Boggle
TOP PROJET ESIAL

Julien VAUBOURG && Igor MARTY
<julien@vaubourg.com>

15 mars 2011

Table des matières

1 Avancement du projet

1.1 Fonctionnalités intégrées

La stratégie adoptée a été de ne pas se contraindre à réaliser l'ensemble des options proposées (uniquement les plus pertinentes, dans le cadre d'une diffusion du jeu), mais de se concentrer sur un modèle clients/serveur.

L'objectif étant qu'un même programme puisse à la fois héberger et rejoindre une partie en ligne. Le serveur est un simple joueur, qui décide d'héberger la partie en plus de jouer (moins contraignant qu'un serveur indépendant, et plus adapté aux petits jeux).

Les fonctionnalités principales sont :

- Mode jeu solo
- Mode jeu multijoueurs (joueurs illimités, sur une même machine)
- Mode jeu en ligne (joueurs illimités, en hébergeant une partie ou en rejoignant un serveur distant)
- Proposition systématique des mots non trouvés présent sur le plateau, par le robot en fin de partie
- Interface graphique
- Interface texte (en se contraignant à proposer en texte quasiment tout ce qui est proposé en graphique)
- Possibilité d'afficher les mots sur le plateau, depuis l'interface graphique
- Chronomètre de partie
- Taille du plateau variable (utilisé pour les tests de rapidité des algorithmes)

La version Klingon était prévue, mais le temps a manqué.

Le jeu lance une interface en mode texte par défaut, et une interface graphique en lui passant l'option `--gui`.

Les clients et le serveur peuvent jouer indépendamment en texte ou en graphique.

1.2 Dépendances

Le jeu dépend du PatriciaTrie Java [1], et de JUnit pour son package tests.

Il doit également avoir accès aux deux fichiers *dice-definition.txt* et *ODS5.txt.gz*.

1.3 Évolutions envisageables

- Création d'une version légère spécifiquement cliente, en plus (prouve que c'est vraiment le serveur qui gère tout le jeu à distance, et que le client n'est qu'un terminal)
- Affichage des serveurs trouvés sur un réseau
- Mode solo contre robot, avec des niveaux de difficulté (le robot trouve plus ou moins de mots, plus ou moins longs selon le niveau demandé)
- Mini-chat intégré à la partie
- Choix de la langue
- Amélioration de l'expérience utilisateur (cf. les commentaires tout en bas)

2 *Backtracking* et place de l'intelligence artificielle

2.1 Algorithmes

L'algorithme de *backtracking* est classique. À noter l'utilisation de *PatriciaTrie* au lieu de *ArrayList* pour les listes de mots des joueurs.

Cette optimisation a été réalisée suite à la remarque d'un camarade [2] concernant le temps d'exécution de la recherche de tous les mots du plateau avec une largeur de cinquante dès. Suite à une série de tests de rapidité, le changement de structure de données a permis de passer de trois minutes à deux secondes.

1. http://code.google.com/p/patricia-trie/
2. Tristan STEF-DI-SCULLIO

2.2 Joueur robot

Le projet a initialement été prévu pour que le robot soit un joueur parmi les autres. C'est pourquoi il est traité en tant que tel dans le code.

Le sujet laissait en effet à penser qu'il était question de l'affronter. Or, le robot étant une intelligence artificielle sans bride, il ne peut pas ne pas trouver tous les mots du dictionnaires présents sur le plateau. En appliquant les règles du Boggle décrites dans le sujet (un mot trouvé par plus de un joueur ne rapporte aucun point), le joueur humain ne pouvait pas s'en sortir avec plus de zéro points.

Suite à la remarque adressée aux enseignants [3], il serait question que le robot ne trouve que les mots que le joueur n'a pas trouvé. Malgré tout, il reste qu'il est encore quasiment impossible de le battre. Le changement de stratégie définitif a donc consisté à séparer progressivement le robot des autres joueurs, pour afficher ses mots en tant que « *mots non-trouvés* », sans lui attribuer de score.

Conceptuellement, il en résulte qu'il aurait été plus propre de traiter le robot à part des autres joueurs. Toutefois, cela facilite la communication des mot-trouvés avec les clients, et permet une évolution intéressante du jeu (en permettant de joueur contre le robot, en le bridant selon un niveau demandé).

3 Clients / Serveur

3.1 Un programme polymorphe

Le programme pouvant être à la fois client et serveur, il doit être polymorphe et changer son comportement selon le contexte. Les clients d'une partie délèguent toute la gestion du jeu au serveur (ils ne stockent même pas leurs propres mots), et ne se contentent plus que de transmettre, recevoir et afficher. Ce sont des terminaux.

Ainsi, la difficulté a été d'utiliser un même code pour deux programmes différents, qui communiquent entre eux. Pour faciliter la tâche, plusieurs classes spécifiques au client (le mode par défaut étant serveur, en ligne ou non), permettant de le différencier un minimum.

3.2 *Multi-threading*

Le serveur héberge aussi un joueur. Il doit donc être capable de simultanément communiquer avec les clients, et faire jouer son joueur dans de bonnes conditions. Côté clients, il faut aussi faire jouer le joueur, en communiquant avec le serveur pour mettre à jour les données.

Plusieurs *threads* ont été déployés pour ça : le serveur lance deux *threads* réseaux principaux, chargés de collecter tous les nouveaux joueurs. Le premier les collecte pour leur attribuer un *thread* spécifique qui s'occupera de les ajouter à la partie et de les gérer tout le reste du jeu. Le second les collecte pour leur attribuer un second *thread* de synchronisation permanente des scores.

Chacun de ces deux *threads* est relié directement au joueur par le *socket* créé par les *threads* de collecte. Ils leur sont dédiés et ne mourront qu'en fin de partie : ainsi, les connexions ne s'établissent qu'une fois.

3.3 Un bus au lieu de deux

L'extension clients/serveur a initialement été développée en prévoyant de n'utiliser qu'un seul *SocketServer*, qui n'écoute que sur un seul port (42000). La gestion du jeu (échange des noms de joueurs, échange des mots, envoi du plateau, etc.) et la synchronisation permanente des scores des joueurs devaient donc se partager le même bus de communication, en fonctionnant en parallèle. Après avoir posé verrou sur verrou jusqu'à en perdre la tête, décision a été prise d'attribuer un bus dédié à la synchronisation des scores (port 42001), allégeant du même coup grandement le code (et le nombre potentiel de bogues).

Il y a donc deux ports à ouvrir en entrée sur le commutateur réseau et/ou pare-feu, du joueur qui héberge la partie.

3. Cf. mail du 3 mars 2011

4 Des utilisateurs un peu trop libres

4.1 Menu Swing

Contrairement à la version texte, avec l'interface graphique, l'utilisateur est libre d'aller cliquer où bon lui semble, n'importe quand : le jeu passe alors d'un mode purement séquentiel à un mode événementiel.

La contrainte de proposer les deux interfaces, et la volonté de ne pas séparer strictement le code dans le modèle a posé certains problèmes de conception (alors que dans un cas il suffit d'attendre que le joueur réponde pour passer à l'instruction suivante, il faut tourner en rond et attendre un évènement dans l'autre). Une classe abstraite a été définie pour permettre au jeu de ne pas avoir à se soucier de la vue utilisée, et de gérer le jeu indépendamment des contraintes spécifiques à l'affichage choisi.

Le problème a souvent été résolu par un peu plus de souplesse dans la conception de la vue, ou des variables drapeaux qui sont consultée en permanence pour détecter un évènement de façon séquentielle. Le code aurait été différent si l'interface texte n'existait pas.

Mais surtout, le menu principal du jeu est disponible en permanence. En permanence, il faut être en mesure de traiter ses actions, et de chambouler le déroulement du programme.

4.2 Retour sur les *threads*

Pour résoudre le problème du menu, le jeu principal a été enfermé dans un *thread*. Ceci permettant par la même occasion d'implémenter la fonction de chronomètre, qui, comme en cas d'utilisation du menu, devait tuer le *thread* pour en créer un nouveau.

Mais c'était sans compter les évolutions de Java. À cause du mauvais traitement accordé aux appels des fonctions *stop()* des *threads* par les systèmes d'exploitation, Sun a décidé de rendre cette méthode dépréciée [4]. La solution proposée en alternative est l'utilisation d'un drapeau de vie ou de mort, consulté à chaque tour du *run* du *thread*. Ainsi, un *thread* ne peut plus être coupé brutalement.

Pour gérer ce problème imprévu, il a fallu adapter tout le programme, pour qu'il soit capable à tous les moments plus ou moins critique, de signaler au jeu qu'il doit arrêter son *thread* principal. Un drapeau général d'interruption vérifié un peu partout, ainsi que l'exception *InterruptedException* remontée régulièrement jusqu'au code du modèle ont ainsi été déployés.

4.3 Perte d'un correspondant

Lors d'une partie en ligne, tout peut arriver. Ainsi, le serveur doit être alerté si un joueur client décide d'utiliser son menu pour interrompre la partie en cours. Idem, le client doit être averti si le serveur est parti. Les mêmes problèmes se posent évidemment en cas de perte du réseau. Le tout, toujours, en rendant l'expérience utilisateur la plus confortable et claire possible.

Pour gérer ces cas, une remontée d'exceptions permet au jeu de s'adapter. Côté serveur, le joueur client est désactivé : ceci consister à tourner un drapeau, et à tuer les deux *threads* qui le concernent, pour qu'il n'y ai plus aucune activité réseau avec lui (cette disparition est signalée graphiquement, dans l'interface graphique). Côté client, ne pouvant plus rien faire sans serveur, la partie est interrompue, le jeu réinitialisé, et le joueur est libre de redémarrer tranquillement un nouveau jeu.

4.4 Annuler

Dernière liberté accordée, nous avons pris soin de modifier les dialogues pour laisser en permanence le choix d'annuler ce que l'utilisateur est en train de faire, avec des boutons adéquats. Il a donc également fallu gérer la possibilité d'annulation à chaque étape des différents processus.

4. http://download.oracle.com/javase/1.4.2/docs/guide/misc/threadPrimitiveDeprecation.html

5 Problèmes en vrac

5.1 Patricia n'est pas très appréciée

Le PatricieTrie Java bénéficie d'une absence flagrante de sources de documentation sur Internet. Ainsi, nous n'avons pas réussi à déterminer la solution alternative à la directive *StringKeyAnalyzer.INSTANCE*, dépréciée, et utilisée à chaque appel du constructeur.

5.2 Risques de bavures

Les *threads* de synchronisation ne connaissent pas l'identité du joueur dont ils s'occupent. Ainsi, lorsqu'ils sont témoins d'un problème de communication, ils doivent rechercher le joueur dans la liste à l'aide de son adresse IP, pour le désactiver (avant de se tuer). Or, dans le cas d'une partie sur Internet, deux joueurs afficheront la même adresse. Le problème est loin d'être insoluble, et sera probablement résolu avant de publier le projet au delà du cadre de la TOP.

5.3 Un peu plus de confort

Afin d'améliorer l'expérience utilisateur, il aurait été souhaitable que le nom du dernier joueur indiqué soit proposé comme nom par défaut lors d'une nouvelle partie. Idem pour l'adresse du serveur. Ces problèmes ont été traités, mais le *setInputValue()* des dialogues, contre tout bon sens, ne fonctionne pas, malgré des exemples similaires trouvables sur Internet.

5.4 Ridicules infobulles

En fin de partie, en utilisant l'interface graphique, le joueur peut cliquer sur les mots de la liste pour voir le mot sur le plateau. Si il se contente de le survoler sur la liste des mots trouvés, il peut voir qui sont les joueurs qui l'ont trouvé. Malheureusement, les infobulles de Swing sont bien trop peu réactives et bien trop peu visibles : cette solution a été adoptée, mais des infobulles personnalisées seraient plus appropriées.

5.5 How-to test ?

Une série de tests unitaires ont été réalisés au début du projet, avant que les cours sur ceux-ci ne commencent. Leur façon de fonctionner est probablement trop compliquée (utilisation de méthodes UNIX pour vérifier les tests). Il y aurait encore beaucoup de tests à faire (ceux-ci ne testent à peu près que ce qui est dans le cadre du projet), mais nous n'avons pas réussi à imaginer des tests des communications réseaux, ou, pire, de l'interface graphique.

6 Temps par étape

Certainement plus de vingt heures nocturnes pour le codage, peut-être la moitié ou autant pour tester (pour les tests unitaires, mais surtout pour les tests des différents cas du clients/serveur et des événements de l'interface graphique).

Quant à la conception, elle a été morcelée. La structure du noyau du projet ayant été plus ou moins imposée, la conception a quasiment été nulle. Pour le reste (principalement l'adaptation du programme pour le clients/serveurs), il aura fallu quelques heures pour trouver un modèle compatible avec les contraintes.