Projet de réseaux

Semestre 8

Simulation d'un aquarium de poissons

Filière Informatique - ENSEIRB-MATMECA

Réalisé par :

Chaguer Reda
Zerrad Zaid
Bahhou Houssam
Ziani Asmae

Introduction

Ce projet s'inscrit dans le cadre des projets du semestre 8 et vient compléter le module sur les applications TCP/IP. Il consiste en la création un programme d'affichage de poissons à travers un nombre d'afficheurs connectés à l'aide d'un contrôleur (figure 1), et ce dans le but de réaliser une simulation naturelle et fluide d'un aquarium de poissons. Ce document présente notre réalisation du projet, son architecture, sa gestion, et les justifications des choix faits pour notre implémentation.

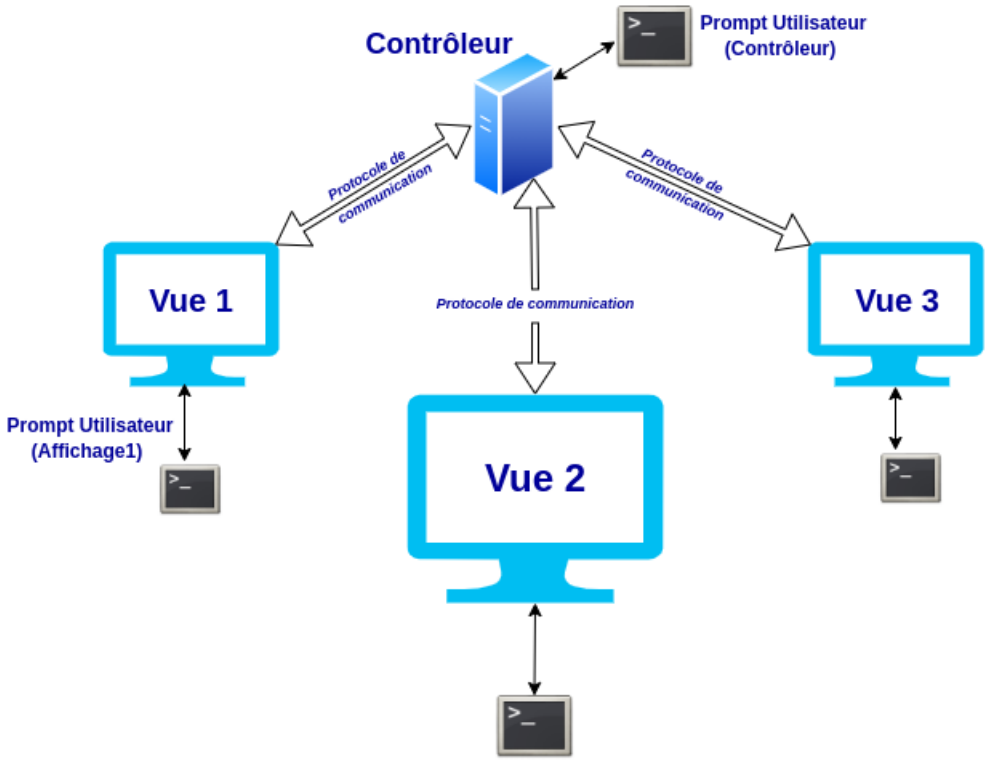

FIGURE 1 – Intéractions entre l'utilisateur, le contrôleur, et les affichages

Gestion de projet en équipe

Nous avons choisi pour la gestion du projet la **méthode agile** avec des sprints qui durent une semaine. Nous avons profité des séances du projet pour discuter de l'avancement du projet, des problèmes survenus durant les sprints, ainsi que des modifications qui doivent être apportés à ce qui était initialement prévu (si de nouveaux besoins se manifestent).
Nous avons aussi utilisé l'outil **Trello**, où toutes les tâches à faire ont été listées, et découpées en sous-tâches si cela se jugeait nécessaire. Cet outil permet de suivre en temps réel l'avancement du projet, ainsi que d'assigner des tâches à des membres de l'équipe. Il permet donc d'éviter le double travail en exposant clairement les tâches terminées, celles en cours, et celles en attente.

1 Architecture

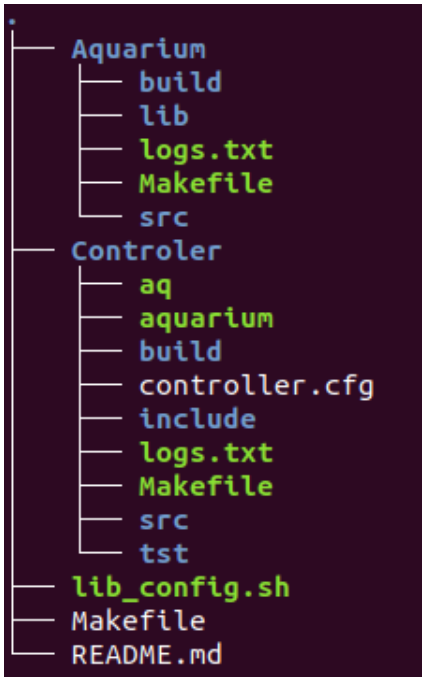

FIGURE 2 – Architecture du projet

Comme illustré par la figure 2, le projet est organisé de la manière suivante :

- **Controler/ :** contenant tous les fichiers responsables de la gestion centralisée de l'aquarium.
- **Aquarium/ :** contenant tous les fichiers responsables de la connexion avec le contrôleur et les programmes d'affichage.
- **lib_config.sh :** Un script pour configurer la librairie à utiliser selon le système d'exploitation. Par défaut, le répertoire **Aquarium/lib/** contient les fichiers de la librarie javafx desitinés pour **Windows**. Si le programme est lancé sous **Linux**, il faut appeler **make lib_config**.
- **Makefile :**
  - **make controler_install :** Permet de compiler les fichiers **C** du Contrôleur.
  - **make client_install :** Permet de compiler les fichiers **Java** du Client.
  - **make controler_run :** Permet de d'exécuter le Contrôleur avec le prompt.
  - **make client_run(`i`) :** Permet d'exécuter le Client avec la console et la visualApp selon le fichier de configuration `i`.
- **README.md :** inclut des règles de compilation et d'exécution du projet, ainsi que quelques détails d'utilisation.

2 Contrôleur

Le contrôleur joue le rôle d'un serveur connecté à tous les afficheurs. Il permet de centraliser la gestion de l'aquarium, en ayant toutes les informations sur l'état de l'aquarium, et donc de synchroniser l'affichage entre toutes les vues, en communiquant en temps réel l'avancement des états des poissons. La réalisation de cette partie a nécessité l'implémentation de plusieurs "modules" de traitement interne au niveau du contrôleur que nous détailleront ci-après, ainsi que des "modules" utilisés principalement pour le protocole de communication que nous aborderons dans la section 4.2.

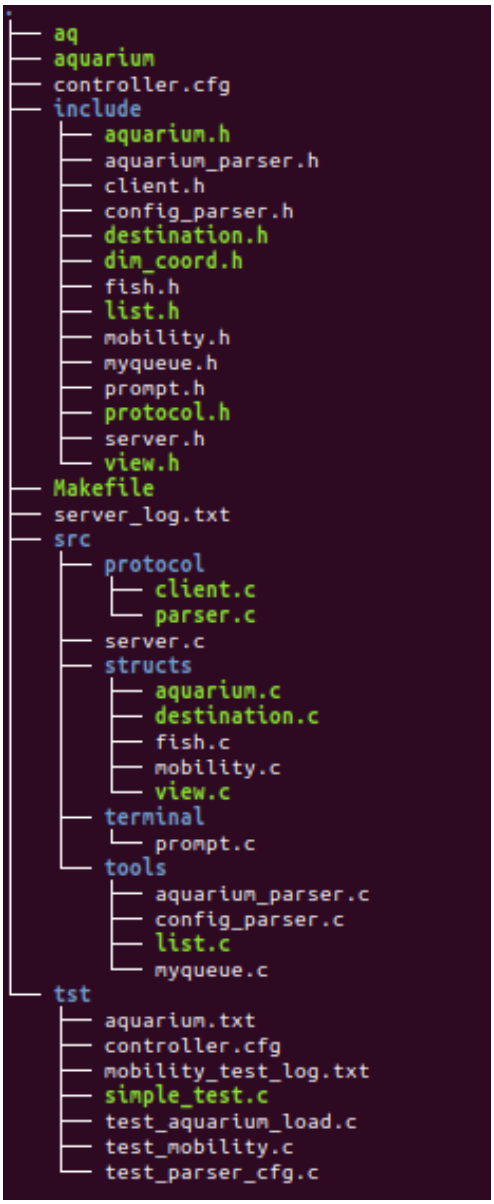

FIGURE 3 – Architecture du répertoire `Controler/`

aquarium.h

Le module ***aquarium*** contient la structure qui représente un aquarium, et les fonctions nécessaires pour sa gestion. Le choix de découpage des fonctions s'adapte au protocole de communication avec les afficheurs, ainsi qu'aux commandes du prompt utilisateur. En effet :

- **aquarium__initialize :** permet de créer un nouvel aquarium.
- **aquarium__load :** permet de charger un aquarium à partir d'un fichier, et

est appelée lors de la commande **load aquarium**.
Par défaut, l'aquarium est chargé à partir du fichier `"aquarium.txt"` contenant les dimensions de l'aquarium et des vues qui le composent.

- **aquarium__save :** permet d'exporter les données de l'aquarium dans un fichier. et est appelée lors de la commande **save aquarium**.
- **aquarium__add_view / aquarium__delete_view :** permettent respectivement d'ajouter et de supprimer une vue de l'aquarium.
- **aquarium__add_fish :** permet d'ajouter un poisson à l'aquarium. Cette fonction implémente la commande **addFish** du protocole de communication, côté aquarium.

fish.h

Le module ***fish*** contient la structure qui représente un poisson dans l'aquarium (du côté serveur), une énumération de tous les types de mobilités possibles, et les fonctions nécessaires pour la gestion de la structure. Les fonctions les plus importantes sont décrites ci-dessous :

- **fish__initialize :** permet de créer un nouveau "fish" avec un id, une taille, une position initiale et un type de mobilité.
- **fish__update :** permet de calculer la nouvelle destination d'un "*fish*" en fonction du type de sa mobilité.

mobility.h

Le module ***mobility*** contient la structure qui représente une mobilité dans l'aquarium, une énumération de tous les types de "mobility" possibles, et les fonctions nécessaires pour la gestion de la structure. Les fonctions les plus importantes sont décrites ci-dessous :

- **mobility__init :** permet de créer une nouvelle "mobility".
- **mobility__calculate :** calcule la destination d'un poisson à laquelle il doit se rendre dans une période définie selon sa position actuelle et son type de mobilité comme illustré par la figure 4.

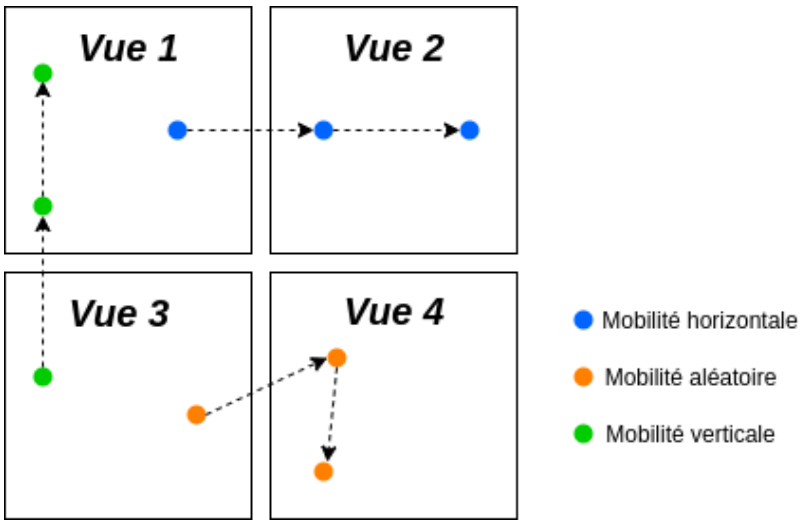

FIGURE 4 – Calcul périodique des destinations selon le type de mobilité

view.h

Le module ***view*** contient la structure et les fonctions qui permettent de gérer une vue de l'aquarium. Pour ce faire, nous avons besoin principalement d'assigner ou éventuellement de détacher une vue à un client d'affichage, d'initialiser une vue à partir d'une description en texte, ainsi que de pouvoir basculer des coordonnées globales dans l'aquarium et les coordonnées locales dans la vue. En effet, toutes sont fonctions sont implémentées dans le "module" ***view*** et dont les principales sont les suivantes :

- **view__assign / view__disallow :** permettent d'assigner et de détacher respectivement une vue à un client d'affichage.
- **view__init_from_string :** permet d'initialiser une vue à partir de sa description en texte de type "*N2 500x0+500+500*".

prompt.h

Le module ***prompt*** permet de gérer les commandes que donne l'utilisateur au contrôleur de type : `add aquarium, show aquarium`. Les fonctions principales qui composent ce module et qui permettent de gérer les commandes au niveau du prompt utilisateur du contrôleur sont :

- **cmd__parser :** permet d'analyser la commande (chaîne de caractère) et fait appel à l'une des fonctions ci-dessous en fonction du contenu.
- **cmd__load :** appelée si la commande est du type "load aquarium1".
- **cmd__add :** appelée si la commande est du type "add view N5 400x400+400+200".
- **cmd__show :** appelée si la commande est du type "show aquarium".

Cette liste de modules n'est pas exhaustive, en effet d'autres modules utilitaires ont été utilisés pour notre implémentation interne du côté du contrôleur à savoir le module ***list*** et les parseurs des fichiers à savoir ***aquarium__parser*** et ***config__parser***, en

plus de l'implémentation de plusieurs tests permettant de valider l'implémentation de ces différents "modules".

3 Affichage

Le programme d'affichage représente le programme client qui se connecte au contrôleur au démarrage du programme et reste connecté durant toute la durée de vie de la session. Afin de développer ce programme l'architecture de classes suivante a été crée :

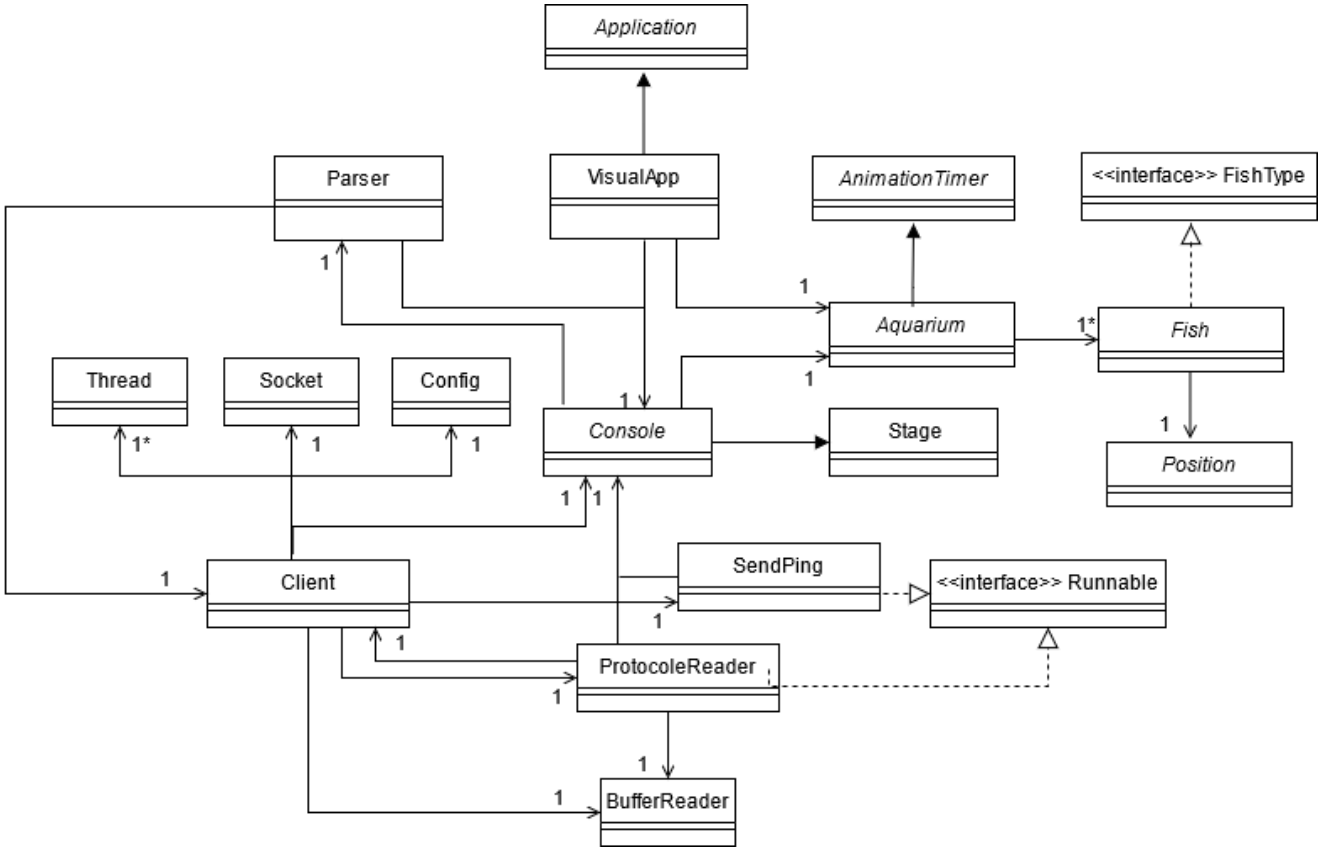

FIGURE 5 – Architecture du programme d'affichage

Aquarium

Cette classe permet d'afficher une vue de l'aquarium et d'animer les poissons qui s'y trouvent. Elle permet l'ajout et la suppression de poissons `Fish`, l'affectation de listes des positions futurs présentées par une classe `Position` aux poissons appartenant à la vue et de gérer les déplacements dans l'animation visualisée.

Fish

Cette classe représente un poisson défini par son ID et son type afin de pouvoir lui assimiler une image. Elle transforme les instances de la classe `Position` en des déplacements sur la vue en respectant la durée et les points de départ et d'arrivée. Elle gère aussi les rotations des poissons dépendamment des directions de mouvement afin d'avoir un effet plus réaliste.

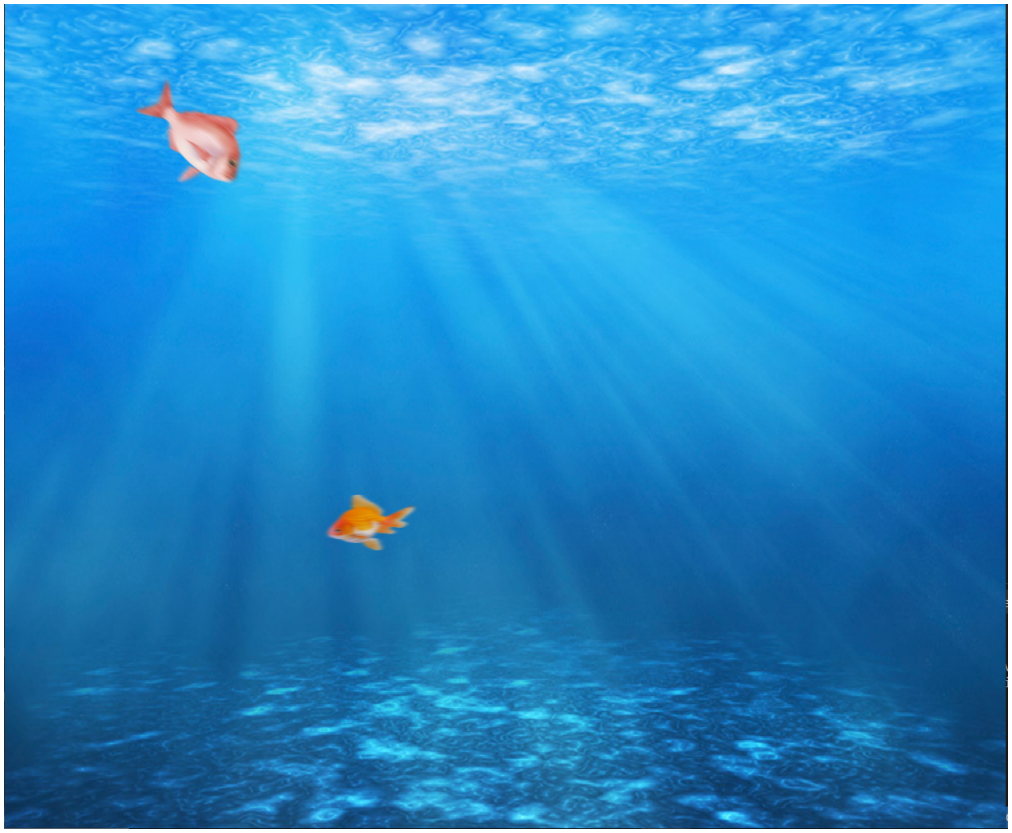

FIGURE 6 – Vue animée de l'aquarium

Console

Cette classe construit un prompt utilisateur d'affichage (IHM), qui permet à l'utilisateur d'interagir avec le contrôleur en envoyant plusieurs commandes comme : `status`, `addFish`, `delFish` ... . Cette interface permet de se rappeler de l'ensemble des commandes envoyées par l'utilisateur dans un historique, ainsi l'utilisateur peut revenir vers des commandes tapées précédemment. De plus, ce prompt fournit dans le cadre d'une barre d'outils un bon nombre d'informations utiles pour l'utilisateur : L'ensemble des poissons qui existent, les types de mobilités, un menu des commandes possibles, ainsi que certaines informations sur le projet.

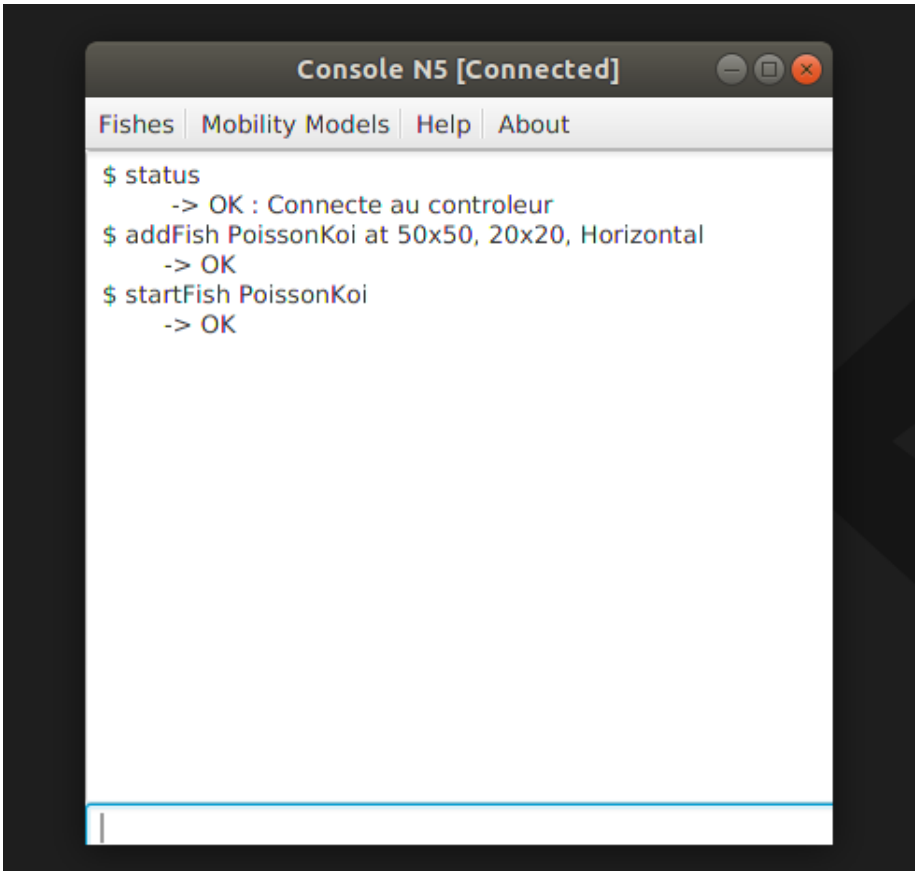

FIGURE 7 – Prompt utilisateur d'affichage

Client

Cette classe définit le client, l'élément qui permettra la communication TCP avec la socket. Afin d'établir une connexion avec le contrôleur, le client fait appel à la méthode `config()` celle-ci configure d'abord la connexion en parsant le fichier **affichage.cfg** à l'aide de la classe **Config**, puis elle essaye de connecter le client au serveur, une fois la connexion est établie cette fonction lance un **ProtocoleReader**, c'est un thread qui lira sur la socket et gérera la réponse du serveur.

Parser

Les commandes entrées par l'utilisateur au niveau du terminal doivent être parsées par le programme d'affichage, c'est exactement le rôle de cette classe. Dans le cas de la validité syntaxique de la commande, celle-ci est stocké dans la **commandList** (les traitements à éxecuter par le thread) du **ProtocoleReader** et puis envoyée par le client au contrôleur. Sinon, un message d'erreur est affiché au niveau du prompt.

4 Protocole des communications internes

Chaque programme d'affichage communique régulièrement durant toute sa durée de vie avec le contrôleur, ces communications s'effectuent avec le protocole TCP.

4.1 Côté Affichage

Comme mentionné dans la partie 3, le thread **ProtocoleReader** lit sur la socket et gère la réponse du serveur, cela est fait par la méthode `run()`.
Plusieurs réponses sont gérés, si par exemple la réponse du contrôleur est **"NOK"**, alors le thread supprime la commande stocké dans sa **commandList** sinon si la réponse est **"OK"**, alors ce dernier appelle une méthode `handleAcceptServer()`, cette méthode parcourt la **commandList** et exécute l'ensemble des commandes stockées (addFish, delFish, startFish...).
Si le contrôleur retourne au programme d'affichage la liste des poissons que doit gérer ce dernier, le thread fait appel à la méthode `handleDestination(String[] args)`, celle-ci parse cette réponse et met à jour l'aquarium en ajoutant l'ensemble des nouvelles positions envoyées.

4.2 Côté Contrôleur

Le fichier **serveur.c** permet de gérer les connexions du côté serveur. Il permet de gérer plusieurs clients en même temps, en utilisant un pool de threads et l'appel système **select**. Les étapes suivantes résume le fonctionnement du serveur :

- Créer une socket
- Attacher l'adresse IP du serveur à la socket à l'aide de l'appel système `bind()`
- Garder la liste de toutes les sockets ouvertes dans un set nommé `current_sockets`. initialement, ce set contient seulement la socket du serveur. Le thread qui exécute `server__wait_connections()` se charge d'accepter les nouvelles connexions et les ajouter au set.
- Tant que le contrôleur est actif, on appelle `select`, et on lui passe en paramètre le set `ready_sockets` qui contient une copie du set `current_sockets`. En fait, `select` est déstructive, il va détruire le set des sockets qui lui est fourni et le remplacer par la liste des sockets sur lesquelles on peut performer une action de lecture ou d'écriture sans bloquer.
- Les sockets qui se trouvent dans le set `ready_sockets` sont ajoutés à une file
- Un total de 5 threads sont lancés pour gérer les messages des clients en parallèles. Il vérifient constamment si la file contient des sockets en attente de traitement. Si c'est le cas, il appelle les fonctions adéquates pour gérer les messages selon les cas. L'accès concurrents des threads à la queue des clients est géré à l'aide de `mutex_queue`, de même pour les accès aux sets `ready_sockets` et `current_sockets` on utilise une `mutex_socket`. De plus, afin de céder la main au client qui est prêt pour interagir avec le serveur on a utilisé une sémaphore, celle-ci reste bloqué tant qu'il n y a pas un client prêt pour échanger avec le contrôleur .
- La commande `exit` du prompt du contrôleur permet de fermer la socket, et de libérer toutes les ressources mémoires.

Deux autres threads sont lancés, ils se chargent de :

— **Gérer le prompt du contrôleur** il s'agit de traiter les commandes entrées par l'utilisateur au niveau du terminal. Ces commandes concernent : l'affichage de la configuration actuelle de l'aquarium, le chargement d'un aquarium, et la fermeture du contrôleur.

— **Gérer l'aquarium** il s'agit de calculer les nouvelles destinations des poissons, gérer leurs états, et mettre à jour périodiquement l'aquarium.

4.3 Le module *parser.h*

Le module `parser.h` se charge de traiter les commandes du protocole côté serveur. Sa fonction principale `protocol__parser()` permet de prendre un message, le parser, et appeler la fonction adaptée selon le type du message.

4.3.1 Traiter un message de type *"greeting"*

Si le message reçu du client commence par *"hello"*, il est considéré de type *"greeting"*, et la fonction *cmd__greeting()* est appelée. Si :

— **Le client est connecté au serveur** alors un message *'Already greeted'* est renvoyé.

— **Le client n'est pas connecté au serveur** si le message du client est de type *'hello in as Ni*, on vérifie si la vue identifiée par i est disponible, si c'est le cas elle est attribuée au client. Sinon, on lui attribue la première vue libre trouvée. Dans les deux cas le contrôleur communique au client l'id de la vue qui lui est attribuée à travers le message *'greeting Nj'*. Si aucune vue n'est disponible, le contrôleur renvoie *'no greeting'*.

4.3.2 Traiter un message de type *"ping"*

Pour traiter un message de type *'ping'*, on fait appel à la fonction *cmd__ping()* qui renvoie un message *"pong numPort"*. Ce message permet de garder la connexion avec le serveur.

4.3.3 Traiter les message de type *"addFish"*, *"delFish"*, et *"startFish"*

Il s'agit dans cette partie des commandes spécifiques à la gestion des poissons. Pour ajouter un poisson, un afficheur envoie une commande *"addFish TypeFishj at hxw, ixj, MobilityType*. Si le message reçu ne respecte pas la forme indiquée, ou si le *TypeFish* n'existe pas, j est déjà attribué, ou encore *MobilityType* n'existe pas. Il ne sera pas traîté par le contrôleur, et un message *"NOK : typeErreur* sera renvoyé. Si le message est valide, un poisson de type *TypeFish* et d'identifiant j -avec $j = 0$ s'il n'est pas spécifié- et de bundler de taille ixj est ajouté à la position hxw, où h est un pourcentage de la longueur de la vue, et w est un pourcentage de la largeur de la vue. La commande *delFish* permet de supprimer un poisson, de même si le format n'est pas respecté ou le poisson n'existe pas, un message d'erreur est renvoyé.
Quand à la commande *startFish*, elle permet de lancer un poisson. Un poisson ne commence son mouvement qu'après avoir reçu ce message. De même, si le poisson indiquée n'existe pas, un message d'erreur est renvoyé.

4.3.4 Traiter un message de type *"log out"*

Lorsqu'un client envoie le message *'log out'*, il est déconnecté du contrôleur, la vue qui lui était associée est libérée, et un message *'bye'* est renvoyé.

4.3.5 Commande particulière : *"getFishesContinuously"*

Pour gérer son programme d'affichage, le client a besoin d'informations en temps réel sur les poissons présents dans l'aquarium et leurs positions futures. Cela peut se faire à travers la commande **getFishesContinuously** du protocole de communication. En effet, suite à la demande d'un programme d'affichage, le contrôleur envoie de manière périodique une liste des poissons présents dans l'aquarium et les destinations auxquelles ils doivent se rendre avant la réception des prochaines informations. Cela est illustré par la figure 8 qui représente une partie du fichier *log* du côté de l'affichage.

```
> getFishesContinuously

< list
< list
> ping 1234
< pong 1234
> addFish PoissonKoi at 50x50, 50x50, RandomWayPoint
< OK
< list
< list
> ping 1234
< pong 1234
> startFish PoissonKoi
< OK
< list [PoissonKoi at 52x35,50x50,5]
```

FIGURE 8 – Partie du fichier `logs.txt` du côté de l'affichage

Du côté du Contrôleur cet envoi périodique, est géré par le thread qui gère l'aquarium, en effet tant que le programme du côté du contrôleur ne s'est pas encore arrêté, on suspend l'exécution du thread pour une durée de période T définie dans le fichier de configuration du contrôleur, ensuite on rafraîchit l'aquarium c'est à dire on met à jour les positions des différents poissons présents, on calcule leurs nouvelles destinations auxquelles ils doivent se rendre, et on envoie la liste aux différents clients connectés au serveur avant de réitérer ce processus à chaque période T.

Conclusion

La réalisation de ce projet nous a permis de découvrir, et de prendre en main les outils de *la programmation socket* en *C* et en *Java*. Il nous a aussi permis de mettre en oeuvre les connaissances acquises en *programmation concurrente*, et renforcer notre sens du travail en équipe sur un projet de longue durée.
De plus, l'aspect visuel du projet lui donnait un côté concret et tangible qui représentait une source de motivation pour mener à bien le projet et arriver à des résultats intéressants. Les tests utilisateurs, quant à eux, étaient aussi faciles à mettre en place, vu que cela se faisait en écrivant des scénarios et en les appliquant.