Rapport de projet :

Programmation Concurrente Réactive et Répartie

Robot Ricochet

Louis Boutin

Master STL

Université Pierre et Marie Curie

2015-2016

1/ Introduction - Description du problème

Le projet demande l'implémation d'un clone du jeu de plateau Ricochet Robot en multijoueur suivant une architecture client serveur.

Le projet a été réalisé avec les langages Haskell et Python (avec PyQt qui effectue un binding vers la célèbre librairie graphique Qt de C++)
La seule extension implémentée est celle d'un système de chat efficace qui utilise les messages "DIREATOUS/message" du client vers le serveur et "CHAT/user/message" du serveur vers le client. J'ai également rajouté au protocole le message "SERVEUR/" qui permet d'envoyer des messages d'information quelconques que j'affiche également dans le chat de l'application

2/ Implémentation du serveur

J'ai choisi de réaliser le serveur en Haskell et de saisir ainsi l'opportunité de programmer en langage fonctionnel. Bien que Ocaml ait été utilisé au sein de l'UE, je m'étais déjà essayé à la programmation fonctionnelle en Haskell que j'apprécie pour le clarté de sa syntaxe. C'est donc avec ce langage que j'ai souhaité poursuivre mes efforts. La gestion des verrous (**MVar**) y est très élégante. Haskell implémente d'autre part le

STM ou *Software Transactional Memory* qui permet de traiter la concurrence de manière analogue aux bases de données via des transactions dont on écrit les effets dans un log avant de les rendre effectives. Cette abstraction simplifie grandement la gestion opérations concurrentes mais bien que j'aie commencé le projet en l'utilisant, j'ai préféré recommencer en utilisant des verrous afin de mieux coller au contenu de l'ue pc2r.

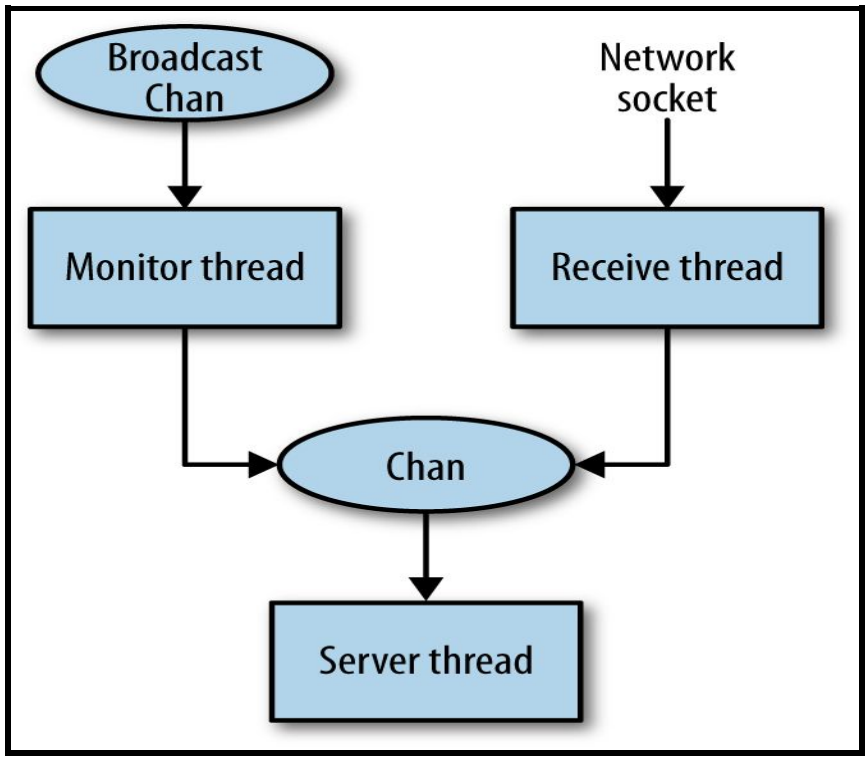
L'implémention se divise en différents module chacun contenu dans un fichier du même nom avec l'extension ".hs" :

- ***Main*** : Module principal permettant de lancer le serveur, d'attendre les clients
- ***ServerData*** : Contient les déclaration de tous les types et structures de données spécfiques au projet ainsi que leur constructeur pour ceux qui le nécessitent
- ***Level*** **:** Contient toutes les fonctions permettant de générer des problèmes aléatoires et de modéliser l'état de la grille de jeu sous forme de matrice à savoir placement des murs
- ***Phase :*** Contient les fonctions de transition d'une phase de jeu à l'autre ainsi que la gestion des comptes à rebours
- ***Solution*** : Contient les fonctions permettant de traiter les solutions envoyées par les clients et donc notamment un algorithme permettant de vérifier si une solution est correcte ou non à partir d'une grille, d'une instance de problème et d'une séquence de coups
- ***Communication*** : Contient les fonctions assurant la transmission des messages aux clients à savoir la traduction des données en chaînes de caractères, les fonctions d'envoi de type broadcast ou client particulier.
- ***TurnTools*** : petit module contenant quelques fonctions permettant de manipuler la structure de données correspondant au tour de jeu

Le répertoire contient également quelques autres fichiers Haskell qui sont des simplement des librairies ou extraits de librairie importées directement dans le répertoire pour des soucis de conflit avec d'autre modules standards Haskell.

Architecture du serveur :

Le serveur se met en attente de connexion de la part d'un client, et lui fournit un thread d'échange. A la réception d'un message d'authentification "CONNEXION" suivi d'un nom de joueur qui n'est pas déjà en cours d'utilisation, on procède à la gestion du client par trois threads suivant l'architecture ci dessous. Si le nom fourni est déjà utilisé, le client reçoit un message d'erreur. Si le serveur reçoit tout autre message de la part du client, la communication est coupée.



L'application utilise un canal de broadcast qui est utlisé lorsque l'on souhaite envoyer un message à tous les clients. Chaque client reçoit une copie de ce canal à l'authentification.
Cette architecture nous évite d'accaparer un verrou sur l'ensemble des clients pour leur envoyer individuellement le message à transmettre. On écrit simplement le message sur le canal de broadcast et le thread moniteur de chaque client se charge de transmettre le message depuis le canal de broadcast vers le canal du client.

Le canal de réception gère la reception et le traitement des messages reçus de la part du client. Le thread serveur se charge quant à lui de lire les messages destinés au client et de lui envoyer

Modélisation du problème et vérification de la solution :

Les grilles de jeu sont stockées sur le serveur sous forme de fichiers texte qui suivent la même forme que celle indiquée sur le protocole à savoir (x,y,z) avec x,y les coordonnées de la cellule et z la position du mur parmi (**H**aut | **D**roite | **B**as | **G**auche). C'est également sous cette forme que la grille est conservée dans la structure de données de la session.
Cette écriture ne permet pas cependant de générer une instance de problème ou des tester une solution. Lors de ces deux opérations de vérification et de génération, on construit une matrice 16x16 où chaque cellule a la forme *(Bool,Bool,Bool,Bool)* où chaque booléen indique respectivement la présence d'un mur en haut, à droite, en bas et à gauche. On a veillé a placé de murs implicites sur toute la bordure de la matrice.

Afin de choisir une case adéquate pour la cible on cherche simple de manière aléatoire une cellule contenant deux murs consécutifs et donc deux booléens ***True*** consécutifs.

Afin de tester une solution on utilise cette même matrice. En partant d'une séquence de mouvements et des positions initiales, on calcule les nouvelles positions des robots après chaque mouvement. On incrémente les coordonnées du robot choisi dans la direction choisie jusqu'à rencontre un mur ou bien un autre robot. Une fois l'intégralité des mouvements traités, on compare simplement la position de la cible et la position du robot supposé l'atteindre. Ainsi si la cible est atteinte pendant la séquence mais pas à la fin, la solution est considérée mauvaise.

3/ Implémentation du client

J'ai d'abord voulu réaliser le client en Javascript mais il me semble pas qu'il existe de librairies permettant la connection directe par socket. Il existe bien sûr les WebSocket mais je ne pensais pas parvenir à les utiliser côté serveur.

J'ai donc choisi d'implémenter le client en Python en utilisant le module ***Pyqt*** dans sa version 4 qui assure le binding entre le langage Python et l'API d'interface graphique **Qt** développée en C++.
J'ai déjà eu l'opportunité d'utiliser Qt pour la réalisation d'interface graphique notamment lors de quelques tp de l'ue IHM. En revanche mon expérience en C++ se limite uniquement à Qt et la modélisation de l'état jeu côté client aurait été fastidieuse pour moi ajoutée à l'apprentissage de Haskell. De plus, afin de palier au fait que je réalise le projet seul, je préférais exploiter la rapidité de développement qu'offre Python tout en profitant de la richesse de Qt qui est largement supérieure à d'autre librairies graphiques comme **Tkinter** ou **Pygame**.

Listing des classes utilisés :

- **mainWindow** (fichier "Client.py") correspond au widget principal. C'est dans cette classe qu'on gère la connexion au serveur et la réception des messages. C'est aussi ici qu'est gérée l'intégralité des feuilles de styles et du layout de l'application.
  Cette classe principale dispose de plusieurs sous Widgets dont certains utilisent leur propre sous-classe.

  Gestion du chat et des messages d'information
  J'ai utilisé un widget ***QTextEdit*** afin de gérer le système de chat dans l'attribut ChatPanel. En outre des messages entre clients, ce widget est également utilisé pour afficher différents messages sur l'état de la partie comme les changements de phases, la réussite ou non d'un joueur à résoudre le problème etc..
  Ce widget assure notamment l'utilisation du HTML dans le texte affiché ce qui me permet d'utiliser un jeu de couleur et de police pour différencier les messages d'informations, les messages du client, les messages des autres clients.

  Gestion du compte à rebours

J'ai utilisé le widget QTimer couplé à un QProgressBar pour modéliser le compte à rebours des différentes phases de jeu. Le QTimer est initialisé, lancer ou stoppé suivant les phases de jeu et se contente d'incrémenter la valeur de la barre de progression en tick de une seconde.
La barre de progression voit son intervalle de valeurs et son texte modifié selon la phase en cours s'anime à la manière d'un sablier

- **GameBoard** (Gameboard.py) : Gestion de la grille de jeu, de la position des murs, des robots et de l'animation des solution. Cette classe contient donc principalement des fonctions de dessins, stocke l'information sur la grille de jeu et permet d'animer simplement les solutions soumises par les joueurs en utilisant une modélisation de la grille analogue à celle utilisée cotée serveur.
  On utilise donc une séquence de 4 bits pour chaque cellule où chaque bit correspond à la présence d'un mur. On modifie les murs à l'aide de ***OU*** logique. On déplace alors les robots et, pour chaque direction de mouvement, on observe la présence ou non d'un mur via un ***ET*** logique.
  ex. *0001 & matrice[x][y]*
  Cette opération permet de tester la présence d'un mur à gauche de la cellule aux coordonnées x,y. On couple ces opérations logique à la comparaison avec les positions des autres robots pour procéder à l'animation des solutions.
- **BiddingBoard** (BiddingBoard.py) : Cette classe gère le tableau des enchères et stocke sous la forme d'un dictionnaire la mise de chaque joueur. A chaque nouvelle enchère le dictionnaire est triée par ordre croissant afin de faire apparaître la meilleure mise en haut du tableau
- **ScoreBoard** (ScoreBoard.py) : Gestion du tableau des scores. Ce widget affiche simplement le score obtenu par chaque joueur dans la session courante ainsi que le numéro du tour

Le répertoire contient également quelques fichiers images utilsées pour l'affichage des robots sur la grille.

4/ Manuel utilisateur

Serveur : Il suffit de compiler avec ghc le fichier Main.hs et de lancer l'exécutable obtenu afin de lancer le serveur local sur le port 2016. Il faut indiquer cependant l'option -XRecordWildCards qui permet d'utliser la notation {..} en Haskell qui permet d'accéder à tous les champs des structures

ghc -XRecordWildCards -o main Main.hs

Il peut être nécéssaire d'installer différents modules via le gestionnaire de paquets haskell *cabal.*

Client : Il suffit de lancer avec python3 le fichier "Client.py" A l'exception de Pyqt4 qui est libre et se télécharge et s'installe facilement, les librairies utilisées sont les librairies standards de Python à savoir ***sys*** et ***re*** pour les expressions régulières