ENSEIRB-MATMECA

Filière informatique : Deuxième année

Projet de Réseaux
Rapport final

Lucien CASTERES
Adrien MAURICE
Cédric NIS
Toby ROPER
Aurore VALADE

Encadrant : M. Toufik AHMED

Table des matières

1 Rappel du projet

Le but de ce projet est d'implémenter un aquarium centralisé de poissons, en suivant le modèle MVC. Le modèle contient les informations relatives à un poisson (vitesse, position...), la vue correspond à l'affichage en temps réel de ces différents poissons, et le contrôleur fait la liaison entre les deux en communiquant à la vue les positions des poissons.

2 Répartition et organisation du travail

Pour représenter ce système, nous avons découpé le projet en deux principales entités :

— Un Backend en C, représentant le contrôleur. Il est composé d'une invite de commande, d'un parseur avec interpréteur de commande, ainsi que d'un serveur pouvant communiquer avec plusieurs clients.

— Un client en Java, représentant la vue. Le client intègre un parseur de commande, un système d'affichage et un système de communication via socket avec le contrôleur.

Puis le projet a été découpé en plusieurs grandes tâches indépendantes, selon les entités établies précédemment, réparties au sein de l'équipe de la manière suivante :

— Lucien CASTERES s'est occupé du parseur et du prompt du côté client

— Toby ROPER s'est occupé de l'affichage des poissons et du client TCP

— Cédric NIS et Adrien MAURICE ont pris en charge l'implémentation d'un parseur qui va lire les commandes envoyées par le client, les exécuter côté serveur et renvoyer les réponses adéquates au client

— Aurore VALADE s'est chargée de l'implémentation du serveur

Concernant l'organisation, le travail a été réalisé sans planification préalable : nous avons implémenté les différentes fonctionnalités et commandes au fur et à mesure, en notant au fil de la réalisation celles qui devenaient fonctionnelles.

Une fois ces grandes tâches réalisées, nous nous sommes réunis pour faire en sorte que toutes ces parties s'imbriquent bien et que le tout fonctionne correctement.

3 Client JAVA

3.1 Structures

Les différentes fonctionnalités du client sont réparties parmi des packages dont la liste actuelle est fournie ci-dessous :

— **aqua** : éléments relatifs aux poissons et à leur affichage

— **fishnet** : éléments relatifs à la communication avec le serveur

— **parser** : éléments relatifs à l'analyse des messages de l'utilisateur et du serveur

— **prompt** : éléments relatifs à l'entrée utilisateur

— **starter** : racine des instances du programme

La figure 1 est un diagramme UML du client.

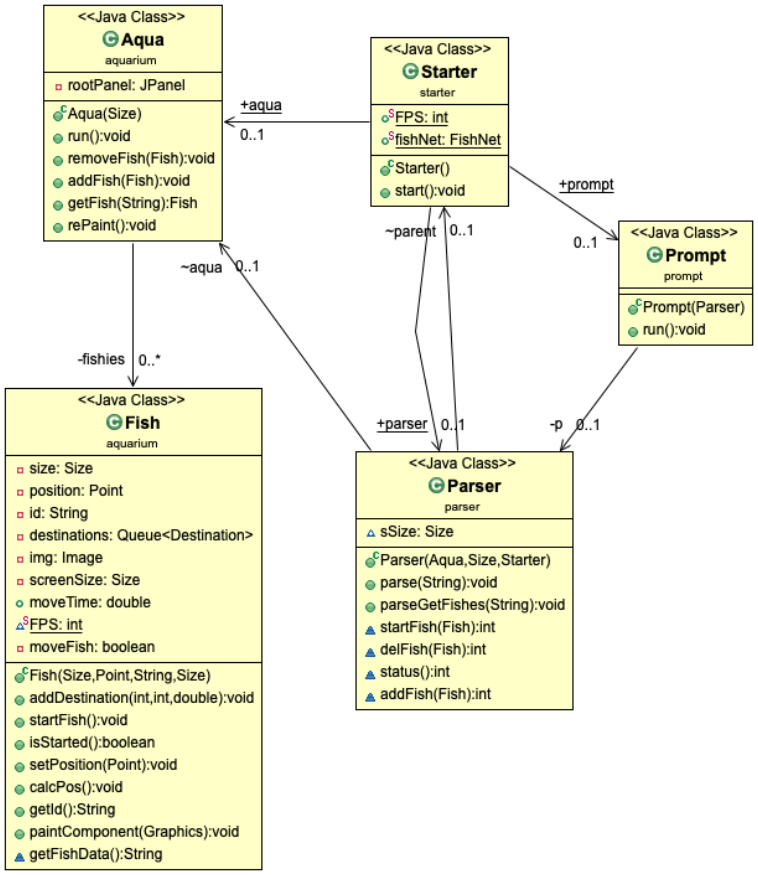

FIGURE 1 – Diagramme UML des objets côté client

3.2 Connexion au serveur

Pour se connecter au serveur en TCP et recevoir des messages, on utilise les sockets en Java. La création du socket ainsi que les Input et Output Streams sont présentés dans le listing 1.

Listing 1 – Connection socket

```
try (Socket socket = new Socket(hostname, port)) { //create socket

        InputStream input = socket.getInputStream();
        InputStreamReader reader = new InputStreamReader(input);
        out = new PrintWriter(socket.getOutputStream(), true); //for sending to server
```

```
int character;
StringBuilder data = new StringBuilder();

while(true){
  while ((character = reader.read()) != '\0') {
      data.append((char) character);
  }
  System.out.println(data.toString());
  parent.parser.parse(data.toString());
  data.setLength(0);
}
```

Pour envoyer des messages au serveur, on utilise le PrintWriter du listing 1.

Notre classe Fishnet permet d'utiliser ces fonctions pour communiquer avec le serveur. On initialise une instance de Fishnet avec un hostname, un port et un Parent (qui sert à accéder aux autres éléments du projet). Celle-ci implémente l'interface Runnable et peut ainsi être exécutée dans un thread indépendant du fonctionnement des autres éléments du logiciel. Lorsqu'on reçoit un message du client, on le passe au parseur qui est responsable de déclencher l'action correspondante.

Dès sa connexion, le client envoie au serveur une commande afin de recevoir les poissons en continu.

3.3 Prompt côté client

Cette classe implémente le prompt de l'utilisateur. Elle dispose d'un parseur afin d'exécuter les commandes. La fonction `run()` attend qu'une ligne de commande soit entrée et fait appel au parser qui va traiter la commande.

3.4 Parseur client

Du côté client, le parseur doit reconnaître les commandes suivantes :

— status : retourne l'état des poissons connectés

— addfish : permet d'ajouter un poisson à l'aquarium

— delfish : permet de supprimer un poisson de l'aquarium

— startfish : permet de mettre un poisson en mouvement

Le parseur correspond à la classe Java suivante :

```
public class Parser {

    Size sSize;
    Starter parent;

    public Parser(Size screenSize, Starter parent)
    public void parse(String s);
    public void parseGetFishes(String msg);

    int startFish(Fish fish);
    int delFish(Fish fish);
    int status();
```

```
    int addFish(Fish fish);

}
```

La fonction `parse(String s)` prend en entrée une commande. Le séparateur utilisé étant le caractère espace, elle découpe la commande selon les espaces. Elle vérifie ensuite que le nom de la commande est valide. Si ce n'est pas le cas, on affiche `"Commande non reconnue"`.Dans le cas d'une commande valide, on découpe le reste de la commande selon la syntaxe établie dans le sujet, et on appelle une fonction qui réalise la commande reconnue avec les arguments de la commande. Une exception est levée si les arguments ne sont pas valides. Cette fonction s'occupe également de gérer les messages envoyés par le server et de les traiter.

L'autre fonction importante du parser est la fonction `parseFishList(String msg)`. Cette fonction prend en entrée une liste de poissons, envoyée par le serveur. Pour tous les poissons de cette liste, on regarde s'ils existent. Si c'est le cas, on leur ajoute une destination. Sinon, on crée le poisson avec les coordonnées passées. Seul les poissons présent dans la liste parsée sont affichés dans la vue qui reçoit le message.

3.5 Affichage des poissons

Pour afficher l'aquarium, on utilise les bibliothèques Swing de Java. On affiche une fenêtre grâce à un objet JFrame. Aqua hérite de JFrame et peut donc être affiché. Les poissons héritent de JPanel et peuvent ainsi être ajoutés à l'Aqua. Aqua implémente également Runnable et peut être exécuté dans un thread. Ainsi, un Aqua calcule et affiche périodiquement le mouvement de ses poissons. Une boucle infinie dans la fonction $run()$ de Aqua avec un appel à $sleep(x)$ actualise et affiche ses poissons toutes les x millisecondes.

Lorsque la fonction paintComponent(Graphics g) d'un Fish est appelée par l'Aqua, celle-ci recalcule ses coordonnées et s'affiche.

4 Contrôleur

4.1 Structures

Pour implémenter les différents objets de ce projet, nous avons créé les structures suivantes :

— fish, qui représente un poisson. Elle contient son nom, sa position, sa taille, un pointeur de fonction définissant son mouvement, un booléen indiquant s'il est en mouvement, et une liste contenant ses futures positions (à chaque fois que l'on doit mettre à jour la position actuelle du poisson, on prend la première position du la liste).

— view, qui représente une vue. Elle contient son identifiant, sa taille, sa position ainsi que l'identifiant du socket associé.

— aquarium, qui représente un aquarium. Elle contient son nom, sa taille, une liste des vues utilisées, une liste des vues disponibles et une liste des poissons qui se trouvent dans l'aquarium.

Nous avons également implémenté des structures de liste doublement chaînées avec un pointeur en tête et en queue de liste. De plus, certaines, notamment les listes de vues, trient leurs éléments par identifiant croissant. Nous disposons également d'un tableau statique contenant les noms des modèles de mobilité supportés.

4.2 Parseur

Dans un premier temps, il fallait s'assurer que toutes les commandes à lire étaient syntaxiquement correctes. Pour cela, il fallait énumérer toutes les commandes possibles, en prenant en compte le fait que certaines permettent plusieurs syntaxes différentes, puis vérifier si la commande lue correspond bien à l'un de ces différents cas. Cela a nécessité la création d'une fonction permettant de séparer une commande en différents mots, selon un séparateur passé en paramètre (en règle générale, l'espace est utilisé, mais dans certains cas, d'autres séparateurs peuvent également être définis), ainsi que l'utilisation de la structure de liste définie précédemment afin de stocker ces différents mots. Ensuite, la liste est parcourue, puis, en fonction des mots rencontrés (correspondant au nom de la commande ainsi qu'à ses arguments) et de la taille de la liste (correspondant au nombre d'arguments), le parseur peut déterminer si la commande est valide ou non.

Nous avons également écrit un parseur permettant de lire, depuis un fichier de configuration, les informations utiles afin de configurer le contrôleur. Pour cela, nous utilisons la fonction **fgets** pour lire chaque ligne du fichier une par une, en ne prenant pas en compte les lignes inutiles (commentaires, lignes vides...). Une fois une ligne sélectionnée, nous faisons appel à la fonction de séparation utilisée précédemment, ce qui nous permet de récupérer le dernier mot de la ligne correspondant au nombre à récupérer. Ces informations seront ensuite utilisées pour configurer le contrôleur.

Nous avons ensuite écrit un ensemble de fonctions, chacune correspondant à une commande pouvant être envoyée par le client. Chaque fonction met à jour l'aquarium et les structures qu'il contient en fonction de la commande correspondante, en utilisant le parseur précédemment implémenté pour récupérer les éventuels paramètres. Par exemple, la fonction associée à la commande addFish va :

- récupérer le nom du poisson à ajouter et parcourir la liste des poissons existants pour vérifier que ce nom n'existe pas déjà
- récupérer la position initiale et la taille du poisson
- vérifier que le modèle de mobilité voulu existe bien
- créer une nouvelle instance de fish et l'initialiser avec les valeurs précédemment récupérées
- ajouter le fish dans la liste des poissons de l'aquarium

Enfin, dans la fonction vérifiant la validité de la syntaxe des commandes (**isCmdValid**), nous avons ajouté, pour chaque commande, un appel à la fonction associée dans le cas où la syntaxe est correcte. C'est la fonction **isCmdValid** qui va par la suite être appelée par le contrôleur à chaque fois que celui-ci va recevoir une commande, ce qui va lui permettre de vérifier la syntaxe de celle-ci, puis d'exécuter la fonction correspondante.

4.3 Serveur

Le serveur est décomposé en deux entités : la gestion des sockets et des communications, et l'invite de commande, qui sont représentés respectivement par un serveur et un client administrateur.

4.3.1 Gestion des sockets et communication

La gestion des sockets et des communications se fait via un élément central, que nous allons dorénavant appeler le serveur. Il permet d'accepter et d'écouter un certain nombre de clients, tout en réalisant des événements à des intervalles réguliers.

Pour ce faire, nous récupérons d'abord les paramètres du serveur en parsant le fichier de configuration. Puis, nous mettons en place une liste de structures **Client**, contenant :

— le descripteur de fichier de leur socket

— le temps écoulé depuis leur dernier message

— un booléen indiquant si le client s'est associé à une vue

— un pointeur vers le buffer de communication

— un booléen indiquant si le client a demandé la liste continue des poissons

— un verrou, de façon à sécuriser nos clients.

Le socket principal est alors ouvert, pour communiquer via adresse IPV4. Il écoute le port configuré précédemment, sur n'importe quelle IP de la machine. L'option `REUSE_ADDR` du socket est activée, de façon à permettre à plusieurs clients de se connecter au même port.

Une fois le socket principal mis en place, deux threads sont créés : un qui assurera les communications, et l'autre qui supervisera les événements temporels.

Communication

Tout d'abord, le thread est configuré pour ignorer les signaux **SIGALRM** utilisés par l'autre thread pour maintenir les évènements temporels. Ensuite, les variables nécessaires au fonctionnement interne sont mises en place, et le serveur débute sa gestion des événements. Nous disposons pour ce faire d'une liste des descripteurs de fichiers de tous les sockets actuellement utilisés.

Cette liste est initialisée avec tous les sockets sur lesquels le serveur est actuellement en écoute : le socket principal acceptant les connexions, le socket admin correspondant à la console, et tous les sockets clients actuellement utilisés.

Quand la liste est configurée, le serveur réalise une attente passive sur cette liste, en utilisant la commande `select`. Ainsi, le serveur ne questionne les sockets que si un événement a eu lieu sur le réseau.

À l'expiration de l'attente, le serveur va chercher la cause de son réveil :

— Si le socket principal l'a réveillé, il a reçu une requête de connexion : il l'accepte alors, récupère le descripteur de fichier associé à cette connexion, et l'ajoute à la liste des sockets clients.

— Si le socket admin l'a réveillé, il a reçu une commande de l'invite de commande, ou alors l'invite de commande s'est déconnectée, ce qui met fin au serveur. L'invite de commande est parsée, et le retour est retransmis à l'invite de commande.

— Si c'est un socket client qui l'a réveillé, il va d'abord vérifier si le client vient de se déconnecter, pour libérer sa structure. Le cas échéant, il a reçu une commande, et vérifie d'abord si le client est valide, i.e. est associé à une vue, ou vient de faire une demande d'association. Si c'est le cas, il transmet la commande au parseur, et transmet la réponse au client. À noter que le client analyse les deux pour des comportements spéciaux, comme la demande continue de poissons, la déconnexion, la validité d'un client... qui ne peuvent être gérés par le seul interpréteur de commandes.

Gestion temporelle

Le serveur dispose en outre d'un thread lié aux événements temporels. Celui-ci repose sur l'émission du signal **SIGALRM** toutes les secondes pour être le plus régulier possible.

À la réception de ce signal, le serveur va, pour chaque client connecté, incrémenter le compteur depuis le dernier message reçu, et éjecter le client si celui-ci n'a pas émis de message depuis le temps indiqué dans le fichier de configuration. L'éjection se fait en simulant la commande "log out" en provenance du client.

En outre, si le client a fait une demande de poissons en continu, le serveur va régulièrement simuler la commande "getFishes", et transmettre au client le résultat.

4.3.2 L'invite de commande

L'invite de commande est un client spécial, désigné précédemment et dans le code par client Admin. Il s'agit d'un client ayant transmis au serveur un buffer spécial, qui ne peut pas être écrit par un client classique. Il permet de récupérer les messages tapés, de les transmettre au serveur, et d'afficher sa réponse.

Le client admin ne peut pas être éjecté du serveur, et ne correspond pas à une vue. C'est lui qui permet la gestion du serveur, avec les commandes présentées dans le sujet. L'interprétation des commandes se fait comme vu précédemment.

4.3.3 Fonctionnement

Le programme principal, **main**, commence par faire un fork. Le processus fils va alors exécuter le serveur, et le processus père va s'y connecter, puis se comporter comme l'invite de commande.

5 Conclusion

Côté serveur, toutes les fonctionnalités requises fonctionnent correctement : en simulant des commandes clients via l'invite de commande, le serveur répond bien et met à jour de manière adéquate les différentes structures utilisées. Côté client, l'affichage est conforme à ce qui est renvoyé par le serveur, mais certains problèmes subsistent (le déplacement des poissons, notamment, n'est pas fluide).