Devoir de programmation PC2R : iSketch

Clara Muller & Théo Lebourg

1 Côté serveur

Pour développer la partie serveur de iSketch, nous avons opté pour le langage OCaml. Nous allons dans un premier temps expliquer quels ont été nos choix pour organiser notre serveur avant de s'attarder un peu plus en détail sur le code.

1.1 Lancement du serveur et de la partie

Lorsque l'exécutable est lancé avec les paramètres qui vont bien, la première étape du programme est la création du serveur (pour qu'il soit prêt à recevoir les connexions des futurs clients) ainsi que l'initialisation de certaines variables qui seront utilisées une fois le jeu lancé.

La création de notre objet serveur reprend les étapes classiques vues en cours : on crée une **une prise** (en utilisant le domaine Internet, le flot d'octets et le protocole de communication par défaut) et on récupère **une adresse dans le domaine internet** (à l'aide de l'adresse Internet de la machine locale et d'un numéro de port défini par l'utilisateur) afin de pouvoir associer les deux, puis on rend la prise capable d'accepter les connexions et enfin, on met en place une boucle infinie qui accepte les demandes de connexions et qui va créer un thread joueur à chaque nouvelle demande (nous verrons plus tard comment nous gérons le nombre limite de joueurs par partie).

Notons qu'avant de commencer à accepter les demandes de connexion, nous lisons le fichier dictionnaire et nous stockons dans une liste chacun des mots présent dans le fichier. Nous avons opté pour cette solution pour éviter de devoir ouvrir et fermer le dictionnaire à chaque tour de round pour piocher un mot.

En parallèle des demandes de connexions des joueurs, on lance un thread qui va commencer par attendre que tous les joueurs soient connectés. Une fois cette condition remplie, le premier round s'exécute : un mot est choisi (et retiré de la liste pour éviter de le tirer plus tard) ainsi qu'un dessinateur et ces informations sont envoyées à tous les joueurs via la commande `NEW_ROUND`. Ensuite, le serveur attend cette fois-ci qu'un mot soit trouvé et une fois cette condition remplie, il envoie à tous les joueurs les commandes `END_ROUND` et `SCORE_ROUND` qui donnent respectivement le nom du vainqueur et le mot qu'il fallait trouver et le score de tous les joueurs. Enfin, un nouveau round peut commencer en suivant le même schéma qui vient d'être décrit.

1.2 Connexion d'un joueur

Plusieurs possibilités sont offertes au joueur pour accéder au jeu :

– Avec la commande `CONNECT/user/` : le serveur vérifie alors que le nom *user* n'a pas déjà été choisi (en vérifiant la base de données des comptes utilisateurs ainsi que les joueurs déjà connectés à la partie en cours) et crée un nouveau nom si ce n'est pas le cas (en concaténant un nombre à la fin de *user*)

– Avec la commande `REGISTER/user/password/` : le serveur vérifie alors que le nom *user* n'a pas déjà été choisi et l'ajoute à la base de données si c'est le cas
– Avec la commande `LOGIN/user/password/` : le serveur vérifie alors que les noms *user* et *password* sont corrects et s'ils le sont, le serveur vérifie également que le joueur n'est pas déjà en train de jouer

Enfin, notons qu'une commande `ACCESSDENIED/` est envoyée ou bien si le nombre maximum de joueurs pour une partie est atteint ou bien si les noms et mots de passe ne correspondent pas ou bien si un joueur tente de se connecter alors qu'il est déjà connecté.

1.2.1 Quand un mot est trouvé par un joueur

Voici le déroulement général des étapes effectuées par le serveur lorsqu'un joueur a trouvé le mot que le dessinateur était en train de dessiner. Notons que puisque les joueurs évoluent chacun sur un thread, nous avons protégé cette série d'opérations par un **mutex**.

1. Le serveur se charge d'envoyer la commande `WORD_FOUND/joueur/` à tous les joueurs de la partie
2. Le serveur attribue le nombre de points qui va bien au joueur qui a trouvé le mot
3. Un timeout est lancé si et seulement si le joueur en question est le premier à trouver le mot et il reste des joueurs connectés (sinon, s'il est le seul, alors un nouveau round peut commencer si la partie n'est pas terminée)

1.3 Extensions

1.3.1 Discussion instantanée

La mise en place de la discussion instantanée côté serveur n'a pas été trop compliquée étant donné qu'il suffit de récupérer la chaîne de caractère contenue dans la commande `TALK` pour la renvoyer à tous les joueurs via la commande `LISTEN`. Par ailleurs, comme suggéré dans l'énoncé, nous avons utilisé la commande `BROADCAST` pour annoncer à tous les joueurs qu'un acte de triche avait été signalé par l'un d'entre eux.

1.3.2 Comptes utilisateurs

Côté serveur, nous avons opté pour le stockage des noms et mots de passe des joueurs dans un simple fichier texte. Cependant, avant d'être stocké sur ce fichier, le mot de passe est salé (dynamiquement pour éviter les attaques par tables arc-en-ciel) puis hashé avec la fonction de hashage MD5 (cette fonction de hachage n'est certes plus fiable depuis longtemps mais nous n'avons pas trouvé dans la librairie standard d'OCaml d'autres fonctions plus performantes telle SHA-256). Ainsi, à chaque inscription, on ajoute une ligne contenant le nom du joueur, le mot de passé hashé et salé et le sel (ainsi que deux 0 symbolisant respectivement le nombre de parties gagnées et perdues et qui seront incrémentés au fur et à mesure des parties et utilisés par le serveur HTTP de statistiques).

Par ailleurs, notons que pour l'instant, les mots de passe circulent en clair lorsqu'ils sont envoyés depuis le client.

1.3.3 Serveur HTTP de statistiques

Afin de rendre les statistiques de iSketch (nombre de parties gagnées et perdues pour les joueurs enregistrés dans la base de données) disponibles via un navigateur internet, nous avons repris la base de notre serveur acceptant les connexions des joueurs mais en fixant cette fois-ci le port 2092 comme énoncé dans le sujet. Il nous a ensuite fallu trouver le moyen de répondre aux requêtes `GET` du navigateur envoyées lorsque le joueur entre l'URL *adresse_du_serveur :2092*. Pour cela, lorsque nous recevons une requête `GET` (de la forme *GET / HTTP/protocole*) nous commençons par récupérer le protocole pour construire notre requête qui sera construite de la manière suivante : une première ligne de la forme *HTTP/protocole 200 OK*, suivie de quelques headers, une ligne vide et enfin notre page HTML contenant les informations des joueurs.

Par ailleurs, nous avons muni la page HTML d'une balise qui permet le rafraîchissement de la page toutes les minutes pour ainsi actualiser les statistiques.

2 Côté client

2.1 L'interface graphique

Le langage JAVA a semblé le plus adapté pour gérer l'interface graphique avec l'utilisateur. En effet, les packages swing, et awt permettent une grande variétés de composant simple à appréhender. La fenetre principale est donc divisé en 4 zones : - en haut à droite, la zone de dessin, active ou non selon le rôle du joueur durant le round - en bas à droite, une zone de chat qui fonctionne en permanence pour tous les joueurs - à gauche, la majeur partie de l'écran correspond au fenêtre de dialogues entre le client et le serveur : on peut y lire la liste des joueurs et les différentes actions de la partie en cours - en bas à gauche, un champ pour envoyer des proposition de mots au serveur. Ce champ n'est actif que lorsque le joueur n'est pas le dessinateur du round.

L'interface graphique transmet les informations entre les différentes hiérarchie de classes jusqu'au classes principales qui font envoyer et recevoir les information du serveur.

2.2 Les classes principales

La principale classe du Client n'est pas *Client.java* comme l'on pourrait le supposer. Néanmoins elle contient des éléments essentiels comme : la fonction main du programme, la lecture des arguments du programme et l'initialisation de la connexion à l'aide de la socket avec la bonne adresse et le bon port. Une fois la socket crée, elle peut lancer un canal de lecture et un canal d'écriture tous les deux lié au serveur qui vont être donné à la classe qui va gérer tous les messages : *Messenger.java* Tous les envois et réception de message transitent par cette classe où ils sont traité et envoyé au serveur si c'est le destinataire, ou vers la zone de l'interface graphique appropriée. Pour cela la classe *MainWindow.java* qui contient tous les composant graphiques traitent également transmet quasiment tous les messages entre l'interface et la classe *Messenger.java*. Elle fait remonter toutes les actions de l'utilisateur vers la classe *Messenger.java* et redistribue toutes les informations du serveur vers les composants graphiques concernés. Mais cette la classe MainWindow est crée uniquement si le joueur à réussi à se connecter correctement. Pour cela, *Messenger.java* propose un choix au joueur sur la manière de se connecter (anonymement, en s'enregistrant ou en utilisant son identifiant et son mot de passe), puis écrit sur le canal d'écriture du serveur la commande connexion correspondante. Elle attend ensuite sur le canal de lecture une réponse. Si cette réponse commence bien par `WELCOME` ce qui signifie que, la connexion a réussi, l'interface graphique se lance ainsi que le système d'écoute et d'envoi

de messages. Si n'importe quelle autre réponse en récupérée, nous considérons que la connexion a échouée, aucun autre objet n'est créé et la classe *Client.java* referme la socket.

2.3 Le système d'écoute et de d'envoi de message au serveur

Le client ayant moins de demandes en parallèle à traiter que le serveur, nous avons pus traiter la principale difficulté avec deux thread lancé tous les deux en même temps dont les rôle respectifs sont d'écouter toutes les commandes du serveur pour l'un et d'envoyer toute les commandes au serveur pour l'autre. Pour que le thread d'envoi de message ne soit actif que lorsqu'un message doit être envoyé, on le le fait s'endormir sur une variable déclarée dans *Messenger.java* qui lorsqu'elle est modifiée, notifie le thread concerné à l'aide de la méthode notify(), qui l'envoie alors au serveur avant de se rendormir. Le thread de réception des messages, lui, lit en permanence sur le canal de lecture du serveur et renvoie l'interprétation de la commande à la classe *Messenger.java* lorsqu'il en reçoit une par l'intermédiaire d'une variable dans lequel il a le droit d'écrire. Ces deux variables sont de type *Message.java* et permettent des actions simple comme écrire un message ou récupérer le message présent. Pour ne pas que les commandes se chevauchent et que des informations soient perdues, toutes les actions des thread, d'écriture ou de lecture dans les variable sont effectuées dans des blocs synchronized ce qui empêche tout écrasement d'information avant qu'elle ne soit traitée.

2.4 Le système de dessin

Lorsque le joueur est le dessinateur du round, la zone de dessin de l'interface devient active. Les lignes simples ont été implémentées avec plusieurs tailles et plusieurs couleur différentes. Pour tracer un trait, l'utilisateur clique sur la zone de dessin d'où il souhaite faire commencer l'extrémité de son trait et relâche la souris à l'endroit souhaité de l'autre extrémité du trait. Grâce à l'interface *MouseListener.java* implémentée par la classe *BoardPanel.java*, nous pouvons récupérer les deux point distinct pour tracer un trait entre les deux, un point étant distingué par sa position, sa couleur et sa taille. Pour pouvoir dessiner plusieurs traits, chaque point est stocké dans une liste qui lorsque nous faisons appel à la méthode repaint() retrace les anciens traits ainsi que les nouveaux qui auraient pu être ajoutés. Pour détecter les changements de couleurs ou de taille sélectionnés par l'utilisateur, on a utilisé des classe implémentant l'interface *ActionListener.java*. Ainsi lorsque le joueur trace un trait, effectue un changement de taille ou de couleur, l'action en renvoyée jusqu'à la classe *Messenger.java* qui va transmettre l'information au serveur à l'aide de la commande correspondante.

Si le joueur n'est pas dessinateur, cette zone n'est pas active mais, il reçoit toutes les informations du serveur qui lui permettent de redessiner sur sa propre zone tous les traits tracé par le dessinateur.