PC2R – Rapport de projet

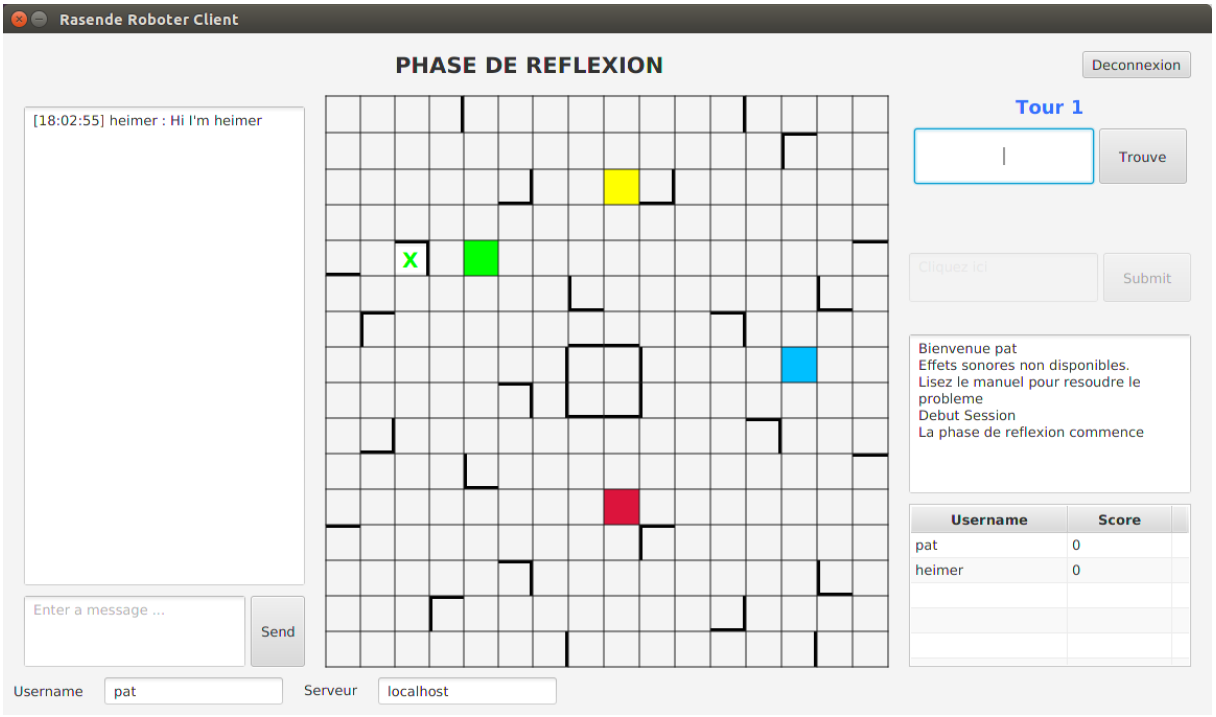
Ricochet Robots

1. Description du jeu

Dans notre version du jeu de plateau Ricochet Robots (originellement Rasende Robot), le jeu est composé d'une matrice 16x16 cases, de 4 pions robots et d'une case cible, initialement placés selon une énigme donnée. Les cases du plateau peuvent être vides ou contenir des murs.
Le but du jeu est de faire arriver le robot qui a été désigné dans l'énigme, à la case cible. Pour cela le joueur doit s'aider des robots et des murs. Les robots ne peuvent se déplacer qu'en ligne droite, vers le haut, la gauche, le bas et la droite et ne s'arrêtent qu'au moment où ils rencontrent un obstacle (un mur ou un autre robot).

Le jeu se déroule en sessions. Chaque session est composée en un nombre indéfini de tours. Chaque tour se décompose en 3 phases :
- Une phase de réflexion : dans laquelle les joueurs ont 5 minutes pour proposer le nombre de coup d'une solution à partir d'un plateau et d'une énigme.
- Une phase d'enchère : qui démarre à la fin des 5 minutes de réflexion ou lorsqu'un joueur trouve une solution, dans laquelle les autres joueurs peuvent enchérir en proposant un nombre de coups différent.
- Une phase de résolution : dans laquelle les joueurs, en fonction du nombre de coups qu'ils ont proposé, ont un à un 1 minute pour proposer leur solution, tant qu'une solution n'a pas été validée par le serveur. Si une solution est correcte, la phase de résolution se termine et le joueur ayant proposé cette dernière gagne 1 point.

Le jeu étant multi-joueurs, le but est d'arriver à un score objectif de 10 avant tous les autres. Une fois le score objectif atteint, une nouvelle session se lance.



2. Serveur – Patrick Tran

Le serveur a été réalisé en C. Il utilise principalement les bibliothèques Sys/Socket, Pthread et String. Nous détaillerons les choix d'implémentation qui ont été faits pour le code du serveur qui est réparti en 5 fichiers C, accompagné de 4 fichiers entête.

2.1 Etablissement du serveur

L'établissement du serveur se fait grâce à la bibliothèque Sys/Socket. Le serveur utilisé est un serveur de type connecté, dont l'adresse est l'adresse locale, écoutant sur le port 2016 et dont le nombre de connexion correspond à SOMAXCONN, une variable déterminant le nombre maximum de connexions possibles, définie dans la bibliothèque. La socket associée au serveur est réutilisable, même en cas d'utilisation récente (ce qui est très utile en phase de tests).

2.2 Connexion

Les connexions se font dans la fonction main. A chaque nouvelle connexion, un thread client est créé et se charge de la réception et l'envoi de messages. Un message est envoyé au client lorsqu'il essaye d'utiliser un nom d'utilisateur déjà pris.

2.3 Structures de données

Pour la gestion des utilisateurs, nous avons créé une structure client (cf. **clientlist.c/clientlist.h**) contenant :

- Le nom
- La socket
- L'id de la thread
- Le score
- Le nombre de coups proposé

du client.
Ces clients sont stockés dans une liste chaînée. Il y a en tout 2 listes : l'une contenant les utilisateurs en train de jouer nommée **clients**, l'autre contenant les utilisateurs en attente d'un nouveau tour ou en cours de connexion, nommée **file_attente**. Les utilisateurs se connectant en plein milieu d'un tour sont directement mis dans la file d'attente et sont transférés vers la liste des joueurs lorsqu'un nouveau tour se lance. Les utilisateurs connectés (du point de vu socket) n'ayant pas encore trouvé de nom d'utilisateur valide, sont aussi dans la file d'attente mais ne peuvent être transférés vers la liste des joueurs tant que leur nom n'a pas été initialisé.

2.4 Traitement des commandes

Les commandes sont traitées directement par les threads clients. Ces threads découpent et récupèrent les commandes en récupérant les chaînes de caractères entre les symboles « / » des messages reçus des clients. Pour éviter de traiter deux commandes critiques qui arrivent en même temps (par exemple SOLUTION), nous utilisons un mutex (**mutex_conflit**) pour traiter ces commandes sans avoir d'incohérence dans le déroulement du jeu.
Toutes les fonctions de traitement de commande se trouvent dans le fichier **funcserver.c**.

2.5 Déroulement du jeu

Un tour étant composé de plusieurs phases, avec à chaque phase un temps limité, il a fallu utiliser des threads permettant la gestion de ces phases. Nous avons une variable globale permettant de savoir dans quelle phase le jeu est actuellement.
Les threads de gestion de phase lancent eux-même un thread servant de timer, et attendent une condition (envoyée par le traitement d'une commande ou la thread timer) qui indique la fin de la phase.

Notre serveur contient en stock 3 plateaux différents. A chaque nouvelle session un plateau est tiré aléatoirement. Les énigmes sont aussi générées aléatoirement en fonction du plateau. Le fichier **board.c** regroupe toutes les fonctions et structures permettant de gérer les plateaux et énigme, y compris la simulation d'une solution permettant de déterminer si elle est correcte. La structure **board** représentant un plateau, est constitué d'une matrice de **square** représentant une case et contenant des variables indiquant la présence de murs et de robot présent, les positions des différents robots et la couleur du robot à amener à la cible.
La fonction de simulation utilise le plateau, l'énigme et la solution donnée. Elle lit dans la solution tous les déplacements et effectue ces déplacements au fur et à mesure. Une fois la solution lue, elle compare la position finale du robot à amener à la cible et la position de la

cible pour déterminer si cette solution est bonne. Enfin, si cette solution est bonne, le nombre de coups effectué (stocké en variable globale) est comparé à au nombre de coups annoncé pour pouvoir être validée et accorder un point au joueur.

2.6 Extensions

2.6.1 Messagerie instantanée

La messagerie instantanée a seulement nécessité l'ajout d'une commande « SEND ». Lorsque le serveur reçoit cette commande, il envoie le message émis à tous les utilisateurs sauf à l'émetteur.

2.6.2 Phase de connexion

Dans la toute première version du serveur, nous avions mis en place une phase de connexion qui se lançait lorsque deux joueurs se connectaient. Cette phase de connexion qui durait 30 secondes, permettait d'attendre d'autres connexions avant de lancer la première session et permettait ainsi d'éviter de jouer le premier tour en duel avec un seul autre joueur.
Malheureusement pensant que cette phase n'était pas permise, nous ne l'avons pas laissé par crainte que le serveur soit incompatible avec un client que le correcteur utiliserait pour le tester.
Cette phase de connexion aurait été similaire à la phase d'enchère, qui n'attend que la fin du timer. A la fin de cette phase, les utilisateurs qui se seraient connectés pendant, auraient pu jouer sans passer par la file d'attente.

2.7 Difficultés

Les principales difficultés ont été rencontrées pendant la phase de test. Elles concernent surtout la synchronisation des threads entre elles et l'accès concurrentiel aux variables globales. Par défaut, les threads stoppés ne relâchent pas les mutex et conditions sur lesquels ils attendent et cela s'est conclu en de longues heures de débuggage.
La définition de multiples mutex nous a semblé judicieuce pour être sûr de ne pas avoir de problèmes en utilisant quelques mutex pour le grand nombre de variables globales que nous utilisions.

2.8 Utilisation

Pour utiliser le serveur, il suffit de compiler grâce au fichier **Makefile**, puis de lancer l'exécutable **main** qui s'est créé dans le dossier **bin/**.

3. Client Java – Ladislas Halifa

Le client du jeu Ricochet Robot a été réalisé avec le langage de programmation Java en version 8. Pour le développement de l'interface graphique, j'ai fait le choix d'utiliser la bibliothèque JavaFX plus récente et plus simple à utiliser que la bibliothèque Swing à mon gout. Les extensions implémentées sont : l'intégration d'un module de chat au jeu, permettant aux joueurs de communiquer entre eux, ainsi qu'un Client autonome se faisant passer pour un humain. Malheureusement celui-ci ne peut pas calculer une solution de manière combinatoire, il ne fournit que des solutions générées aléatoirement, mais il nous a été utile lors de nos différents tests. Ces derniers ont été réalisés en début de développement sur un serveur factice permettant de recevoir les commandes envoyées par le client et d'y répondre à la main, puis sur le serveur en C fourni en fin de développement.

3.1 Mode d'emploi

Le programme se lance en ouvrant l'exécutable RasendeRoboter.jar, pour pouvoir utiliser les effets sonores, ce dernier doit être dans le même dossier que celui contenant le dossier 'audio'.

L'interface graphique du client est composée d'une fenêtre unique. Celle-ci se décompose en 4 parties :

- Sur la gauche la zone de discussion
- Au centre le plateau de jeu
- Sur la droite la zone de contrôle composée de zones de saisies permettant les interactions de l'utilisateur, d'une zone de texte pour les informations émises par le serveur et d'un tableau pour l'affichage des scores des joueurs ayant participés à la session
- En bas la zone de connexion proposant à l'utilisateur d'entrer son pseudo et l'adresse du serveur

Des effets sonores sont disponibles dans le programme, le code responsable du chargement des fichiers audio ne lève pas d'exception uniquement sur mes deux machines personnelles et je n'ai pas trouvé comment les rendre disponibles sur une machine à la PPTI ou sur la machine personnelle de mon binôme. Les classes utilisées pour les effets sonores sont les classes Media et MediaPlayer du package javafx.scene.media. Les exceptions levées lors de l'initialisation du programme sont attrapées, l'utilisateur est informé que les effets sonores sont indisponibles et le programme ne tente ensuite jamais de lire les fichiers audio.

3.1.1 Connexion

A son lancement, le programme affiche la fenêtre principale, mais seule la zone de connexion est disponible, les zones de discussion et de saisies sont désactivées et aucun plateau n'est affiché. Si la tentative de connexion échoue, un message d'erreur indique la raison de l'échec (champ non ou mal rempli, serveur non disponible, nom d'utilisateur déjà utilisé etc..). Si la tentative de connexion réussie (c'est-à-dire si le serveur y répond par

‘BIENVENUE’), la zone de connexion n’est plus utilisable mais reste visible pour afficher le nom d’utilisateur choisi ainsi que l’adresse du serveur et la zone de discussion devient active. Pour discuter avec les autres joueurs, il suffit d’entrer son texte puis d’envoyer son message avec la touche ‘Entrée’ ou bien de cliquer sur le bouton ‘Send’.

Le plateau de jeu s’affichera dès la réception de ‘SESSION’. Si l’utilisateur a rejoint la session au milieu d’un tour de jeu, il commencera à participer au tour suivant, pendant ce temps, il peut seulement discuter avec les autres joueurs, mais il n’a aucune information sur l’état courant de la session et ne sera pas notifié par le serveur des enchères et des propositions de solutions par les autres joueurs.

3.1.2 Phase de réflexion et d’enchère

Dès la réception de ‘TOUR’, la phase de réflexion commence : l’énigme à résoudre s’affiche sur le plateau de jeu, le tableau des scores se met à jour, et la zone de saisie du nombre de coups devient active. Si l’utilisateur souhaite annoncer qu’il a une solution, il saisit le nombre de coups et valide avec ‘Entrée’ ou clique sur le bouton ‘Trouve’. Sinon si un autre utilisateur annonce qu’il a une solution (‘ILATROUVE’), ou bien si la phase de réflexion arrive à son terme (‘FINREFLEXION’) alors la phase d’enchère commence : la zone de saisie du nombre de coups permet désormais d’enchérir et le bouton ‘Trouve’ devient le bouton ‘Enchérir’. Si l’utilisateur décide d’enchérir, il suit exactement la même procédure que lors de la phase de réflexion.

3.1.3 Phase de résolution

A la réception de ‘FINENCHERE’, la phase d’enchère est terminée et la phase de résolution commence : la zone de saisie d’une enchère est désactivée. Si l’utilisateur est le joueur actif, le programme le prévient avec un message dans la zone de texte des informations du serveur, et la zone de saisie d’une solution est activée. Pour entrer sa solution dans la zone de saisie, l’utilisateur doit taper la série de coups. Pour taper un coup, il doit d’abord taper la première lettre de la couleur du robot qu’il souhaite déplacer (R, B, V ou J) puis taper sur une flèche directionnelle pour indiquer la direction du déplacement. Le programme empêche l’utilisateur de saisir une solution erronée : il doit commencer sa solution par la couleur du robot, s’il essaie de taper deux fois de suite la couleur d’un robot ou sa direction, sa deuxième saisie sera ignorée, et enfin s’il ne termine pas sa solution par une direction il ne pourra pas soumettre sa solution. Pour soumettre sa solution, il peut appuyer sur la touche ‘Entree’ ou bien cliquer sur le bouton ‘Submit’. Si l’utilisateur n’a pas envoyé sa solution au bout d’une minute le serveur répond par ‘TROPLONG’ et la zone de saisie d’une solution se désactive. Sinon une fois sa solution soumise, la zone de saisie d’une solution se désactive également, et l’animation de sa solution commence. A la fin de l’animation le serveur notifie le client avec ‘BONNE’ si la solution est acceptée, ‘MAUVAISE’ si la solution est refusée ou ‘FINRESO’ s’il ne reste plus aucun joueur. Lorsque qu’il n’est pas le joueur actif, l’utilisateur peut visualiser les solutions proposées par les autres joueurs sur le plateau et à la fin de l’animation, en fonction de la commande envoyée par le serveur, le programme notifie par un effet sonore et par un message dans la zone d’information si la solution a été acceptée par le serveur. Si la solution est acceptée par le serveur ou s’il ne reste plus de joueurs actifs,

le programme retire les robots et la cible sur plateau de jeu et attend la réception de 'TOUR' pour lancer un nouveau tour et mettre à jour le tableau des scores, sinon il replace les robots à leurs positions initiales et attend la réception de 'SASOLUTION' pour afficher la solution du prochain joueur.

3.1.4 Particularités du client

Concernant l'affichage d'une solution, nous avons choisi coté client de mettre 7,5 secondes pour l'animation d'une solution quelconque. En conséquence plus une solution aura de déplacements (case par case), plus les robots se déplaceront vite sur le plateau. Cela permet au serveur d'attendre 8 secondes avant de notifier l'utilisateur s'il a accepté ou non une solution directement à la fin de l'animation et d'éviter soit une attente trop longue pour l'utilisateur, soit de recevoir la réponse du serveur avant la fin de l'animation. Avec un autre serveur envoyant directement sa réponse au client, notre client prendrait un retard d'environ huit secondes pour la phase de réflexion. Si aucun utilisateur n'annonce qu'il a trouvé une solution pendant cet intervalle de temps, cela ne posera un problème qu'à l'utilisateur de notre client qui serait pénalisé. En revanche si joueur connecté sur le serveur avec un autre client que le nôtre annonce qu'il a trouvé une solution pendant cet intervalle de temps, notre client pourrait avoir un comportement indéterminé et ne plus être fonctionnel. Nous n'avons pas pu le tester avec un autre serveur et nous ne savons donc pas comment il réagirait.

3.2 Modélisation et implémentation

L'architecture du client suit le design pattern Modèle-Vue-Contrôleur. J'ai fait ce choix afin de séparer et de simplifier le développement de l'application.

Le Modèle est représenté par toutes les classes du package rasendeRoboter. Il gère toute la partie logique de l'application :

- le plateau de jeu (Plateau.java) est constitué d'un tableau 16x16 de cases (Case.java), d'une énigme (Enigme.java), et d'un ensemble de Points représentant les positions actuelles des robots. Il permet de gérer les déplacements des robots et de positionner les murs du plateau, etc.…
  Le système de coordonnées utilisé est le suivant : la case (0,0) représente le coin en haut à gauche du plateau, (15,15) représente le coin en bas à droite du plateau, (0,15) représente le coin en haut à droite du plateau et la case (3,7) est située à la quatrième ligne et à la huitième colonne du tableau.
- les scores des joueurs (Bilan.java)

La Vue qui s'occupe d'afficher les différents composants graphiques pour communiquer avec l'utilisateur est représentée par le fichier Game.fxml présent dans le package application.

Le Contrôleur représenté par les classes Client, Outils et Protocole du package application permet d'initialiser l'interface graphique (boutons, champs de saisies) et de calculer sur le modèle les divers événements générés par les interactions de l'utilisateur avec l'interface. Il s'occupe aussi de la communication avec le serveur, d'un coté en traduisant en commandes

les actions de l'utilisateur, et de l'autre en réceptionnant les réponses du serveur qui démarreront des calculs sur le modèle et des modifications sur la vue. La classe principale de l'application est la classe Client.

3.3 Concurrence

Le programme nécessite d'exécuter plusieurs taches en parallèle : mettre à jour les éléments de l'interface graphique, permettre à l'utilisateur d'interagir avec l'interface à n'importe quel moment, et être en écoute permanente du serveur. J'utilise l'API native de Java ainsi que JavaFX Application thread car l'interface graphique JavaFX n'est pas thread-safe et ne peut être accédée et modifiée que par cette dernière.

Le principal thread de l'application est un objet Receive (classe interne de Client) héritant de la classe javafx.concurrent.Service, dont le rôle est d'exécuter la javafx.concurrent.Task à l'écoute continue des commandes du serveur. Lors de la réception d'une commande, le traitement de cette commande est effectué dans un thread secondaire. Le traitement des commandes 'CHAT', 'CONNEXION' et 'DECONNEXION' est séparé des autres commandes afin de pouvoir notifier l'utilisateur de la connexion/déconnexion d'un utilisateur ou d'un message reçu à tout moment, surtout lors de l'animation d'une solution qui aurait pu monopoliser l'interface utilisateur, voir ci-dessous. Le traitement des autres commandes s'effectue les unes à la suite des autres grâce à une méthode synchronized afin de s'assurer que les mise-à-jour de l'interface graphique associées aux différents calculs sur le modèle pour la réception d'une commande du serveur s'exécutent en exclusion mutuelle. Enfin les mise-à-jour de l'interface sont toutes effectuées par la méthode runLater de la classe javafx.application.Platform qui permet de ne pas freeze l'application lors des mises à jour.

3.4 Communications Client-serveur

Comme spécifié dans le sujet du devoir, la connexion s'appuie sur un protocole TCP sur le port 2016. Celle-ci est effectuée au moyen des java.net.Socket et les lectures et écritures sont réalisées par des canaux tamponnés. Clients et Serveur échangent avec un protocole texte. La version du protocole utilisée est celle présente dans le sujet du devoir du 10.3.16, enrichi de la commande SEND/user/message pour un échange du client vers le serveur, et de la commande CHAT/user/message pour un échange du serveur vers les clients, excepté user l'expéditeur du message. Les données sont transmises vers le serveur en utilisant un PrintStream, crée de façon à ce qu'il flush automatiquement le buffer à chaque écriture. Les lectures sont quant à elles reçues par un BufferedReader avec la méthode readline(), les commandes échangées se terminant toutes par \n. Par sécurité, les différentes commandes sont toutes présentes dans le fichier Protocole.java en tant qu'attributs static final et le programme n'envoie une commande au serveur en n'utilisant uniquement les méthodes static de la classe Protocole.

3.5 Difficultés

La principale difficulté dans le développement du client a été le développement d'une interface graphique simple, efficace et fluide. En effet j'étais tout d'abord parti pour

utiliser la bibliothèque Swing que j'avais déjà eu l'occasion d'utiliser l'année dernière dans les UE de LI314 et LI357, mais n'étant pas satisfait du résultat et du temps perdu à écrire en code Java les quatre composants principaux de l'interface puis de les agencer comme je le souhaitais, je me suis tourné vers la bibliothèque Java FX que je ne connaissais pas. Je suis alors reparti de zéro quand les autres binômes commençaient déjà à tester la connexion Client-serveur. Mais ce fut un mal pour un bien, car le langage FXML m'a permis de construire facilement l'interface graphique, de plus Java FX n'est pas si éloigné de Swing et le résultat obtenu me satisfait. Une fois cette étape réalisée, l'implémentation complète du modèle et du contrôleur fut assez rapide et stressante car il ne me restait plus qu'une semaine et demie pour terminer et debugger le client avant la date limite.