Chapitre 1

Serveur OCaml - Mathieu Chailloux

Le serveur a été réalisé en OCaml (version 3.11.2, pas besoin de la 4.00). Il utilise les modules Thread, Unix, et Str (pour les expressions régulières). Les extensions réalisées sont la discussion instantanée, les comptes utilisateurs, et les spectateurs. Tout est centralisé dans le fichier my_serveur.ml, bien qu'il eût été possible de séparer les différents modules en fichiers distincts. Nous détaillerons par la suite les choix d'implémentation qui ont été faits.

1.1 Etablissement du serveur

L'établissement du serveur s'est fait en s'appuyant sur le chapitre 20 du livre Développement d'Applications en Objective Caml (Client/Serveur). Ainsi, pour permettre l'utilisation simultanée des modules Thread et Unix, on utilise le module ThreadUnix pour la communication (les fonctions de lecture/écriture sur les sockets sont donc redéfinies). Enfin, la socket associée au serveur est réutilisable, même en cas d'utilisation récente (ce qui est assez utile en phase de tests). Il est possible de fournir l'adresse sur laquelle on veut lancer le serveur par l'option "-adress", sinon il se lance sur le localhost.

1.2 Traitement d'une commande

Afin de traiter les commandes reçues, on les transforme en une liste dont chaque élément est délimité par "/" (non précédé par un "\" afin de respecter le protocle), il a donc fallu redéfinir une fonction quasi-équivalente à Str.split mais qui respectait le protocole. Ainsi, commande et arguments sont aisément identifiés. Le traitement des commandes se reposera par la suite sur la liste ainsi obtenue.

1.3 Connexion

La phase de connexion (module Connexion) est séparée de la boucle principale de jeu afin d'identifier les conditions de connexion (joueur ou spectateur) et lancer le traitement

adéquat. Si le pseudo reçu est déjà utilisé (joueur présent, ou utilisateur enregistré), alors un nouveau pseudo est généré en rajoutant un entier à la fin. Au bout de 2 joueurs connectés, un timer est lancé, implémenté par un thread qui, une fois les 30 secondes écoulées, lance la partie si c'est nécessaire. Sinon la partie est lancée au bout de 4 joueurs connectés.

1.4 Structure de données

Lors de la connexion, la structure représentant le client est alors initialisée. Elle comprend :

- son nom
- la socket grâce à laquelle il communique avec le serveur
- la liste de ses bateaux
- la phase de jeu courante
- la position de son drone

Les joueurs sont stockés dans une liste (de taille maximale 4 donc), ce qui permet d'utiliser les nombreuses fonctions du module List.

Les spectateurs sont eux stockés dans une autre liste, et ne sont représentés que par leur socket.

La phase de jeu est représentée par un type somme qui nous permettra ainsi de vérifier si une commande est licite à un moment de la partie donné.

Un bateau est une suite de cases et possède un état. Une case est une position munie d'un état.

Le type état est représenté par un type somme (Alive ou Dead). Il peut paraître ici assez inutile, mais il a été implémenté initialement dans l'objectif de réaliser d'extensions qui n'ont finalement jamais vu le jour (réparer un bateau, lancer une épidémie, ...).

Enfin, la grille de jeu est simplement représentée par une matrice de noms (à partir desquels on pourra retrouver la structure du client).

1.5 Extensions

1.5.1 Discussion instantanée

Cette extension a été plutôt aisée à implémenter, dès que le serveur reçoit la commande "TALK" correctement formée, il renvoie à tout le monde (clients et spectateurs) le message passé en argument précédé de son émetteur. Le split de la commande permet de respecter le protocole.

1.5.2 Comptes utilisateurs

Cette extension repose sur un simple fichier texte où sont stockés logins et mots de passe. Le fichier ("logins.txt") est créé s'il n'existe pas.

1.5.3 Spectateurs

Pour cette extension, il a fallu ajouter un mécanisme qui stocke les commandes voulues et les envoie aux spectateurs déjà présents. Dès qu'un spectateur se connecte, il reçoit alors la liste des commandes stockées. Grâce au split des commandes, il a suffi de changer le nom de la commande ("PUTSHIP" en "PLAYERSHIP", ...) qui correspond au premier élément de la liste résultante.

1.5.4 Serveur statistiques

Cette extension lance juste un thread qui va créer une socket sur le port 2092 et attendre les requêtes "GET" de la part du client (navigateur, telnet, ..) puis va lire le fichier de statistiques "logins.txt", et générer une page html avec en-tête. Les données lus sont inscrites dans une balise *table* qui contient les informations des joueurs inscrits (ceux ayant effectués un register).

1.6 Architecture

L'architecture n'est pas forcément très rigoureuse car elle n'a pas été développé dans un esprit de réutilisabilité, mais en voici une brève description. Lors de la connexion d'une nouvelle socket, un thread exécutant "main_client" est lancé. Cette fonction lance la phase de connexion évoquée plus haut, et selon le résultat exécute "main_joueur" ou "main_spectator". La deuxième fonction se contente de réceptionner les commandes déjà effectuées et permet au spectateur de parler. La première effectue une boucle qui filtre à chaque tour la phase du joueur et la commande reçue afin de lancer le traitement adéquat.

Ci-suit une liste des modules :

- RegExp : stocke les expressions régulières, mais elles ont été beaucoup moins utilisées que pensé initialement
- Utils : toutes les fonctions utilitaires, ne se rapportant pas spécifiquement à un autre module
- Next : gère le correct enchaînement des "YOURTURN"
- Register : gère les comptes utilisateurs
- Stop : gère les déconnexions
- Connexion : cf plus haut
- Placement : gère la phase de placement des bateaux (et oui, quel nom adéquat !)
- End_of_game : gère la fin de partie (plus ou moins bien, des tests exhaustifs n'ont pas été effectués)
- Game : gère les phases d'actions

1.7 Remarques

En plus de l'option "-address" qui prend en argument l'adresse du serveur, est présente la possibilité d'afficher la trace par l'option "-debug". La plus grande difficulté a

été, à mon sens, de tester le code, telnet ayant ses limites (notamment pour finir une partie...) et les versions finales s'étant développées sous la pression des délais. De plus, les commandes envoyées via telnet n'étaient pas forcément exactement les mêmes que celles reçues lors des phases de tests avec les clients des mes collègues. Un bug m'a été signalé par mes collègues lors de la fin du timer en phase de connexion, mais je n'ai jamais pu le constater chez moi ou à l'ARI, même lors des phases de tests finales. Si cela se reproduit, une solution est de commenter le lancement du timer dans la fonction "treat_connexion" (affectation de la référence timer) et de décommenter la ligne "start_game ()". En espérant que cela ne soit pas nécessaire :)

Bon jeu !

Chapitre 2

Client Java - Vincent Botbol

Ce client a été réalisé en *Java* (version 7). L'interface graphique utilise la bibliothèque *Swing*. Les extensions implémentés sont : la discussion instantanée, les comptes utilisateurs et le mode spectateur. Les tests ont été réalisés sur le serveur OCaml fourni ainsi que sur un serveur factice java également présent dans l'archive.

Le programme démarre sur une fenêtre de connexion proposant à l'utilisateur d'entrer l'adresse du serveur, son pseudo, son mot de passe (pour les comptes utilisateurs), de choisir le mode spectateur, et de se connecter (protocole "LOGIN" si le mot de passe est fourni, "CONNECT" sinon) ou d'enregister son compte.

Après la réponse du serveur, l'interface graphique du jeu s'affiche. Celle-ci se décompose en trois parties :

- La grille de jeu
- Une zone de texte pour les informations émises par le programme et par le serveur
- La zone de discussion

Pour communiquer avec les autres joueurs (et spectateurs), il suffit d'entrer son texte puis de valider avec "entrée" ou bien de cliquer sur "Envoyer".

La grille de jeu permet les interactions de l'utilisateur et la zone de notification transmet les informations importantes.

Dès la réception du "SHIP", l'utilisateur place son bateau en déplacant la souris sur la grille de jeu (clique droit pour le tourner), et valide sa position avec clique gauche. Le client réalise une vérification interne sur le placement avant d'envoyer la commande "PUTSHIP" au serveur. Les bateaux sont représentés par des rectangles de couleur rouge, grise, verte ou jaune selon leur positions dans la liste de joueurs.

La disposition des bateaux terminée, le client attend le "YOURTURN" avant de redonner la main à l'utilisateur. Lorsque son tour est arrivé, le drone apparait ainsi qu'une zone d'action basée sur les mouvements accordés. L'utilisateur finit son tour lorsqu'il ne lui reste plus de mouvement ou bien qu'il passe son tour (clique droit). En cliquant sur sa position, le joueur active le laser. Si un bateau a été touché pendant ce tour, l'utilisateur peut le constater grâce à une croix verte apparaissant à la position du bateau touché. Si le coup est manqué, un rond magenta est disposé. Si l'un de ses bateaux est touché pendant le tour d'un des adversaires, le client est notifié d'une croix

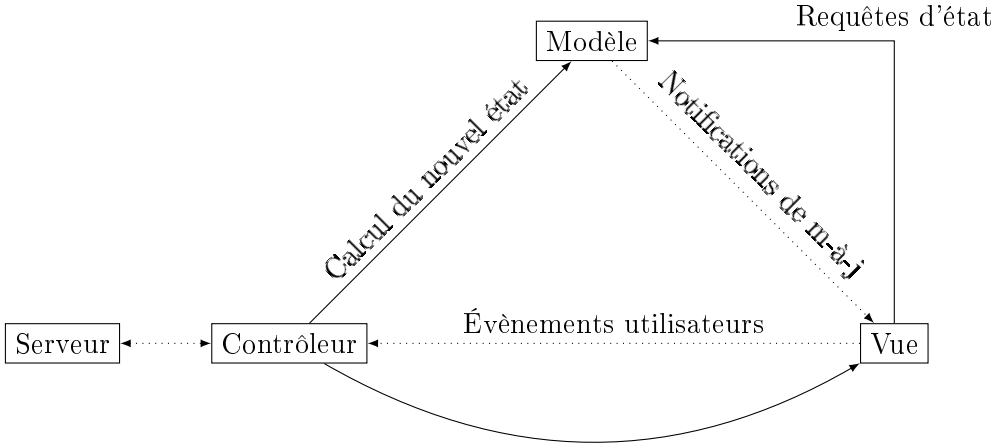
rouge à l'emplacement du bateau détruit.

En mode spectateur, l'utilisateur ne peut intéragir. Il peut uniquement communiquer par le biais de la discussion instantanée. Il est notifié des touches de bateaux par un code couleur propre au joueur ayant effectué l'action.

2.1 Implémentation

2.2 Modèlisation

L'architecture du client suit un patron de conception MVC strict. Ce schéma illustre la modélisation de l'application :



On distingue donc trois parties principales :

- Le modèle, gérant toute la partie logique de l'application : la grille, le calcul des déplacement, la validité des coups pour la vérification côté client, etc..
- La vue, s'occupant d'afficher les composants graphiques pour communiquer avec l'utilisateur. Cette dernière va s'enregistrer auprès du modèle qui va notifier la vue lorsqu'elle est censée se mettre à jour.
- Le contrôleur, connaissant le modèle et la vue, va permettre de connecter l'interface graphique (boutons, champs textes, ..) et ainsi de traiter les divers évènements que l'utilisateur est susceptible d'envoyer par le biais de l'interface. En réceptionnant ces évènements, le contrôleur établit les calculs à effectuer par le modèle. Le contrôleur s'occupe également de la communication avec le serveur, en traduisant en requêtes les actions de l'utilisateur transmises par la vue et en réceptionnant les réponses du serveur qui démarreront les calculs.

Pour expliciter cet modélisation, on peut détailler un exemple, celui de l'envoi d'un message instantané :

1. Le client saisi son message dans le champ de texte puis clique sur "Envoyer"
2. Le bouton "Envoyer" de la vue étant relié au contrôleur, celui-ci procède à la récupération du texte au travers de la vue. Puis envoie une requête de type "TALK/message/" au serveur

3. Le serveur répond en transmettant la commande “HEYLISTEN”. Le contrôleur traite cette nouvelle commande reçue en appelant le module discussion du modèle qui va ajouter ce nouveau message, prévenir la vue qu'il y a un changement de type “discussion instantanée” et qu'elle doit donc se mettre à jour.
4. La vue étant enregistrée sur le modèle, elle reçoit cet événement, récupère le nouveau message et enfin l'ajoute à sa zone de discussion.

Cette séquence de communication se repète pour la plupart des interactions du programme.

2.3 Concurrence

Le modèle de concurrence employé est préemptif puisqu'il utilise exclusivement l'API native de Java. L'interface graphique est exécutée à l'intérieur de l'“Event Dispatcher Thread” (EDT) pour garantir la fluidité de l'interface graphique même lors de calcul long réalisés par le modèle. Ces calculs sont eux-mêmes exécutés dans des threads[1] dédiés et extérieurs à l'EDT. Des “SwingWorker” ont pour cela été utilisés.

Le principal thread (explicite) de l'application reste la lecture continue des réponses du serveur incluant donc les différents calculs impliqués. D'autres threads secondaires sont également déployés. Notamment pour la mise-à-jour de l'interface graphique ou encore pour effectuer des effets visuels sur la grille de jeu (qui ne sont, certes, pas nombreux).

2.4 Communications Client-Serveur

Comme spécifié dans le sujet du projet, la connexion s'appuie sur un protocole TCP sur le port 2012. Les sockets utilisées sont celles de Java provenant du paquet “java.net” et les lectures/écritures sont réalisés par canaux tamponés. Les envois et récéptions sont textuelles. Les données sont transmises au moyen d'un “DataOutputStream” pour augmenter la portabilité des données envoyées. Les lectures sont quant à elles interprétées directement par un “BufferedReader” lisant les commandes reçues au moyen d'un “readLine()” (puisque les commandes sont préfixés d'un retour à ligne).

Le protocole de communication fourni a été respecté. Le traitement des commandes reçues s'effectue par expression régulière. L'utilisation du *look-behind* a permis de s'abstraire du cas particulier (tels que le “”). Il a donc suffit de séparer la chaine de caractère reçue par l'expression “(?<!\\)/” qui peut-être traduite par : “sépare les champs de la chaine par les slashs présents mais uniquement si celui-ci n'est pas précédé d'un anti-slash”.

Les slashs saisis par l'utilisateur dans un message instantané, par exemple, sont traités avant l'envoi (en les sécurisant par un anti-slash) et de même pour les anti-slashs. La réciproque est employée lors de la réception.

1. Fil d'exécution

Pour plus de sécurité, les commandes possible à l'envoi sont typés par énumération (ERequest). Les commandes reçues sont traités et coércées vers une deuxième énumération (EResponse) qui générera une exception si la commande reçue ne fait pas partie de l'énumération. Le tout est encapsulé dans un objet "Command" contenant la commande protocolaire et les arguments reçus ou à envoyer.

2.5 Difficultés

La principale difficulté de ce projet a été d'établir la structure principale de l'application. Une fois cette phase réalisée, l'enchainement de l'implémentation des différentes parties du jeu fut triviale.

Le temps accordé a également été une problématique. J'aurais souhaité améliorer l'aspect visuel et sonore de l'application, ajouter quelques extensions et peaufiner l'implémentation mais la date limite approchant, je n'ai pas eu cette occasion.

Chapitre 3

Client Python - Matthieu Dien

3.1 Choix Techniques

Pour ce client, le choix du langage d'implémentation a été Python car c'est un langage haut niveau, simple, avec une librairie standard très fournie et dont une majorité de librairie graphique implémente un binding pour lui. La librairie graphique choisie est wxPython, s'appuyant sur wxWidgets (écrite en C++), elle est libre, portable (disponible à l'ARI), complète (nombreux widgets) et simple d'utilisation. Ce choix Python/wxPython garantit la portabilité du client et facilite un peu sa conception.

3.2 Manuel d'utilisation

3.2.1 Connexion

La connexion se fait en entrant l'adresse du serveur ainsi que son pseudo. Dans le cas où vous choisiriez un mot de passe le client vous demandera si vous êtes déjà inscrit, si non il le fait pour vous (en envoyant "REGISTER/" à la place de "LOGIN/"). Dans le cas où vous seriez déjà inscrit ou que vous vous seriez trompé de mot de passe, le serveur vous refoulera et vous devrez réessayer.

3.2.2 Placement des bateaux

Le placement des bateaux se fait en cliquant sur les cases que vous voulez. Si vous vous trompez, le serveur vous demandera de réessayer.

3.2.3 Phase de jeu

Vous devez attendre votre tour pour vous déplacer. Vous ne pouvez vous déplacer que sur les cases surbrillées en vert et la position actuelle de votre drône est surbrillée en rouge. Vous pouvez continuer à vous déplacer après avoir activé le laser de votre drône s'il vous reste des points d'action. Vous pouvez à tout moment passer la fin de votre tour de jeu en appuyant sur la touche "s" de votre clavier.

3.2.4 Chat

Un chat est à votre disposition pour toute conversation ou tentative de triche avec vos adversaires.

Bon jeu.

3.3 Implémentation

Le client se présente sous la forme d'un fichier *main.py*, de deux dossiers *connection* et *client*, ainsi que de deux fichiers *mybuttons.py* et *mysocket.py*.

3.3.1 Socket

Dans *mysocket.py* on trouve la fonction *readline* qui permet la lecture sur socket. C'est une petite surcouche au socket fournie par la librairie standard python qui ne permettait pas cette lecture ligne à ligne et qui m'a valu pas mal de bug.

3.3.2 Bouton

Dans *mybuttons.py* on trouve une surcouche au "BitmapButton" qui permet la gestion des différents types de case que l'on peut rencontrer durant la partie en utilisant un système de "flag" binaire pour pouvoir ajouter facilement des caractéristiques aux cases. Par exemple, une case peut être avec un sous-marin, touché et rouge (car sous le drône).

3.3.3 Client

Le principal du code se trouve dans le dossier *client*, notamment dans le fichier *controler.py*. On y trouve une classe principale *Controler* qui initialise l'interface graphique (UI) et sert de gestionnaire d'événements pour procéder à tous les traitements graphiques, ainsi que trois autres classes : *WaitPlayers*, *PutShip* et *Action*. Chacune de ces classes hérite de *threading.Thread*, équivalent Java de *Thread*, ce qui permet de ne pas bloquer l'UI lors de l'attente d'événements tels que la réponse du serveur ou l'appui d'un bouton. Ces classes gèrent chacune une phase de jeu d'une partie de "Bataille Navale Royale". La structure globale de ces classes est une boucle qui lit sur le socket à chaque itération et attend, en fonction de la phase de jeu, une ou des commandes particulières du serveur.
Le code contenu dans *view.py* est le code l'UI, il se compose donc de 256 boutons sous forme de quadrillage (la grille de jeu) et d'une zone de chat surmontée par la liste des joueurs présents dans la partie.
Cette partie du code utilise les *threading.Event* fourni par Python, qui permettent une synchronisation entre thread, mais aussi les événements fournis par wxPython, qui permettent une communication entre les threads et l'UI par passage d'objets.

3.3.4 Connection

Le dossier *connection* contient le code de la première fenêtre que vous trouverez en lançant le client. le fichier *connection/view.py* contient l'interface, tandis que le fichier *connection/controler.py* contient la phase de communication avec le serveur et la gestion des événements graphiques.

3.4 Conclusion

L'architecture de mon client et loin d'être parfaite, elle est assez monolithique et donc peu modulable, mais elle a quand même eu un avantage : la gestion du recommencement de partie, le client n'a qu'à tuer la fenêtre de jeu et en relancer une nouvelle. Les extensions demandées et implémentées sont la gestion du "REGISTER/LOGIN/ACCESSDENIED" et le chat.
Mes principales difficultés ont été la connaissance du langage et de la programmation d'interface graphique. J'avais choisi le langage Python malgré le fait que je ne le connaissais que très peu car il me semblait simple et rapide de coder avec. Ce sentiment s'est confirmé. Pour la conception d'interface graphique, cela a été un peu plus délicat car quelques bugs résident encore dans wxPython, mais globalement cela s'est bien passé grâce aux forums utilisateurs.

3.5 Remerciements

Je remercie tous les anonymes (ou pas) qui ont répondu à des questions liées à Python (surtout ses sockets) et à wxPython.