



Filière informatique
ENSEIRB-MATMECA

Rapport de Projet de réseaux

Yassir LAFFANI Saad ABIDI
Ahmed ELMARKEZ Guillaume MANNONE Anas TALBI

22 mai 2021

Table des matières

1	Introduction	3
2	Organisation du travail	3
3	Programme Contrôleur	3
3.1	Structure principale	3
3.2	Choix des structures de données	3
3.3	Parser	4
3.4	Clients	5
4	Programme d’affichage	6
4.1	Structure principale	6
4.2	Spécification du code	6
4.3	Interface graphique	8
4.4	Les difficultés rencontrées	9
5	Conclusion	10
5.1	Objectifs atteints	10
5.2	Pistes d’améliorations	10

1 Introduction

Le but de ce projet est de pouvoir réaliser un aquarium qui contient des poissons en mouvement. Ce dernier nécessite deux programmes : Un programme contrôleur jouant le rôle du serveur qui permet la gestion des poissons dans tout l'aquarium et d'un programme d'affichage qui représente le client et permet de visualiser une partie de l'aquarium que nous allons appeler par la suite une vue. Le serveur fournit les informations concernant les positions des poissons aux programmes d'affichages pour qu'ils puissent représenter ces derniers dans leurs vues.

2 Organisation du travail

Pour l'organisation du travail, on s'est mis d'accord en premier sur l'ensemble des tâches à faire. Après, on s'est réparti en deux groupes. Le premier s'est occupé de la partie d'affichage, et le second de la partie du serveur, tout en gardant la communication entre les deux groupes.

3 Programme Contrôleur

L'objectif du contrôleur est de traiter les informations relatives aux données des clients, ainsi que de leurs requêtes et de pouvoir enregistrer les modifications de leurs informations.

Pour se faire, chaque client dispose de plusieurs commandes que celui-ci peut utiliser afin de mettre à jour ses informations d'affichage.

3.1 Structure principale

En analysant le sujet nous avons eu une idée claire des structures nécessaires pour le développement du contrôleur. Cette analyse nous a permis de découper le contrôleur en plusieurs fichiers.

3.2 Choix des structures de données

Au début nous avons travaillé sur des tableaux pour représenter les poissons et les vues mais il s'est avéré que pour connecter plusieurs clients et pour une bonne gestion de mémoire il fallait utiliser une structure de données plus adaptée, donc nous avons utilisé des files pour gérer les données.

3.2.1 Fish

Ce fichier nous a permis de représenter les poissons. Chaque poisson possède un nom, une position, des dimensions, un int *is_started* qui permet de savoir si le poisson à démarrer ou non ainsi que 3 champs qui permettent de gérer la mobilité du poisson.

Ce fichier contient des fonctions permettant la gestion des poissons tels que l'initialisation du poisson, l'ajout du poisson dans l'aquarium, le démarrage du poisson et aussi des fonctions pour récupérer les informations utiles du poisson dans un buffer ou la fonction permettant de savoir si le poisson est dans une vue.

3.2.2 View

Cette structure permet de contenir les poissons. Elle possède plusieurs champs à savoir : le nom, l'ID, les dimensions, les coordonnées d'origine, ainsi qu'un entier permettant de savoir si la vue est libre ou non.

Ce fichier contient des fonctions pour manipuler les vues à savoir ; ajouter ou supprimer une vue,

l'enregistrer, l'afficher, la marquer comme libre, ainsi qu'une fonction permettant de trouver une vue à partir de son nom (cette fonction serait utile dans la partie du parseur).

3.2.3 Mobilité

Pour gérer la mobilité des poissons on a utilisé une structure qui représente le trajet du poisson. Elle contient le temps d'arrivée du poisson et ses nouvelles coordonnées. On a utilisé des fonctions adaptées pour chaque mouvement : verticale, horizontale, aléatoire. ainsi qu'une fonction qui permet d'appliquer la mobilité demandée au poisson.

3.2.4 Aquarium

L'aquarium contient toutes les structures précédentes ainsi que ses dimensions. Ce fichier contient des fonctions permettant la manipulation de l'aquarium tel que : l'initialisation de l'aquarium il faut donc initialiser les vues, les poissons, et les mobilités des poissons, il contient aussi des fonctions permettant de charger l'aquarium, l'enregistrer et l'afficher. Pour savoir si l'aquarium est chargé ou non, on a déclaré une variable globale.

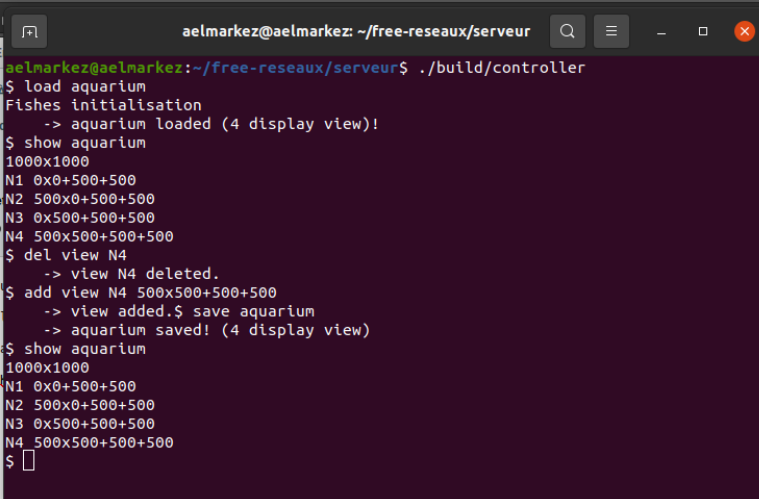
3.3 Parser

Pour pouvoir lire les commandes de l'utilisateur et les traiter nous avons besoin d'un parser qui analyse les chaînes de caractères pour en identifier le sens. On a donc deux types de commandes à traiter côté serveur et côté client.

3.3.1 Commandes côté serveur

Les commandes imposées par le sujet côté serveur sont load aquarium pour charger l'aquarium, save aquarium pour l'enregistrer, et show aquarium pour l'afficher. En plus des commandes de l'aquarium il faut gérer les commandes des vues à savoir : addView qui permet d'ajouter une vue à l'aquarium, delView pour la supprimer. On a donc implémenté une fonction *parse_command* qui parse toutes ces commandes en utilisant les fonctions présentes dans la bibliothèque *string.h* à savoir : **strcmp**, **strncmp** et **sscanf** en plus des fonctions implémentées dans les fichiers view.c, aquarium.c.

Voici un exemple du chargement d'une aquarium, son affichage ainsi que l'ajout et la suppression d'une vue.



```

aelmarkez@aelmarkez: ~/free-reseaux/serveur
aelmarkez@aelmarkez:~/free-reseaux/serveur$ ./build/controller
$ load aquarium
Fishes initialisation
-> aquarium loaded (4 display view)!
$ show aquarium
1000x1000
N1 0x0+500+500
N2 500x0+500+500
N3 0x500+500+500
N4 500x500+500+500
$ del view N4
-> view N4 deleted.
$ add view N4 500x500+500+500
-> view added.$ save aquarium
-> aquarium saved! (4 display view)
$ show aquarium
1000x1000
N1 0x0+500+500
N2 500x0+500+500
N3 0x500+500+500
N4 500x500+500+500
$ 

```

FIGURE 1: Exemple de commandes internes au serveur

3.3.2 Commandes côté client

Concernant les commandes côté client, Nous avons implémenté toutes les commandes demandées par le sujet à savoir ; **Hello**, **AddFish**, **startFish**, **delFish**, **status**, **log out**, **ping**, **getFishes** et **getFishesContinuously**. Par exemple pour implémenter la commande **addFish**, nous sommes passés par un handler qui répond au message du client : Si c'est un **Hello** sans argument le handler lui attribue un identifiant de vue libre, si le client ne spécifie pas l'identifiant renvoie si possible un autre identifiant libre sinon une erreur. Pour répondre à la demande continue des poissons nous avons implémenté la commande **getFishes** qui retourne au programme d'affichage la liste des poissons que doit gérer ce dernier en effet on stocke les informations du poisson dans une chaîne de caractère pour les envoyer au client avec la commande (send) de la bibliothèque **socket.h**. Pour gérer la demande continue des poissons nous avons donc implémenté la commande **getFishesContinuously** qui envoie de nouvelles mobilités toutes les 100 ms. Cela est fait grâce à la méthode **getFishes** et en mettant à jour le champ de la structure du client.

3.4 Clients

De par l'existence de plusieurs clients, il est nécessaire de pouvoir traiter plusieurs requêtes en même temps venant de plusieurs clients, il faudra donc avoir plusieurs sockets, pour être capable de distinguer plusieurs clients différents et ne pas avoir un appel bloquant dès le premier client faisant une requête.

Afin de réaliser cela, nous avons implémenté un serveur multi socket en utilisant les fonctions de la bibliothèque `<sys/socket.h>`. pour le réaliser, nous utilisons les fonctions et objets suivants :

- `setsockopt(int socket, int level, int option_name, const void *option_value, socklen_t option_len);`

permet de définir l'option spécifiée par l'argument `option_name`, au niveau du protocole spécifié par l'argument `level`, à la valeur indiquée par l'argument `option_value` pour la socket associée au descripteur de fichier spécifié par l'argument `socket`.

- `int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen);`

Lit sur les socket file descriptor `&read_fds` et retourne quand l'on peut lire quelque chose dans l'un d'eux.

- `int accept(int sock, struct sockaddr *adresse, socklen_t *longueur);`

Cette fonction accepte la connexion d'un socket sur le socket `sock`. Le socket aura été préalablement lié avec un port avec la fonction `bind`.

- `int listen(int s, int backlog);`

permet de marquer la socket référencée par `sockfd` comme une socket passive, c'est-à-dire comme une socket qui sera utilisée pour accepter les demandes de connexions entrantes en utilisant `accept()`.

De même, il est nécessaire de pouvoir traiter plusieurs requêtes en même temps étant donné qu'il est possible de communiquer avec plusieurs clients, ainsi il faudrait avoir un traitement multithreadé des requêtes.

Pour le réaliser, nous utilisons plusieurs fonctions ainsi que la librairie `<pthread.h>` :

- `pthread_init(pthread_data_t, i, sockfd, &client_address, file)`

qui permet d’initialiser les données fournies au thread ainsi que sont id.

— `pthread_create(thread_id, NULL, (void *) (socket_corresp), arguments)`

créer le thread en faisant appel à la fonction `socket_corresp` qui parse le message passé dans le buffer et appel la fonction nécessaire en fonction du message envoyé.

— `pthread_join(thread_data[i].thread_id, retval)`

attend que tous les threads aient retourné avant la fin d’une boucle de lecture sur les sockets file descriptor.

Ainsi, il est possible d’avoir plusieurs clients et de traiter toutes leurs commandes en même temps.

4 Programme d’affichage

Le programme d’affichage est un programme client qui a pour objectif l’échange de données avec le contrôleur afin d’afficher une partie de l’aquarium. Pour cela, il nous a fallu d’abord choisir un langage de programmation. Une analyse des avantages et des inconvénients a mené vers le choix du langage Java. Ce choix a été fait à la lumière des raisons suivantes :

- Tous les membres de l’équipe maîtrisent le langage java.
- Le langage java dispose d’une librairie `JAVAFX` très puissante et facile à manipuler permettant la création d’une interface graphique que nous allons utiliser pour la représentation de nos vues.
- Le choix d’un langage de programmation différent de celui utilisé pour le contrôleur qui est le `c` au niveau des méthodes de gestion de sockets.

4.1 Structure principale

L’analyse du sujet nous a permis d’avoir une idée claire sur les différentes classes qui sont nécessaires à la conception de l’afficheur. Cette conception nous a permis d’avoir six classes que nous avons regroupées dans un paquetage **view**. Le diagramme suivant met en évidence la structuration des classes qui implémentent l’afficheur.

L’explication de ce diagramme ainsi que la justification des différentes utilisation de chaque classe sera détaillé dans la partie suivante.

4.2 Spécification du code

Notre dossier **src** contient nos six classes mentionnées ci-dessus. Chaque fichier prend en charge une fonctionnalité que nous avons jugé nécessaire pour notre programme.

4.2.1 La classe **Fish**

Cette classe permet la représentation d’un poisson. Chaque poisson a un nom (qui sert aussi d’identifiant). Le poisson a également une image suivant son nom que nous allons utiliser pour le représenter dans notre affichage. Il a aussi des coordonnées dans le plan et un but (goal) qui représente son but. Le poisson se déplace vers son but quand son booléen `started` est activé (vrai).

Cette classe contient plusieurs méthodes permettant la manipulation des poissons tels que la récupération des images représentant le poisson à partir du dossier **resources** ainsi qu’autres qui permettent de bien définir son goal prochain et de modifier ses données. Parmi ces méthodes, on peut citer :

- La méthode **setGoal** : permet de définir la position prochaine du poisson dans l’afficheur, elle prend en argument la position du poisson, le temps à passer pour arriver au goal, ainsi que le temps écoulé.

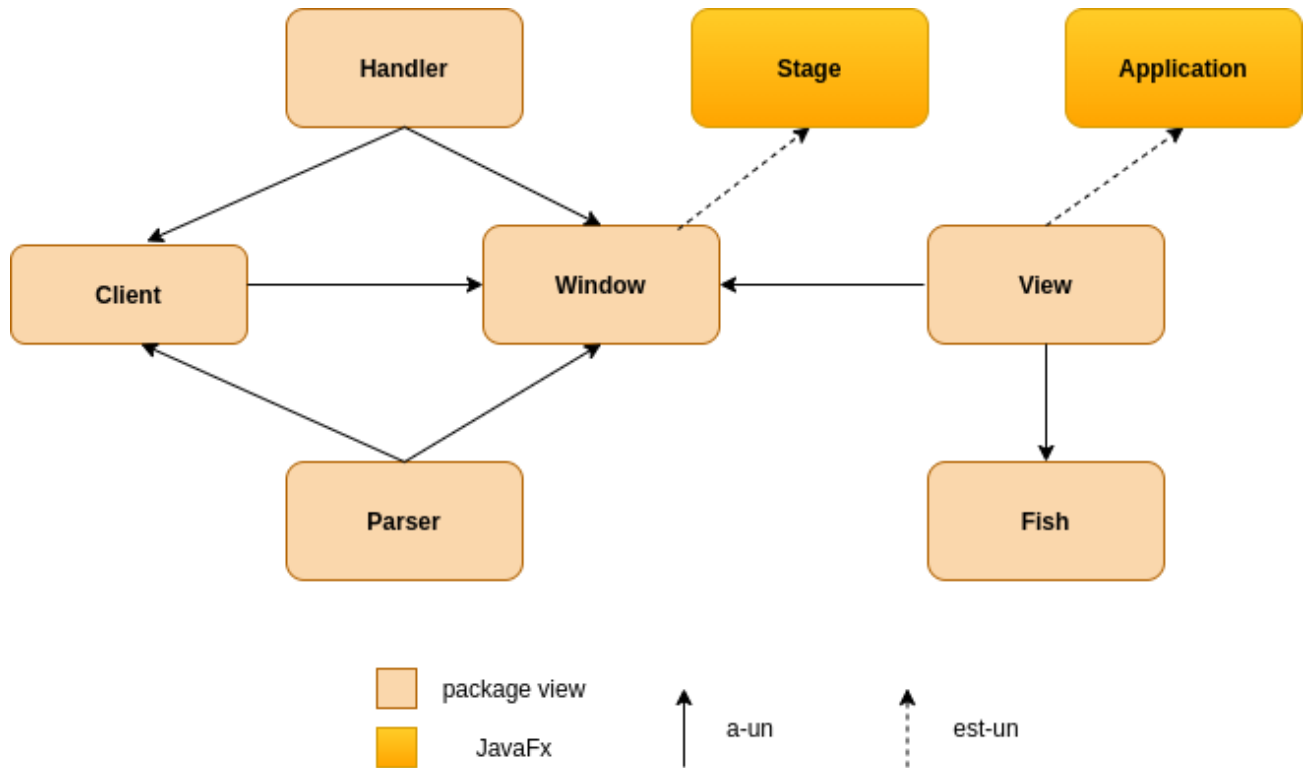


FIGURE 2: Structuration des classes de l'afficheur

- La méthode **update** : met à jour la position du poisson pour lui permettre d'atteindre son goal qui utilise la méthode **setGoal**.

4.2.2 La classe Handler

La classe Handler s'occupe de la communication entre l'afficheur et le <https://www.overleaf.com/project/60a81c4...> serveur : récupère tout d'abord l'adresse ip du serveur et le numéro du port du fichier "affichage.cfg", crée une socket qui lui est propre et établit la connexion avec le serveur via la méthode

4.2.3 La classe Client

La classe Client s'occupe de la communication entre l'afficheur et le serveur : récupère tout d'abord l'adresse ip du serveur et le numéro du port du fichier "affichage.cfg", crée une socket qui lui est propre et établit la connexion avec le serveur via une méthode de configuration et stocke les messages d'entrée du serveur et les fait passer au handler qui les gère par la suite. Après avoir analysé les réponses de l'utilisateur, le parser les envoie au serveur et se déconnecte du serveur quand l'utilisateur le souhaite.

4.2.4 La classe Parser

Cette classe s'occupe d'analyser les demandes de l'utilisateur. Selon l'entrée passée, elle contient une méthode **parser()** qui vérifie la validité syntaxique de la demande. Si la demande est connue, on envoie la demande au client sinon on signale la non-conformité de la demande.

4.2.5 La classe Window

C’est la fenêtre qui permet l’interaction avec l’utilisateur en permettant à l’utilisateur de fournir des commandes et de recevoir des réponses de la part du serveur. On a défini dans cette classe un historique de commande qui permet à l’utilisateur de revenir aux commandes précédentes juste en cliquant sur les boutons de navigation dans le clavier (les flèches). On y initialise aussi les différentes propriétés de notre fenêtre à savoir les alertes, le menu, la barre où on écrit nos commandes, et le champ permettant d’afficher les commandes passées et les réponses du serveur.

4.2.6 La classe View

C’est l’application de notre interface graphique qui permet l’affichage d’une partie de l’aquarium et permet aussi de lancer la console window. Cette classe contient plusieurs attributs à savoir : la largeur et la hauteur de notre fenêtre, l’identifiant d’affichage, la liste des poissons qui se trouvent dans l’afficheur, et d’autres attributs nécessaires pour la manipulation de l’interface graphique. Elle manipule aussi les poissons dans la fenêtre (ajout, suppression, etc...). Elle contient des méthodes d’initialisation des différents paramètres de configuration, et d’initialisation des outils d’affichage.

4.3 Interface graphique

L’interface graphique de notre programme d’affichage a été réalisée en utilisant la bibliothèque JAVA FX. Pour chaque programme d’affichage qui est connecté au serveur, nous générons deux fenêtres :

La première fenêtre qui est représentée dans notre structure par la classe window est celle qui permet à l’utilisateur de communiquer avec le serveur, et donc c’est une fenêtre de "chat" qui lui permet de faire passer des commandes pouvant être introduit dans un champ input et aussi d’afficher les messages retournés par le serveur.

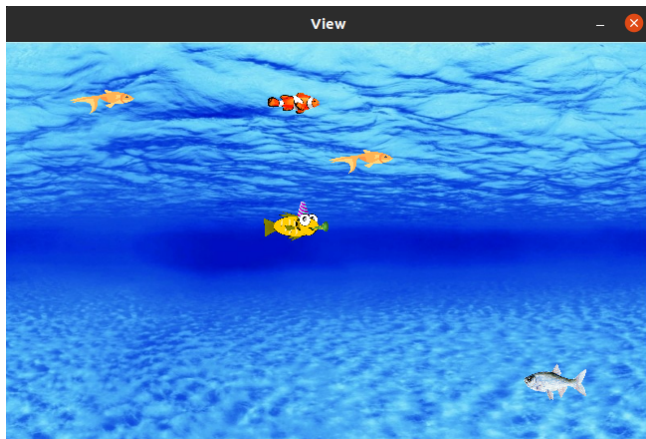
Tandis que la deuxième fenêtre est celle qui représente la vue du programme d’affichage et donc elle permet de visualiser une partie de l’aquarium. Après ajout de poissons dans l’aquarium et les autorisant à se déplacer suivant leur modèle de mobilité, nous pourrions observer qu’ils ont changés de positions quand ils sont dans la zone de notre vue en écrivant un `getFishes` dans la première fenêtre, ou même les observer en train de se déplacer en appelant `getFishesContinuously`.

Afin de représenter les poissons nous avons choisi différentes images avec des fonds transparents pour un affichage plus beau lors de la superposition des images des poissons sur l’arrière-plan de la vue. Voici les modèles de poissons que nous avons choisi :

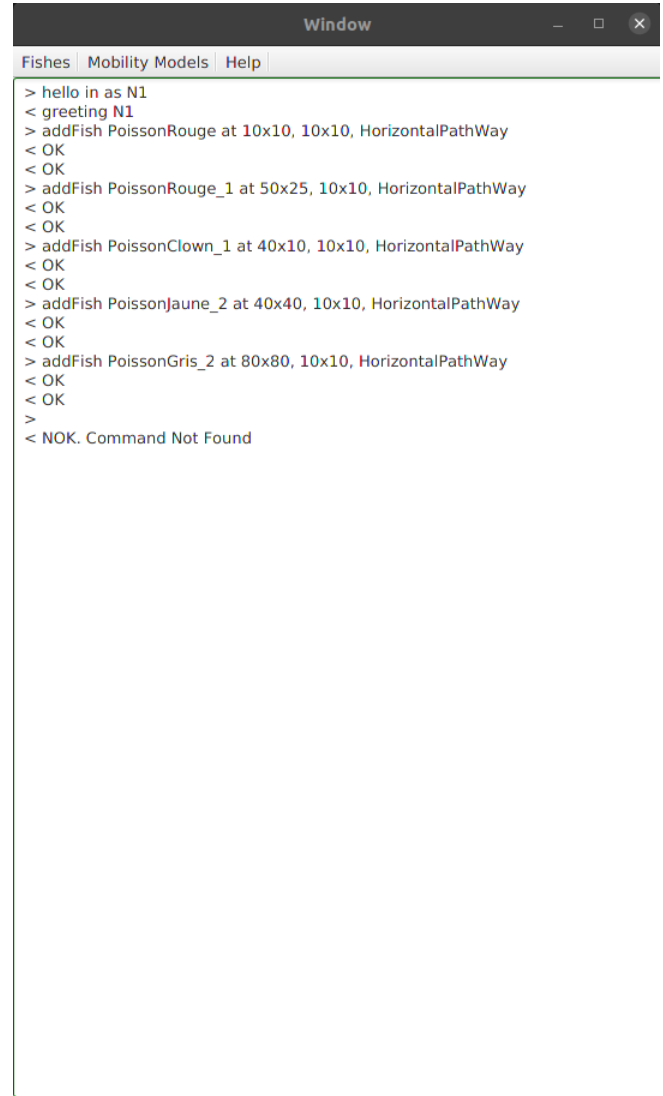


FIGURE 3: Poisson (a)Clown (b)Jaune (c)Rouge (d)Gris

Après un établissement de connexion avec le serveur, l’attribution d’une vue à notre programme d’affichage par ce dernier et l’ajout d’un nombre de poissons, voici à quoi ressemblent nos deux fenêtres console et view :



(a)



(b)

FIGURE 4: Poisson (a)Clown (b)Jaune (c)Rouge (d)Gris

4.4 Les difficultés rencontrées

Lors de la réalisation du programme d’affichage, nous avons fait face à plusieurs problèmes :

Premièrement, certains membres de l’équipe n’ont pas réussi à installer proprement la bibliothèque JavaFx dans leurs machines, ce qui a rendu la tâche plus difficile puisque tout le monde ne pouvait pas visualiser l’interface et donc pouvoir tester le programme.

Un second problème rencontré est que le programme d’affichage a été réalisé avant que le serveur soit complété, ce qui nous a mené à perdre du temps à l’attendre pour pouvoir visualiser les fenêtres et commencer les tests.

Durant la réalisation de quelques scénarios, plus précisément dans le cas d’un appel à un `getFishesContinuously` le programme s’arrête au bout d’un moment à cause de la saturation de la console par les listes envoyées par le serveur.

5 Conclusion

5.1 Objectifs atteints

Nous avons pu réaliser lors de ce projet deux programmes serveur/client qui permettent les fonctionnalités suivantes :

Le programme contrôleur peut établir des connexions avec plusieurs clients à la fois. Lors des simulations, nous avons pu établir la connexion avec 4 programmes d'affichages avec une vue qui est propre à chacun de ses affichages.

Toutes les commandes demandées dans le sujet ont été effectuées sauf la commande “ls” qui est similaire à notre commande getFishes.

L'implémentation de 3 méthodes de mobilités pour les poissons qui sont les suivantes : **RandomWayPoint**, **HorizontalPathWay** et **VerticalPathWay** qui déplacent les poissons respectivement de façon aléatoire, horizontale et verticale.

La réalisation de scénarios de déplacement des poissons dans la même vue et entre les vues.

5.2 Pistes d'améliorations

Pendant ce travail on a pu réaliser la majorité des fonctionnalités principales demandées. Cependant, on a pas pu traiter le cas de saturation lors de l'appel à la commande **getFishesContiuously** dans la console. En effet après un nombre important d'appels à getFishes la console n'arrive plus à afficher les messages reçus ce qui arrête la console.

Comme piste d'amélioration, on pourra considérer l'amélioration de l'interface graphique de la console par exemple ajouter de l'indentation chose qui va faciliter l'écriture des commandes pour l'utilisateur, on pourra ajouter un mode sombre à la console qu'on peut ajouter en cochant un box qui sera placé dans le menu de notre console, et on pourra ajouter la modification de l'image du poisson selon sa direction. En ce qui concerne les améliorations du serveur, on peut ajouter d'autre mode de mobilités afin de voir un mouvement réaliste du poisson.