Filière informatique
ENSEIRB-MATMECA

— PROJET DE RÉSEAUX —

Simulation d'un aquarium de poissons

Imad BOUDROUA
Nolan BREDEL
Enzo CARRÉ
Benjamin DARMET
Saad ZOUBAIRI

1 Introduction

Ce projet s'inscrit dans le cadre des projets du semestre 8 et vient compléter le module sur les applications TCP/IP. Le but du projet est la réalisation d'un aquarium centralisé de poissons. Ainsi, il consiste en la réalisation d'un programme d'affichage représentant une interface homme/machine contenant différents afficheurs connectés à travers un contrôleur. La figure 1 représente l'interaction entre les différentes parties du projet.

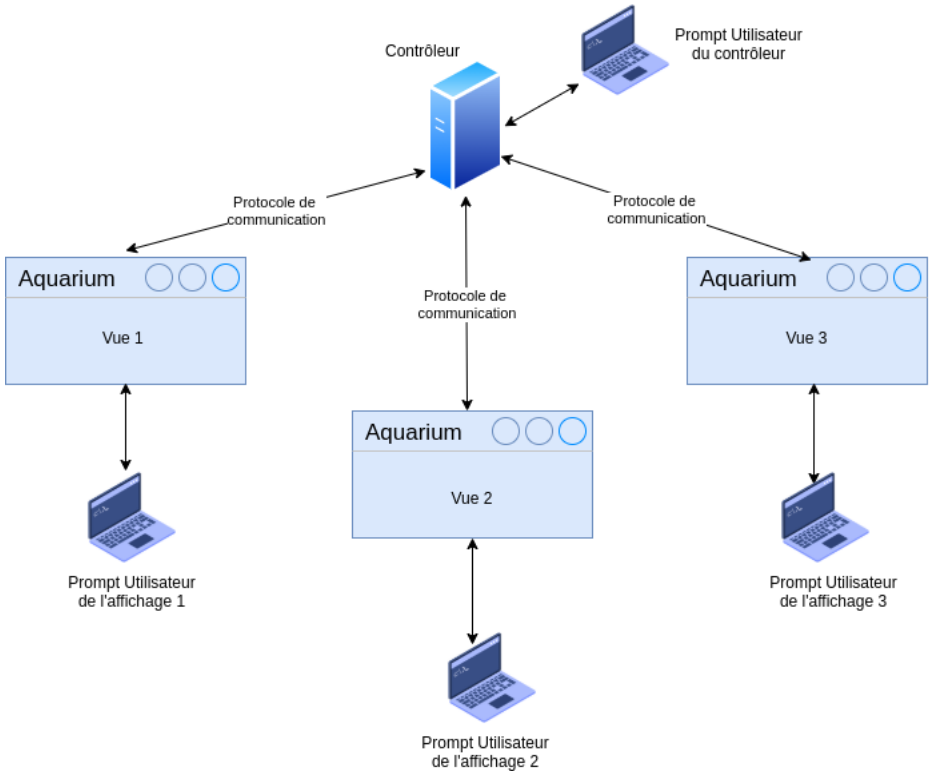

FIGURE 1 – Schéma d'interaction entre l'utilisateur, le programme d'affichage et le contrôleur

2 Objectifs et organisation du projet

L'objectif principal est de réaliser un aquarium centralisé de poissons composé d'un programme d'affichage et d'un programme contrôleur. Le contrôleur permet de centraliser la gestion de l'aquarium. Chaque poisson a pour mission de se déplacer dans l'aquarium selon un modèle de mobilité prédéfini (vitesse, direction, etc.). Le programme contrôleur s'occupe d'informer les programmes d'affichage (Vue) de la position actuelle des poissons.

Nous avons donc séparé notre groupe de travail en deux équipes. Une équipe de trois personnes s'est occupée de mettre en place le serveur et les diverses fonctionnalités qui lui sont liés, tandis que l'autre équipe composée de deux personnes s'est chargée de mettre en place un programme client et son affichage graphique. Comme le projet est étalé sur de nombreuses semaines et avec une certaine quantité de travail à réaliser, nous avons opté pour une organisation agile. Nous nous sommes principalement basés sur les outils tels que Kanboard et Telegram/Discord. Kanboard nous a permis de mettre en place les diverses tâches à réaliser durant le projet, de pouvoir les organiser et de constater l'état d'avancement du projet. Néanmoins, il a été difficile de voir directement à travers cet outil le travail réalisé par chaque personne car on ne peut mettre qu'une personne par tâche. Notre communication s'est donc appuyée énormément sur Discord et télégram pour pouvoir échanger entre nous. Nous avons utilisé un git sur la forge *Thor* pour nous permettre de mettre en commun le code produit et plus particulièrement dans l'objectif de le versionner. En effet, comme le travail a été séparé en deux

équipes, qui elle-même se redécoupe le travail, chacun a pu travailler sur sa propre branche. Une fois le travail terminé, nous avons pu fusionner chaque module entre eux. Le code du serveur a été réalisé en C alors que le code côté client a été réalisé en Java avec la bibliothèque **JavaFX** pour l'affichage graphique.

Les tâches côtés clients ont été découpées en deux parties : une partie réseau et une partie graphique. Nous avons tout d'abord réalisé la partie réseau en mettant en place une connexion avec le serveur et un transfert de message avant de mettre en place l'affichage graphique. Côté serveur, nous avons découpé l'implémention de ce dernier en plusieurs modules chacun gérant une partie du travail du contrôleur. Ainsi, nous avons travaillé sur ces différents modules tout au long du projet en commençant par la gestion des clients et des réceptions/envoies de messages. Puis petit à petit l'ajout des différentes fonctionnalités.

3 Fonctionnalités implémentées

Nous avons globalement implémenté la majeur partie de ce qui est demandé pour le projet.

3.1 Côté serveur

Les fonctionnalités suivantes ont été implémentées :

1. Envoie et réception d'un message
2. Connexion / Déconnexion multi-client
3. Gestion des différentes commandes du serveur
4. Gestion des différentes commandes du client
5. Gestion des poissons et de leur déplacement
6. Log (dans le terminal ou dans un fichier)
7. Configuration du serveur depuis un fichier de configuration.

3.2 Côté client

Les fonctionnalités suivantes ont été implémentées :

1. Configuration du client depuis un fichier de configuration.
2. Connexion / Déconnexion au serveur.
3. Réception d'un message envoyé par le serveur.
4. Parsing / Traitement du message.
5. Stockage de l'aquarium et des poissons.
6. Affichage du/des poisson(s) et déplacement dans la vue.

4 Architecture

Nous avons découpé les différentes architectures en modules. Nous présentons tout d'abord l'architecture côté serveur, puis côté client.

4.1 Serveur

Le contrôleur a le rôle principal de rester connecté à tous les clients. En effet, il permet la centralisation de la gestion de l'aquarium en ayant accès à toutes les informations de ce dernier et permet ainsi de synchroniser l'affichage entre toutes les vues grâce à la communication avec les clients, du changement d'état des poissons. La réalisation de cette partie a été décomposée en plusieurs modules qui sont utiles, soit au traitement interne au niveau du contrôleur ou soit au protocole de communication avec les clients. Un client est représenté par une structure, stockant la vue qui lui est affectée, le descripteur de fichier qui lui est associé, un champ `timeout` permettant de savoir depuis combien de temps le client n'a pas communiqué et un champ pour savoir si un client veut qu'on lui envoie en continu les informations concernant les poissons de sa vue.

4.1.1 Threads et envoie continu de donnée

Le serveur fonctionne avec 2 threads. Le thread du programme principal et un thread gérant le temps. Ce dernier permet une itération de boucle par seconde, ainsi on peut gérer le temps secondes après secondes. Ce thread sert donc à la commande ls et à la gestion des mobilités des poissons.
Commençons par expliciter la gestion des mobilités. Un poisson a un comportement qui définit ses possibilités de déplacement dans l'aquarium, (par exemple : aléatoirement, horizontalement etc...) et ce, pour une certaine durée définie en seconde. Ainsi dans la structure qui définit ce comportement, il y a un champ qu'on décrémente, chaque seconde pour savoir si le poisson est arrivé à destination ou non. Si cette valeur vaut 0, le poisson est arrivé à destination et le contrôleur doit lui redonner une destination.

L'autre utilité de ce thread est la gestion des envois continue de positions. Nous avons décidé d'implémenter la commande ls, qui envoie la position des poissons aux vues toutes les X secondes (indiqué dans le fichier controller.cfg). Nous avons trouvé redondant l'ajout de la commande getfishcontinuously dans sa première version, et ls étant plus court à écrire. Donc, on regarde si le temps calculé est égal au temps défini pour l'envoi des données concernant les poissons, puis on regarde si le champ de la structure du client indiquant si ce dernier a fait appel à ls, dans ce cas on lui envoie simplement tous les poissons présents dans sa vue.

4.1.2 Gestion des clients

Pour la gestion des clients nous avons opté pour un traitement séquentiel via la fonction `select`. Cette fonction est un appel système qui permet de surveiller une liste de descripteur de fichier et détecte ainsi quand un descripteur est prêt pour une certaine catégorie d'opération (ici nous nous intéresserons à l'état en lecture des descripteurs de fichier puisque cela correspond à un message reçu). Quand un descripteur est considéré prêt, alors `select` retourne et réinitialise entièrement la liste des descripteurs de fichier. Dans notre situation, nous appellerons la fonction sur la liste appelée `readfds`. Il nous suffit alors de placer cette fonction dans une boucle infinie pour gérer plusieurs clients de manière simultanée. Pour rentrer dans les détails de cette boucle, voici un schéma 2 explicatif :

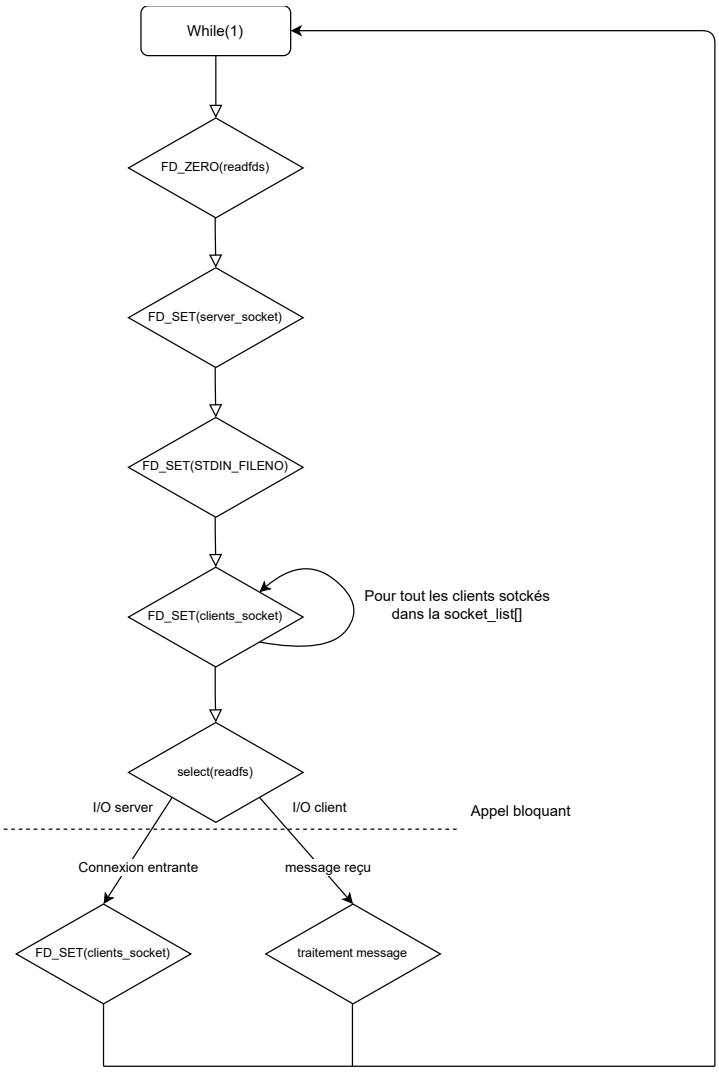

FIGURE 2 – Schéma illustrant la boucle de gestion des interaction I/O

4.1.3 Gestion des aquariums

Ce module sert à la gestion de l'aquarium. Il permet de stocker les vues et les poissons dans des listes. Il permet de sauvegarder l'état de l'aquarium dans un fichier texte. Pour ce faire on stocke d'abord les dimensions de l'aquarium, puis on utilise comme séparateur un V pour signifier que la partie suivante concerne le stockage des vues. Enfin on utilise comme séparateur un P pour signifier que la partie suivante concerne les poissons. Pour charger un aquarium on lit simplement ce fichier via la commande `load`.

4.1.4 Gestion des vues

Le module `view` contient la structure et les fonctions qui permettent de gérer une vue de l'aquarium. Ainsi, nous avons eu à attribuer pour chaque vue, un identifiant, une dimension, une position par rapport à l'aquarium ainsi qu'un booléen permettant d'indiquer si la vue est associée à un client. Pour envoyer tous les poissons liés à une vue nous utilisons la fonction `fish_view_decision`, qui permet de vérifier si un poisson est présent ou atteindra la vue en arrivant à sa destination afin d'envoyer une liste au client.

4.1.5 Gestion des poissons

Le module `fish` permet de définir un poisson de l'aquarium. En effet, un poisson est défini par une structure dont les champs définissent un type, une position, une taille, un comportement à suivre durant sa mobilité ainsi qu'un champ qui permet de définir son état actuel (actif ou pas). Les fonctions implémentées permettent d'initialiser un poisson, à travers la fonction `fish_create()`, ou mettre à jour sa position grâce à la fonction `fish_update()`.

4.1.6 Gestion de la mobilité

Le module de mobilité contient la structure qui représente le comportement d'un poisson au niveau de l'aquarium. Comme illustre la figure 3 nous avons essayé d'implémenter 3 méthodes de déplacement : aléatoire, horizontal et vertical. La fonction **behaviour_init()** permet d'initialiser la structure définie, ainsi que la fonction **behaviour_calculate()** permet de calculer la nouvelle durée de déplacement du poisson ainsi que sa nouvelle destination. Nous nous assurons à chaque calcul de destination que le poisson ne soit pas à l'extérieur de l'aquarium.

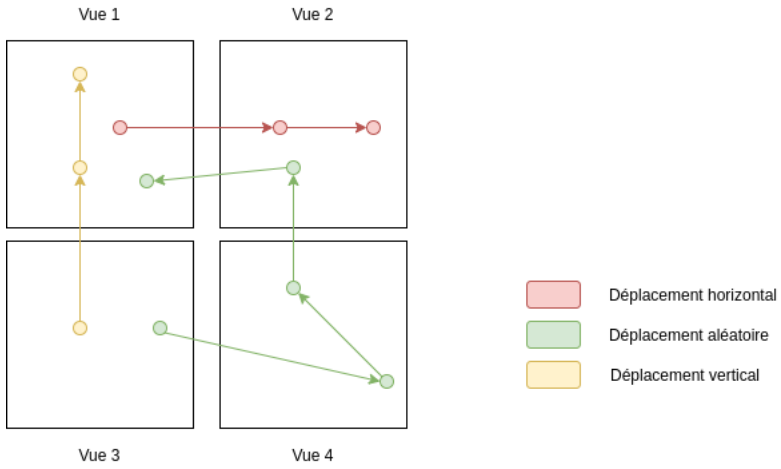

FIGURE 3 – Schéma illustrant les types de déplacement implémentés

4.1.7 Commande du terminal

Les commandes du terminal sont lues avec l'appel système `read` et stockées dans un buffer. On envoie ensuite le contenu de ce buffer dans la fonction `check_cmd_server` afin de déterminer quelle est la commande rentrée. Cette fonction renvoie ensuite le type de la commande à exécuter. Tout ce processus est englobé par la fonction `parse_cmd_server`, à savoir :

- **add** : Pour une commande de type `add view N1 0x0+500+500`.
- **del** : Pour une commande de type `del view N1`
- **load** : Pour une commande de type `load aquarium1`
- **show** : Pour une commande de type `show aquarium1`
- **save** : Pour une commande de type `save aquarium1`
- **exit**

Une fois le type de la commande déterminée, on utilise la fonction `execute_cmd_server` pour exécuter et vérifier que les arguments donnés sont conformes à ceux attendus par la commande. Une fois cela fait et validé, elle est exécutée. Par rapport au sujet nous avons rajouté la commande exit afin de pouvoir quitter le serveur correctement.

4.1.8 Commande du Client

La gestion des commandes du client est très similaire à celle des commandes du terminal. Le buffer contenant les informations transmises via le réseau est passé à la fonction `cmd_parser` qui détermine le type, à savoir : `hello, hello in, status, addfish, delfish, startfish, getfish, ls, logout, ping`.
Une fois le type déterminé, on vérifie que les arguments sont conformes, puis on l'exécute via la fonction `execute_cmd` et encore une fois tout ce processus est englobé par la fonction `parse_cmd`. C'est-à-dire remplir un autre buffer correctement vis-à-vis de ce qu'à demander le client puis on lui envoie.

4.1.9 Traitement d'un message de type « greeting »

Il existe deux types de commande que le contrôleur peut recevoir : « hello » ou « hello in as Nx ». Commençons par la commande « hello ». Si le contrôleur reçoit un message « hello », alors il va parcourir la liste des vues stockées et à la première qu'il trouvera de disponible, c'est-à-dire le champ used de la vue à 0, il va le passer à 1 puis renvoyer « greeting N<id de la vue> », sinon il renvoie simplement « no greeting ».

Le cas « hello in as Nx » est un peu différent. Le contrôleur va d'abord commencer par vérifier si la vue associée à l'id "x" est déjà associée. Si c'est le cas alors, on réitère le même procédé que pour hello afin de lui associer la première vue disponible. Dans l'autre cas, on lui associe simplement la vue qu'il a demandée avant de retourner.

4.1.10 Traitement d'un message de type « ping »

Lors du traitement d'un message de type `ping`, nous faisons appel à la fonction **cmd_ping()** qui renvoie un message : « pong [NumPort] ». Ce message permet de garder la connexion entre le client et le serveur puisqu'au bout d'un temps définit dans le fichier de configuration, si le client ne répond pas il sera déconnecté.

4.1.11 Traitement d'un message de type « addFish », « delFis », « startFish »

Nous nous intéressons maintenant à la partie qui permet la gestion des poissons. En effet, pour ajouter un poisson au niveau de l'aquarium, le client doit envoyer une commande de type `"addFish TypeFish at XxY, WxH, BehaviourName"`. Dans le cas où le message reçu ne respecterait pas la forme indiquée ou le type n'est pas valide ou encore le BehaviourName n'existe pas, le message ne sera pas traité par le contrôleur et un message d'erreur`NOK: Command not found` sera renvoyé. Si le message est valide, un poisson de type `TypeFish` et de nom `TypeFish_j`, tel que j représente l'identifiant du poisson, de taille `WxH` est ajouté à la position de coordonnées `XxY`.
La commande **delFish** permet de supprimer un poisson de la vue du client, notons qu'il est nécessaire le nom du poisson (avec son identifiant) au lieu de n'indiquer que le type.
Quant à la fonction **startFish**, elle permet de lancer le poisson donné, puisqu'un poisson n'a la possibilité de commencer à se déplacer que lors de la réception de ce message. De même, si le nom du poison n'est pas indiqué ou n'existe pas un message d'erreur est renvoyé.

4.1.12 Traitement d'un message de type « logout »

Lorsqu'un client envoie un message de type `« logout »`, il est déconnecté du contrôleur, la vue qui lui était associée est libérée ainsi qu'un message `« bye »` est renvoyé.

4.2 Côté client

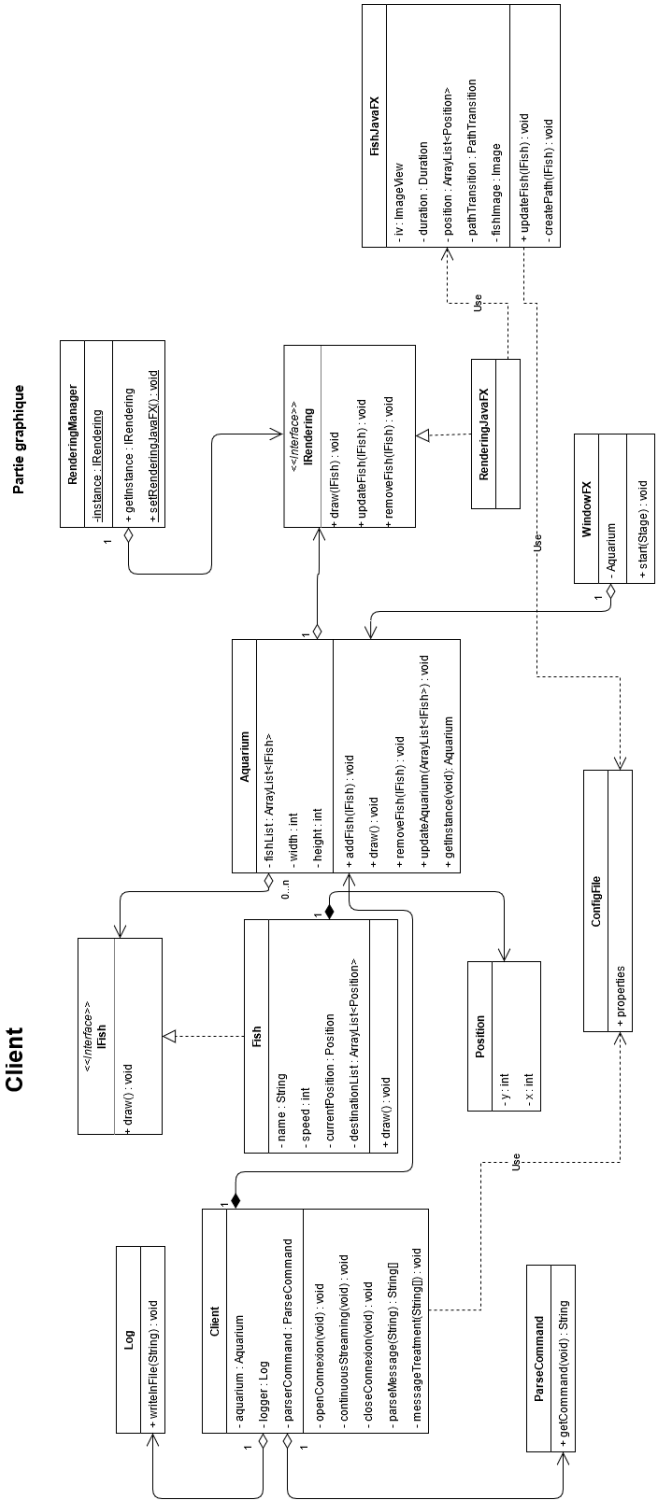

FIGURE 4 – Diagramme UML partie cliente

La partie cliente est divisée en trois sous module : un module réseau, un module modèle et un module affichage. Sur l'architecture cliente, la classe `Main` n'y figure pas. Néanmoins, c'est par son intermédiaire que nous lançons à la fois l'affichage graphique et la partie réseau. Nous détaillons ce fonctionnement au sein des différents modules.

4.2.1 Module réseau

Il concerne principalement les classes Client, Log et ParseCommand. La classe Client va s'occuper de communiquer avec le serveur (ouvrir, échanger et fermer la connexion). Lors du lancement de l'application, la classe Client est placée sur son propre thread d'exécution pour éviter tout blocage avec la partie graphique qui sera elle aussi sur son propre thread d'exécution. Lors de l'initiliasation de cette classe, le client récupère l'ensemble des informations nécessaires pour se connecter (port, nom du serveur, etc) depuis la classe `ConfigFile`. Cette dernière s'occupe de lire chaque ligne du fichier est de stocker la valeur au sein d'un objet static *properties*. Ainsi de cette manière, nous avons la possibilité de consulter le fichier de configuration n'importe où au sein de l'application. Une fois l'objet client initialisé, nous appelons sa méthode `start` qui va mettre en place la connexion avec le serveur. Cette méthode va d'abord ouvrir la connexion en créant un socket et en envoyant le message défini dans le protocole *"hello in as N1"* (N1 étant variable en fonction du fichier de configuration). Si le serveur lui renvoie un *"no greeting"*, la connexion est rompue. Nous fermons alors le socket et nous informons l'utilisateur de la fin de connexion par un *"End of connexion"* dans le terminal. Dans le cas inverse, nous récupérons l'ID donné par le serveur et lançons une communication complète. La méthode `continuousStreaming` est alors appelée. Cette dernière consiste à mettre en place une boucle while qui, tant que la connexion est ouverte (pas de fermeture avec la réception d'un message bye), nous envoyons un message en fonction d'un certain délai. Nous mettons aussi en place l'envoi d'un message ping, dont le délai d'envoi est défini par la valeur présente dans le fichier de configuration. Pour prendre en compte l'envoi des commandes tapées par l'utilisateur dans le terminal, nous passons par l'intermédiaire de la classe `ParseCommand`. Elle même lancée sur son propre thread pour éviter tout blocage, elle va principalement récupérer ce qui est tapé dans le terminal par l'utilisateur et le stocker dans une variable interne. A chaque tour de boucle dans `continuousStreaming`, nous vérifions qu'il n'y a pas eu de commande tapée par l'utilisateur en consultant la méthode `commandToSend`. Si elle nous retourne *True*, c'est que nous avons une commande à récupérer, et nous la récupérons en la plaçons dans le message à envoyer. Le message est alors envoyé par l'intermédiaire de `out.println` et `out.flush()`.

Nous avons donc la possibilité d'envoi de messages au serveur via cette implémentation. Il faut désormais pouvoir traiter la réception des différents messages. La réception des messages depuis le serveur ce fait par un buffer que nous devons lire. Cette lecture se fait via la méthode `readLine`, qui retourne une suite de caractères qui termine par un "_n". Néanmoins, cette méthode est bloquante tant que le message n'est pas terminé par ce caractère. Donc ne pouvons donc pas la mettre dans la méthode `continuousStreaming` car elle bloquerait tout envoi de messages tant que nous n'avons pas de réponse. Pour pallier à cela, nous allons placer la lecture de réponse sur son propre thread d'exécution. Ainsi, l'envoie et la lecture est totalement dissociée et ne peut pas bloquer l'un ou l'autre. Ce thread d'exécution est en revanche initialisé et lancé au sein de la méthode `continuousStreaming`, car nous ne voulons la mettre en place qu'une fois la connexion correctement établie. Dans le cas de l'ouverture de la connexion via `openConnexion`, comme nous n'attendons qu'une réponse et que nous ne pouvons pas avancer sans elle, nous pouvons l'appeler sur le même thread. Nous avons donc créé une classe interne à la classe Client nommée `ReceiveThread` qui va donc exécuter, tant que la connexion est ouverte, la méthode `readLine`.Pour chaque message reçu, nous le séparons dans un tableau de String par rapport aux espaces pour déterminer le premier mot présent. Nous aurions pu néanmoins passer par la méthode `startWith`, mais il aurait quand même fallu parser de cette manière le message pour traiter son contenu. Ce message parser par rapport aux espaces est ensuite traité par la méthode `messageTreatment`. Cette dernière va donc, en fonction du préfixe présent (list, NOK, bye, etc), lui assigner le traitement correspondant. Dans le cas où l'on recevrait une list de poissons, nous parsons la liste pour créer un tableau de Poisson. Nous la passons par la suite à l'aquarium, qui va s'occuper de mettre à jour les poissons présents. Nous détaillons cela à la section 4.2.2. Néanmoins, nous démontrons le fonctionnement par l'intermédiaire du diagramme de séquence à la figure 5

Dans le cas de la fermeture de connexion, ordonnée soit par le client, soit par le serveur, l'attribut *connected* est passé à faux, stoppant le blocage par la boucle while dans la fonction `continuousStreaming`. La méthode `closeConnexion` est alors appelée, fermant le socket, mais aussi les différents buffer de lecture/écriture ainsi que les différents threads (celui pour lire le terminal et les réponses du serveur).

Chaque transaction est stockée dans un fichier log. Cela est réalisé par la classe Log. Elle s'occupe uniquement de créer un fichier de log et d'écrire à l'intérieur. Ce dernier est fermée à la fermeture de la connexion avec le serveur. On ne peut donc pas le consulter pendant l'exécution.

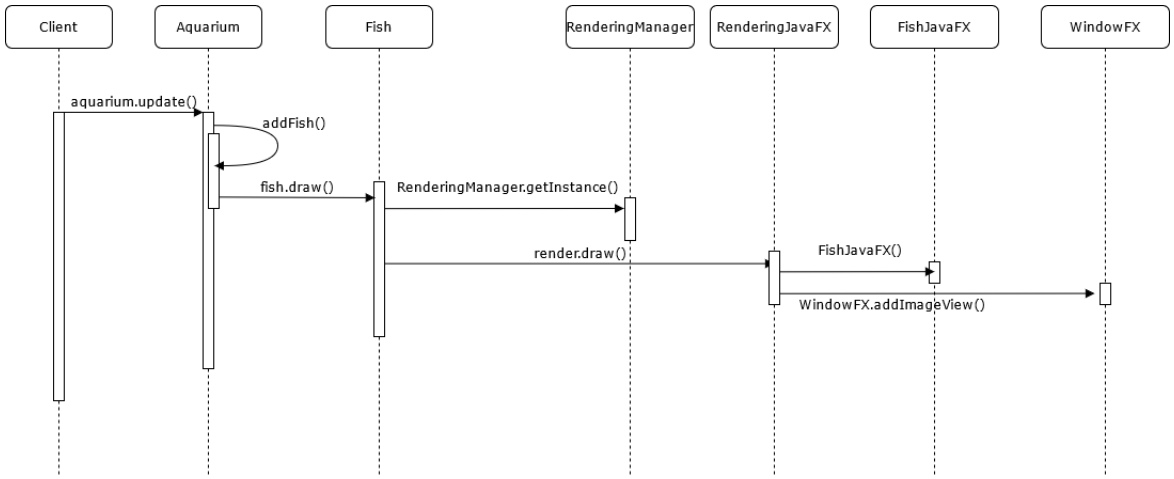

FIGURE 5 – Diagramme de séquence montrant l'ajout d'un nouveau poisson dans la view

4.2.2 Module modèle

Il concerne l'ensemble des données concernant l'aquarium, les poissons qu'il contient (ceux qu'il doit afficher) et leur position. L'Aquarium est représenté sous la forme d'un Singleton. En effet, il est unique et nous devons avoir accés à la même instance à la fois dans la partie cliente et dans la partie graphique. L'aquarium est créé au moment de l'exécution du programme. L'ajout de nouveau poisson se fait au sein de la méthode `updateAquarium`. Cette dernière reçoit la liste des poissons renvoyés par le serveur. Elle va comparer alors chaque nom de poisson qu'elle stocke avec celui de la liste donnée en paramètre. Si le poisson est déjà présent, elle va mettre à jour sa destination et le temps d'exécution. Dans le cas où il serait nouveau, elle va l'ajouter. L'ajout d'un nouveau poisson se traduit par son affichage. En effet, `updateAquarium` va appeler la méthode `addFish`. Cette dernière va alors l'ajouter dans la liste des poissons à gérer au sein de la view, et va aussi faire appeler au Rendering qui va dessiner le poisson. Nous expliquerons plus en détail le fonctionnement du rendering dans la section 4.2.3. Néanmoins, nous avons voulu au sein de l'application client, dissocier le modèle et sa représentation graphique pour permettre une meilleure extensibilité et une modification facile de l'affichage. Dans le cas où un poisson serait à supprimer, cela signifie qu'il est présent dans notre liste de poissons, mais dans pas celle envoyée par le serveur. Nous les comparons alors dans l'autre sens (Dans le cas d'un ajout, nous faisions soit les poissons nouveaux, soit les poissons dans l'aquarium), et s'il n'est pas présent dans la liste envoyée, nous le supprimons. Le supprimer le retire de notre liste, mais aussi de l'affichage en faisant une nouvelle fois appelle au Rendering. L'ensemble des classes correspondantes au model sont présentes dans le package model.

4.2.3 Module affichage graphique

Ce module concerne les classes du package rendering ainsi que celle dans view (WindowJavaFX). Afin d'éviter toute dépendance graphique à une bibliothèque en particulier, nous avons mis en place un Bridge par l'intermédiaire du package rendering. En effet, nous n'avons accés à la partie graphique que par l'intermédiaire du RenderManager, qui est Singleton qui stocke une seule instance de type IRendering. Ainsi, nous passons par le RenderManager pour obtenir la classe qui s'occupera de l'affichage graphique. Il suffit alors juste d'ajouter une nouvelle classe qui implémente l'interface et changer l'instance stockée dans le RenderingManager (actuellement forcément RenderingJavaFX car il n'y a que cette bibliothèque utilisée). Nous permettons alors une extensabilité tout à fait transparente dans le client au niveau de l'affichage graphique.

Pour représenter un poisson au sein de l'affichage, nous avons décidé de lui créer une classe qui

serait sa représentation graphique. Celle-ci est dénommée `FishJavaFX`. Cette dernière va alors stocker l'ensemble des informations utiles à la fois pour représenter le poisson sur l'écran, mais aussi pour faire le lien avec le poisson dans le model. Ainsi, cette dernière contient principalement une image qui est la représentation graphique du poisson et dessiner sur l'affichage, le chemin qu'elle doit parcourir, son temps d'exécution, sa position courante et son nom. Comme chaque poisson à son propre nom, ce dernier peut alors servir d'identifiant. Ainsi, pour faire correspondre une représentation graphique et le poisson présent dans l'aquarium, nous comparons les noms. Nous stockons aussi la position courante dans la vue en récupérant la position courante qui est donnée lors de l'ajout d'un poisson dans la vue. Une fois le poisson ajouté dans la vue, la position courante est mise à jour à chaque fin de destination. Cette position n'est utilisée que pour l'affichage graphique et n'intervient à aucun moment dans le model. En effet, nous avons besoin de cette position pour dessiner la première fois le poisson au bon endroit.

Le déplacement du poisson s'effectue en utilisant les objets de types `Path` et `PathTransition`. `Path` permet de construire un chemin par l'intermédiaire d'un ou plusieurs points. Ceci nous permet alors dans le cas où une destination serait découpée en plusieurs points intermédiaires, de construire une trajectoire passant par tous ces points. Ce chemin est ensuite donné à un objet de type `PathTransition` qui va animer l'imageView et la déplacer en un temps que nous lui passons en paramètre. Néanmoins, ce déplacement se fait via des positions virtuelles. Ainsi, à la fin de chaque destination, nous redessinons l'imageView à la coordonée finale. Si nous ne faisons pas ça, comme l'imageView n'a pas réellement bougée par l'intermédiaire de l'animation, nous avons des décalages et le poisson n'est plus déplacé au bon endroit.

5 Problèmes rencontrés

Durant le développement de ce projet, nous avons dû faire face à plusieurs problèmes, notamment d'implémentation qui seront expliqués dans les deux sous-parties suivantes. Néanmoins, que ça soit côté client ou serveur, nous avons eu énormément de mal dû à la compréhension du sujet. En effet, ce dernier manque énormément de clarté et de précision. Par exemple, il n'est dit nulle part ce que doit renvoyer le serveur à une vue si elle n'a pas de poisson. La gestion des positions est aussi tout à fait hasardeuse et sujette à interprétation. Les corrections apportées par monsieur Ahmed n'étaient présentes nul part sur le sujet, et nous ont forcé à revoir une bonne partie de l'implémentation.

5.1 Côté serveur

Le seul et unique problème se trouve au niveau de l'envoie et de la réception des commandes du client par le serveur. En effet il nous a fallu du temps pour comprendre que les commandes du client pouvaient être concaténées dans un seul buffer que nous recevions voire commencer par le caractère « _n ». Cela nous à coûté pas mal d'heures de debug.

5.2 Côté client

— Durant le développement, nous avons connu quelques difficultés d'implémentation sur diverses zones de l'application. Nous n'avons pas l'habitude d'utiliser les threads au sein d'une application, et se voir créer autant de thread pour éviter les différents blocages nous a un peu surpris au départ. Par exemple, pour éviter le blocage de l'appel à la méthode`readLine`, nous n'avions pas de suite pensé à un thread, car la méthode `isReady` pour un buffer en java pouvait nous dire si celui-ci était prêt à être lui ou non (même s'il s'avère au final que cette dernière est peu utile).

— Nous nous sommes attachés à développer une application la plus modulaire et extensible possible, même si ce n'était pas forcément le but principal de ce projet. Mettre en place une architecture la plus propre possible nous a forcé à revoir le code à plusieurs reprises, en plus des modifications suite aux mauvaises compréhensions du sujet.

— Enfin, nous avons rencontré la plus grosse difficulté au sein de JavaFX. Nous avons mis énormément de temps à comprendre le système de coordonnées utilisées et les différentes manières de faire bouger l'image. En effet, comme les possibilités sont diverses, nous ne savions pas quoi choisir et nous avons dû les expérimenter toutes, avec plus ou moins de réussite. Qui plus est,

il arrivait que la bibliothèque rentre en conflit avec des instructions de notre OS qui nous amenaient à des erreurs que nous n'avions pas sur l'ordinateur d'un autre membre du groupe. Ainsi, à plusieurs reprises, nous avons cru à un bug qui n'en était pas un. La majorité du développement côté client s'est focalisé sur la partie graphique, chose que nous n'avions pas anticipée au début. Qui plus est, malgré la réussite de l'affichage graphique, nous n'avons pas réussi, lorsque le client est déconnecté, à fermer proprement l'affichage du client.

6 Conclusion

Nous avons pu, par l'intermédiaire de ce projet, découvrir la programmation réseau. Nous avons pu concevoir à la fois un serveur et les diverses fonctionnalités qui lui sont liés, mais aussi un client et son affichage graphique. Nous nous sommes alors confrontés aux différents problèmes et difficultés qu'il peut y avoir dans le développement d'un module réseau. Nous avons pu aussi travailler notre organisation du travail en séparant le projet en deux sous-modules implémentés par deux équipes de développeurs. De plus, le côté visuel du projet lui donnait un aspect concret et authentique qui représentait une source de motivation pour essayer d'arriver à des résultats intéressants. Même si le rendu est fonctionnel, il est encore possible de l'améliorer notamment au niveau de l'affichage graphique.