrit, et lorsqu'il le fait, le tour est déjà fini.
- *map* : hashmap qui associe un numéro de connexion à un client <int, Info>. Lue par le processus principal, et écrite et lue par les threads s'occupant des clients : un mutex est nécessaire.

- *dejaDit* : liste de Liste_mot * correspondant aux mots trouvés par tous les clients au cours du tour. Ecrite et lue par tous les processus, il est nécessaire d'avoir un mutex.
- *dico* : liste de Liste_mot *. Chargée par le processus principal au lancement du serveur, cette liste n'est accédée qu'en lecture, alors il n'y a aucunement besoin de la protéger.

Gestion des accès concurrents

Comme mentionné précédemment, il fallait absolument empêcher les accès concurrents à la map et à la liste de mot dejaDit.

- *mutex_map* : pour protéger la map. Chaque processus doit prendre le mutex avant de modifier la map, et aussi avant d'envoyer un message aux autres utilisateurs que son client lui-même. En effet, quand on envoie un message aux autres utilisateurs, on doit récupérer tous les numéros de connexion. La modification de la map à ce moment-là peut être problématique, c'est-à-dire la deconnexion ou la connexion d'un nouveau client.
- *mutex_dejaDit* : pour protéger la liste de mot dejaDit, car plusieurs threads pourraient modifier la liste pour ajouter le mot valide de leur client.

Le processus principal

Une fois le serveur lancé, le processus principal attend sur deux descripteurs de fichier : un correspondant à la socket d'écoute sur le port donné en argument (ou sinon celui par défaut), et l'autre sur un timer.

Quand un client veut se connecter, le processus principal va créer un thread qui s'occupera de la connexion avec ce client.

Avec chaque fois que le timer finit, le processus principal va effectuer une tâche. Voici la séquence de tâches réalisées par le processus :

1) Au lancement, le processus envoie le message SESSION/

2) Au bout de 10 secondes, le timer envoie un message au processus via le descripteur de fichier pour lui signaler que le temps imparti est écoulé
3) Le processus charge une grille et envoie à toutes les connexions établies avec des clients le message TOUR/tirage/
4) Au bout de 3 min, le timer envoie un message au processus
5) Le processus envoie RFIN/ et BILAN/motsproposes/scores/ à tous les clients. Toutes les données concernant les mots validés au cours du tour sont effacées.
6) Les étapes 3, 4 et 5 se répètent autant de fois qu'on a de tours par session
7) Le processus envoie VAINQUEUR/bilan/. Les scores de joueurs sont remis à 0.
8) On retourne à l'étape 1

Le thread s'occupant des échanges avec un client donné

Le thread reçoit les différents messages envoyés par son client, et lui envoie la réponse adéquate. Ces messages peuvent concerner le jeu ou le chat. Son architecture se résume en une boucle infinie sur l'attente d'un message envoyé par le client. A chaque message reçu, son traitement et sa réponse adéquate !

Réception de CONNEXION/user/

Le thread récupère la chaîne de caractère *user*, prend le mutex_map, envoie le message CONNECTE/user/ aux autres joueurs, ajoute la structure de données correspondant au nouveau client dans la map, recupère les scores actuels et rend le mutex_map. Finalement, il envoie BIENVENUE/tirage/scores au client.

Réception de SORT/

Le break de la boucle est effectué. Le thread prend le mutex_map, enlève la structure de données correspondant au client de la map_map, envoie DECONNEXION/user/ aux autres joueurs, puis rend le mutex_map. Pour finir, il ferme la connexion avec le client.

Réception de TROUVE/mot/trajectoire/

Le thread vérifie que le mot est de longueur suffisante, que la trajectoire du mot est correcte, que le mot est dans le dictionnaire. Finalement, il s'assure que le mot n'a pas déjà été pris par quelconque joueur, c'est-à-dire en vérifiant sa non appartenance à la liste de mot *dejaDit*. Si le mot est valide, il ajoute ce mot à la liste de mots déjà dits. Pour ce faire, il prend le mutex_dejaDit correspondant, effectue l'ajout puis le rend.

Si au moins une des conditions précédentes n'est pas respectée, il y aura respectivement l'envoi du message :

- MINVALIDE/TAILLE la taille du mot doit être supérieure ou égale a 3/
- MINVALIDE/POS la trajectoire est erronée/
- MINVALIDE/DIC le mot n'appartient pas au dico/
- MINVALIDE/PRI le mot a déjà été propose/

Sinon le message qui sera envoyé est :

MVALIDE/mot/

Réception de ENVOI/message/

Le thread prend le mutex_map et envoie à tous les joueurs RECEPTION/message. Ensuite, il lâche le mutex_map.

Réception de PENVOI/message/user

Le thread prend le mutex_map, cherche dans la map le numéro de connexion correspondant au pseudo *user* et envoie le PRECEPTION/message/user à ce client. Finalement, il rend le mutex_map.

Le client

Le client a été écrit en Java en s'appuyant sur JAVAFX, nous avons donc codé une interface graphique dont nous allons expliciter chaque fonctionnalité.

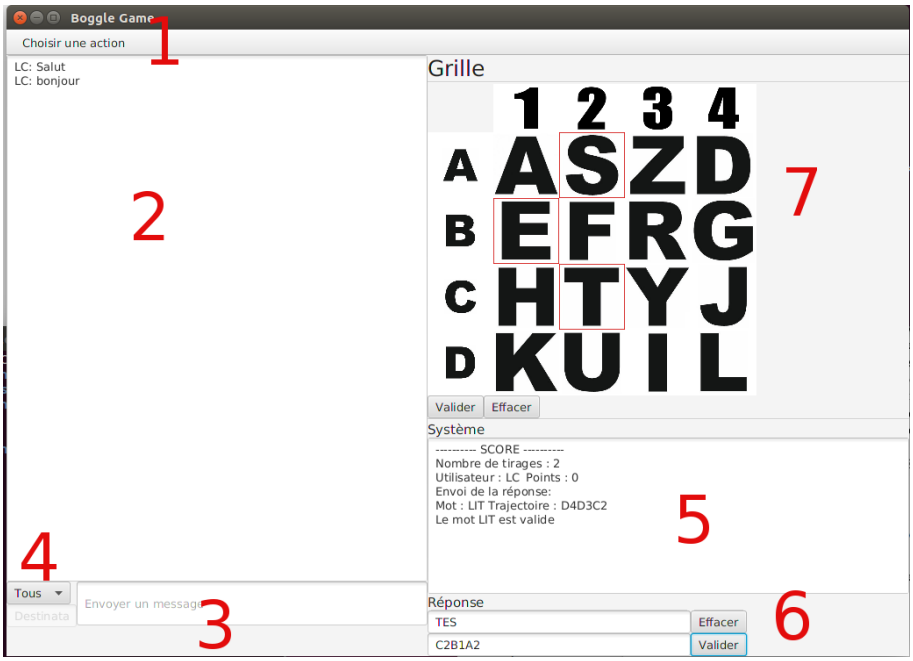

Figure 1 - Capture d'écran du client JAVA

1) Ce menu permet de choisir une action à effectuer:
- Se connecter à un serveur, un popup apparaitra et demandera les informations du serveur ainsi que le pseudo du joueur
- Se déconnecter d'un serveur de jeu
- Quitter l'application

2) Cette zone correspond au chat, les messages envoyés et reçut y sont affichés

3) Cette zone correspond au contenu que vous souhaitez envoyer au chat

4) Ce menu déroulant vous permet de choisir si votre message sera public ou privé, s'il est privé, il y a un champ pour indiquer à qui vous souhaitez l'envoyer.

5) Cette zone affiche les informations relatifs à la session, c'est-à-dire les connexions de joueurs, le début et fin de tour ainsi que les résultats.

6) Permet d'écrire et d'envoyer une réponse (un mot trouvé) au serveur, il faut spécifier le mot et la trajectoire (pas obligatoire cf.7).

7) Cette zone contient la grille du Boggle, il est possible de directement tracer la trajectoire, il suffit d'effectuer des clics successifs dans les cases, un encadrement rouge est afficher dans chaque case sélectionnée, dans ce cas, le mot et la trajectoire est construit au fur et à mesure des clics et est visible dans la zone 6.

Fonctionnement

Cette application peut être divisée en deux parties, l'interface graphique et l'envoie de message, mais aussi la récupération et traitement d'information des messages envoyés par le serveur.

Interface graphique et envoi de messages

Le fichier BoggleWindow.java contient toutes les instanciations des éléments graphiques de l'application du client Boggle.
Lorsque la demande de connexion est envoyée, le client récupère le flux qui sert à envoyer vers le serveur et toute action de l'application, c'est-à-dire la validation d'envoi d'un mot, la validation d'un message écrit dans ce flux, et donc envoie l'information au serveur.

Traitement d'informations envoyées par le serveur

Le fichier GameRunner.java contient le code permettant de récupérer les informations envoyées par le serveur.
Ainsi, celui-ci revient simplement à une boucle qui attend une entrée sur le flux entrant, et après parsing et analyse de l'information, le client effectue l'action adéquate telle que charger une nouvelle grille, récupérer un message, afficher le résultat.
Il est à noter que si l'action doit modifier l'applcation JavaFX, il faut l'englober dans la méthode Platform.runLater() en raison du fait que le Thread GameRunner n'est pas un thread JavaFX.

Fiche d'utilisation du serveur et du client

Le serveur

Un makefile est disponible dans Boggle_PC2R/Boggle_Serveur.

Pour lancer le serveur avec les options de vous voulez, veuillez entrer les commandes suivantes dans une console, à partir de Boggle_PC2R/Boggle_Serveur :

- make
- ./bin/serveur [–port] *num_port* [–tours] *nb_tours* [–grilles] *grille1 … grilleN*

On note que chaque option est facultative, et les valeurs par défaut sont :

port=2006, tours=3, et les grilles sont générées automatiquement

Ainsi vous pouvez lancer le serveur en ne faisant que :

- ./bin/serveur

Ou bien une composition de votre choix, comme par exemple :

- ./bin/serveur –tours 10

Si vous optez pour l'option –grilles *grille1 … grilleN*, sachez que ces grilles seront proposées en boucles.

Pour fermer le serveur proprement, il suffit d'appuyer sur une touche.

Le client

Un fichier build.xml est disponible pour générer une distribution avec ant dans Boggle_PC2R/Boggle_Client.

Pour générer une distribution, veuillez entrer les commandes suivantes dans la console, à partir de Boggle_PC2R/Boggle_Client :

- ant clean
- ant jar

L'exécutable jar se trouvera dans le dossier Boggle_PC2R/Boggle_Client/jar.

Pour lancer le client, veuillez à partir de Boggle_PC2R/Boggle_Client écrire dans la console:

- java -jar jar/Boggle_Client.jar [-serveur] *ip* [-port] *numport*

Vous pouvez soit lancer avec aucun argument comme ceci :

- java -jar jar/Boggle_Client.jar

Ou alors spécifier chaque champ par exemple:

- java -jar jar/Boggle_Client.jar -serveur 127.0.0.1 -port 2018

Le client autonome

Un fichier build.xml est disponible pour générer une distribution avec ant dans Boggle_PC2R/BoggleCheaterClient.

Pour générer une distribution, veuillez entrer les commandes suivantes dans la console, à partir de Boggle_PC2R/BoggleCheaterClient :

- ant clean
- ant jar

L'exécutable jar se trouvera dans le dossier Boggle_PC2R/BoggleCheaterClient/jar.

Pour lancer le client, veuillez à partir de Boggle_PC2R/Boggle_Client écrire dans la console:

- java -jar jar/Boggle_Cheater_Client.jar -serveur *ip* -port *numport*

Par exemple :

- java -jar jar/Boggle_Cheater_Client.jar -serveur 127.0.0.1 -port 2018

Conclusion

Le projet s'articulait entre la conception du serveur en C et du client en JAVA. Chacun des langages offraient des atouts intéressants, comme en C la bibliothèque de gestion des sockets, compliquée d'utilisation mais qui offre un large panel de possibilités. Quant à Java, il s'agit de la simplicité de la conception graphique qu'offre JavaFX.