Filière informatique
ENSEIRB-MATMECA

Projet de Réseau (RE203)
Rapport final

BERNARD Hugo BOUDALI Mohammed
EL HAMMADI Mouhcine IZEKKI Jalal ZGHARI Hicham

Table des matières

1 Introduction

L'objectif du projet est de réaliser un programme permettant de simuler un aquarium centralisé de poissons. Le programme se décompose en deux parties, une jouant le rôle du contrôleur et l'autre du client. Les deux parties communiquent par le biais d'un réseau client/serveur. Ce programme suit une architecture logicielle de la forme modèle-vue-contrôleur. Le but du projet est de permettre à plusieurs clients de se connecter au serveur, lancer un affichage et communiquer avec le serveur pour obtenir les positions d'un ensemble de poissons en temps réel , puis les afficher dans un afficheur. Ces poissons peuvent se déplacer d'une vue à l'autre, c'est à dire entre chaque client. Ce rapport présente notre réalisation du projet, certaines pistes améliorations possibles ainsi que quelques notes sur notre méthode d'organisation.

2 Programme contrôleur

Le contrôleur a un rôle de serveur. Il permet d'initialiser un modèle, ici un aquarium, puis d'envoyer des informations sur cet aquarium aux différents clients connectés, à leur demande ou non. Indépendemment des clients connectés, il gère toute la vie de l'aquarium, en calculant et en actualisant toutes les données nécessaires, en particulier la position de chaque poisson. Dans cette section, nous décririons le fonctionnement de ce contrôleur.

2.1 Structure

Nous pouvons séparer notre implémentation en deux partie : une partie purement contrôleur, qui gère les interactions serveur-client et serveur-utilisateur, et une partie modèle, qui définis comment manipuler des objets tels que les poissons et aquariums.

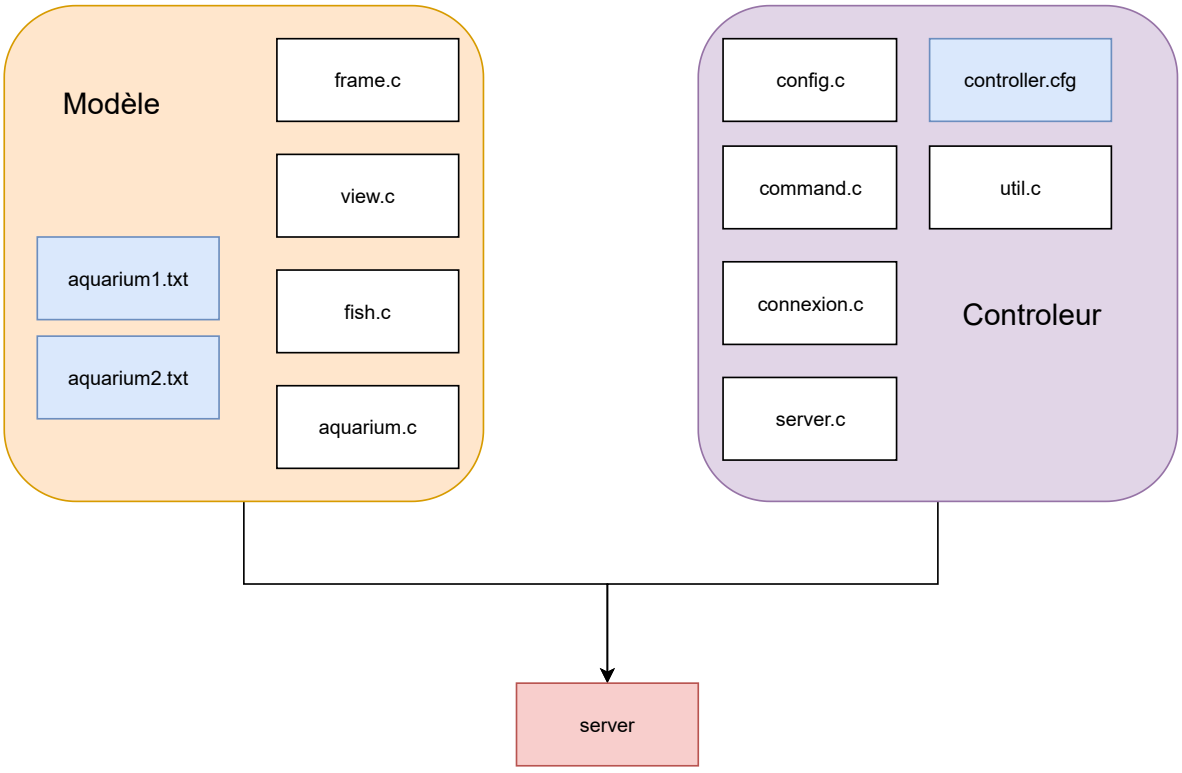


Figure 1: Organisation du code du contrôleur

La figure 1 présente les différents fichiers nécessaires au fonctionnement du serveur. Les fichiers C

sont compilés à l'aide de GCC en un exécutable `server`. Voici comment nous avons organisé notre code.

- **Coté modèle** :
  - Le fichier `frame` définit une structure `frame` qui représente un rectangle de position (x, y) et de taille (width, height). Nous pouvons alors grâce à des méthodes associées transformer une chaîne de caractère de la forme `x*y+a+b`, présente dans les descriptions d'aquarium sous ce format protocolaire (dans les fichiers `aquarium<i>.txt` par exemple), en une structure simple à utiliser en C. Cette structure est la base permettant de définir sans duplication de code nos vues, poissons et aquariums. Notons que la position (x,y) d'une frame correspond à la position dans un repaire du coin haut gauche du rectangle.
  - Le fichier `view` définit une structure view représentant une partie de l'aquarium. Une view est composé d'une frame, ce qui lui donne donc une taille et une position, ainsi que d'un nom. Afin de simplifier le parcours et les manipulation d'ajout/ suppression des view, nous avons choisi de lier ces view avec une liste chaînée. Chaque view possède donc un champs `next` représentant la view suivante dans la chaine. Nous utilisons les macros STAILQ de la bibliothèque `queue.h` pour manipuler nos listes chaînées.
  - Le fichier `fish` définit une structure `fish` représentant un poisson. Un poisson est très proche d'une `view` dans sa structure : il est composé également d'une frame, d'un nom et d'un champs `next` permettant de chainer des poissons. Il possède également un champs `is_started` permettant de savoir si le poisson doit être en mouvement ou non.
  - Le fichier `aquarium` définit un aquarium. Un aquarium est une frame contenant une liste de poissons et une liste de vues. La grande majorités des fonctions utilisées par le contrôleur pour manipuler l'aquarium se trouvent dans ce fichier.

- **Coté contrôleur** :
  - Le fichier `config` permet de lire le fichier de configuration `controller.cfg`.
  - Le fichier `command` contient les fonctions permettant de traiter toutes les demandes des clients et de l'utilisateur. Ces fonctions récupèrent un message sous format texte, le lit et le traite en conséquence. C'est donc là qu'est effectuer la liaison entre le modèle et le contrôleur.
  - Le fichier `connexion` contient la fonction utilisée par les threads gérant chaque client, dont nous détaillons le processus plus bas dans ce rapport. Afin de transmettre les différentes information nécessaires au client lors de la création du thread, nous avons créé une structure `connexion`.
  - Pour finir le fichier `server` contient le main de notre programme. Il gère également l'initialisation du serveur et la connexion des clients.

2.2 Fonctionnement général

Dans cette partie, nous allons décrire le fonctionnement général de notre serveur.

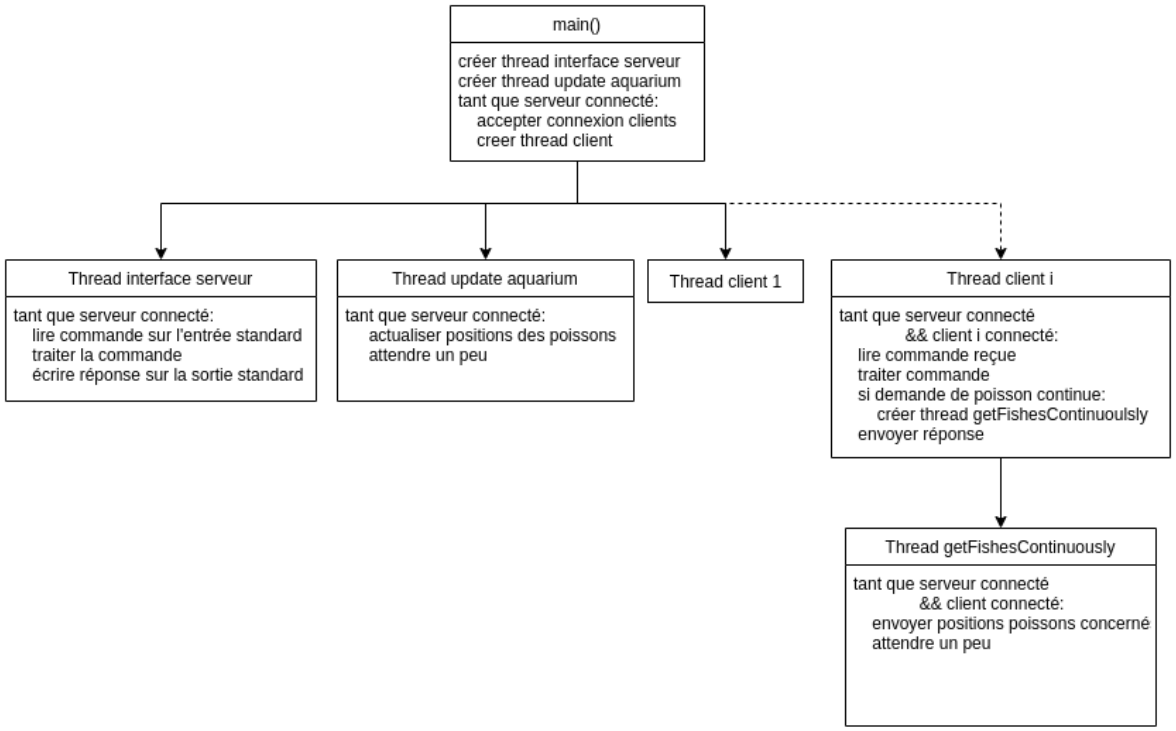


Figure 2: Fonctionnement du contrôleur

Comme le montre le digramme de la figure 2, notre programme commence d'abord par initialiser l'aquarium, puis il crée les threads qui vont gérer les commandes internes au serveur, les commandes issues du protocole et finalement les commandes d'actualisation d'un aquarium. Ces différents threads permettent d'exécuter toutes ces taches en parallèle.

Afin de décrire d'avantage le fonction du contrôleur, nous pouvons découper l'exécution de notre programme en plusieurs étape :

— **Lacement du contrôleur**.
La première tache du contrôleur est de récupérer la bonne configuration (port d'écoute, intervalle de rafraîchissement de l'aquarium, durée du timeout avant déconnexion d'un client inactif).Ensuite, le contrôleur alloue et initialise la structure de l'aquarium et finalement ouvre le serveur.

— **Ouverture du serveur**.
Nous utilisons dans le cadre de notre serveur l'API Socket. Au lancement du serveur, nous devons donc afin d'ouvrir la connexion utiliser successivement les méthodes `socket()`, `bind()` et `listen()`. Chaque instruction du serveur est muni d'une sortie d'erreur pour détecter facilement les erreurs de bind ou de création du socket. Nous créons ensuite un thread associé au serveur, qui nous permet d'entrer les commandes internes au serveur dans le terminal puis de traiter ces requêtes, en exécutant une fonction `server__interface()`.

— **Mise à jour de l'aquarium**
Avant la connexion des clients, un thread est créé afin de gérer l'évolution de l'aquarium au cours du temps. Ce thread rafraîchit les positions des poissons contenus dans l'aquarium (initialement vide). En accédant à la structure aquarium, on parcourt la liste des poisson actifs (`is_started == 1`), puis nous incrémentons l'attribut position de chaque poisson à chaque rafraîchissement. Le taux de rafraîchissement est défini dans le fichier de configuration.

— **Connexion des clients**.
Un attribut `statut` permet de savoir si le serveur est connecté ou non. Nous entrons alors dans une boucle qui tourne tant que le serveur est connecté. Pour y connecter un client, nous utilisons la fonction `accept()`, qui attend une connexion puis crée un socket associé au client, l'empile dans la liste des clients connectés, puis crée un thread qui va gérer ce client. L'utilisation des thread était indispensable pour assurer une gestion parallèle et séparée des clients.

— **Gestion des threads client/ échanges de messages**.
Le thread créé par le serveur à l'issue de la connexion d'un nouveau client exécute une fonction `connexion__start()`. Cette fonction consiste en une boucle qui tourne tant que le serveur et le client sont connectés. Elle lit le message envoyé par le client grâce à un `read()` puis appelle la fonction de traitement des messages de type protocole qui enregistre la réponse à envoyer dans un buffer. Cette réponse est alors renvoyée au client par un `write()`.

— **Déconnexion des clients inactifs**.
Nous avons implémenté une fonctionnalité de déconnexion des clients inactifs, i.e n'ayant pas envoyé de message dans les *timeout* secondes (timeout étant défini dans le fichier de configuration). Pour cela, nous utilisons la fonction `select()` dans la boucle d'attente du message, qui permet d'effectuer passivement une attente de *timeout* secondes en attendant un message. Si un message arrive, le minuteur sera réinitialisé en début de boucle. Si aucun message n'est arrivé au bout de *timeout*s, `select()` renvoie 0 et le client est déconnecté. Plus précisemment, dans notre cas la fonction `select()` renvoie au bout de *timeout*s le nombre de descripteurs de fichiers surveillés prêts à effectuer une opération de lecture. Si ce nombre est égal à 0, on en déduit qu'aucun caractère ne peut être lu, et donc que le client n'a pas envoyé de message.

— **Lecture de commandes**.
Lorsqu'une ligne de commande est récupérée, elle doit d'abord parsée, qu'elle soit interne au serveur ou envoyée par un client. Pour cela, la ligne de commande est d'abord séparée en tokens grâce à la fonction `strtok()` de la bibliothèque `string.h`, qui correspondent aux mots séparés par des espaces. Ensuite, en fonction du premier token, qui correspond à la commande, et du nombre de tokens qui indiquent les options de cette commande, la fonction correspondant au traitement de cette commande est appelée.

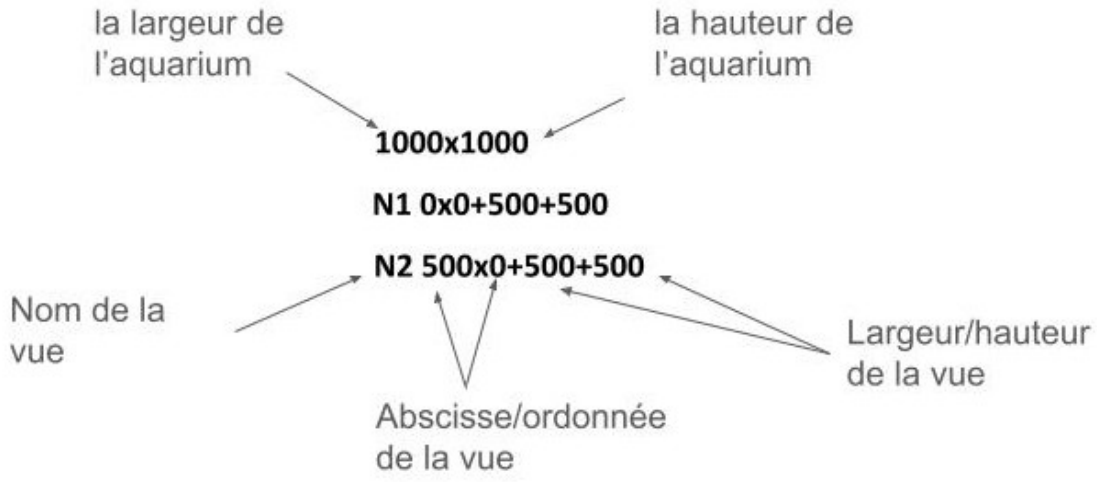
— **Réponse aux commandes**.
Une fois le traitement d'une commande effectué, la réponse est enregistrée dans un buffer. Deux cas de figures se présentent :
□ Commandes protocoles : la réponse est envoyée au socket correspondant grâce à un `write()`.
□ Commandes serveur : la réponse est affichée directement dans le terminal.
Le thread concerné peut alors à nouveau attendre l'arrivée d'une nouvelle commande.

2.3 Traitement des commandes

Les commandes serveur sont les commandes issue au serveur directement à travers la console et pas par socket .

— **load aquarium** : afin de traiter cette commande, le serveur ouvre le fichier qui se trouve dans le dossier data si il existe, sinon la commande retourne **NOK**. Au cas ou le fichier existe il commence par le lire, extrait ses donnés et finalement il stocke ses données dans la structure aquarium.

— **show aquarium** : cette commande affiche les dimensions de l'aquarium et de ses vues sous la forme suivant :



— **add view** : cette commande permet d'ajouter une vue dans l'aquarium actuel sous la forme du schéma précédent.
— **del view** : La commande supprime une vue s'elle existe dans l'aquarium.
— **save aquarium** : Après des modification sur l'aquarium actuel, cette commande permet de sauvegarder ces changement dans le fichier source.

2.4 Tests réalisés

Afin de s'assurer de la robustesse de notre code, nous avons implémenté des tests unitaires. Ces tests testent la globalité des fonctions qui manipulent les entité de l'aquarium (poissons, frames, views).

De plus nous avons, pour tester notre serveur en condition d'utilisation et donc simplifier l'intégration serveur-client, implémenté un faux client. Celui-ci peut se connecter au serveur, lui envoyer et recevoir des messages, qu'il affiche ensuite. Cela permet de tester toutes les fonctionnalités de la communication serveur/client à la main, avec un ou plusieurs clients.

3 Programme du client

Le programme du client permet de créer des instances de clients et de les connecter au serveur afin d'échanger des données. Ce programme doit établir une connexion au démarrage avec le contrôleur. Celui-ci lui communique les informations concernant les poissons existants dans sa vue (nom, coordonnés et taille). Ces données doivent être traitées et passées au programme d'affichage implémenté avec `JavaFX` pour dessiner les poissons dans la vue associée au client.

3.1 Structure

La partie client est implémentée de façon à pouvoir associer à chaque client une vue qui lui est propre. Ceci est assuré avec l'utilisation d'une classe `Connection` ayant plusieurs liens **est-un** dont un avec un `Client` et un autre avec `Viewer` et un lient avec `Prompt` (figure **??**).

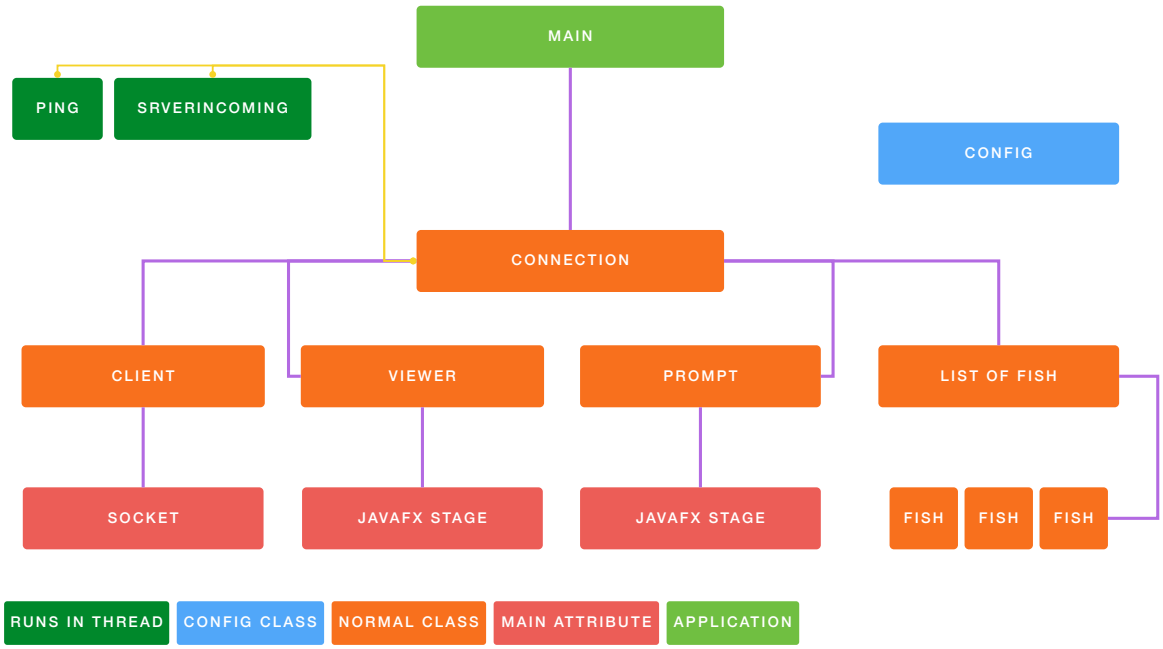


Figure 3: Composition du programme d'affichage

3.1.1 La classe `Client`

Un client définit un socket TCP/IP qui lie le programme d'affichage avec l'aquarium. Un socket est crée à l'instanciation du client avec une adresse et un numéro de port qui sont lus directement du fichier de configuration `affichage.cfg`. Dès que cela est fait, les flux d'entrée et sortie liés au socket sont configurés respectivement avec un `BufferedReader` et un `PrintWriter`.

3.1.2 La classe `Viewer`

Cette classe définit la fenêtre de l'affichage. Á son initialisation, elle définit la largeur et longueur de la fenêtre en lisant ces paramètres du fichier `affichage.cfg` et prépare la fenêtre pour l'affichage en initialisant son titre et son image d'arrière-plan.

3.1.3 La classe `Fish`

Comme son nom l'indique, cette classe représente l'objet poisson, il contient des informations essentielles sur le poisson comme son nom et destination ainsi qu'une méthode `updateFish()` pour mettre ces informations à jour. Le nom d'un poisson est utilisé pour lui attribuer un visuel. Les visuels possibles sont mis dans le dossier `resources/images` et sont `PoissonBleu`, `PoissonClown`, `PoissonCrabe`, `PoissonGlobe`, `PoissonOctopus`, `PoissonOrange`, `PoissonRouge` et `SpongeBob`. Il est bien entendu possible d'étendre cet ensemble de visuels en y ajoutant d'autres images. Le nom d'un poisson doit commencer par le nom dun visuel qu'on souhaite lui attribuer suivi de `_`, puis une chaîne de caractères qui permettra de le distinguer du reste des poissons ayant le même visuel.

3.1.4 La classe `Prompt`

Cette classe permet de créer un prompt qui permettra à l'utilisateur de communiquer avec l'aquarium à travers une interface graphique simplifiée. Cette interface contient des boutons pour effectuer lancer rapidement certaines commandes comme `status` et `log out` ainsi que d'un bouton permettant de charger et lancer un ensemble de commandes qui servent à tester rapidement la communication avec le contrôleur en faisant appel à plusieurs commandes comme `addFish`, `delFish` et `startFish`. Le prompt permet également à l'utilisateur d'entrer ses propres commandes à la main et de visualiser les réponses du contrôleur.

3.1.5 La classe `Connection`

C'est la classe qui englobe l'ensemble du programme d'affichage. Á l'instanciation d'une connection, un client est crée, un afficheur (`Viewer`) et un prompt lui sont associés. Plusieurs threads sont ensuite lancés pour permettre de gérer des tâches différentes, notamment un thread pour parser les réponses du serveur et un autre pour pinger régulièrement le contrôleur. Afin de permettre de lancer ces tâches dans des threads différents, plusieurs classes qui implémentent l'interface `Runnable` ont été implémentés.

3.1.6 La classes `Ping`

Cette classe est lancé dans un thread indépendant pour envoyer régulièrement la commande `ping n` au contrôleur, la valeur par défaut pour la durée qui sépare deux pings successifs est lue à partir du fichier `affichage.cfg`.

3.1.7 La classe `ParseServerIncoming`

Cette classe dérive de `Runnable` et est aussi lancé dans un thread indépendant pour effectuer une attente pour la réception et traitement des messages envoyer par le serveur. Cette classe utiliseent certaines méthodes statistiques implémentées dans la classe `Parser` pour traiter chaque réponse reçue d'une manière différente selon le type de la réponse. Par exemple, si la réponse reçue du serveur est `bye`, cette classe termine immédiatement la connexion et ferme le programme d'affichage, alors que si la réponse commence par `list`, la classe effectue un parsing de la liste selon la dernière commande envoyée (`getFishes, getFishesContinuously ou ls`) et met à jour la liste des poissons dans l'afficheur.

3.1.8 La classe `Config`

Cette classe permet de parser le fichier `affichage.cfg` et en extraire l'ensemble des paramètres qu'il définit. Ceci est réalisé à l'aide de la classe `java.util.Properties` qui simplifie la lecture et écriture des fichiers de configuration. Certaines données de ce fichier peuvent être modifiée au cours de l'exécution du programme, notamment la valeur de l'identifiant du client (`id`).

3.2 Fonctionnement

Lors de Démarrage de client , la première étape du programme est d'initialiser la connexion avec les paramètre communiqués, Cette opération va déclencher la création de deux thread ;

- le premier thread crée va servire pour le parsing des commandes reçues. On peut prédire la réponse à l'aide de message envoyé qui peuvent être récupérer via l'instance de l'objet connexion .
- le deuxième thread a pour but de faire constamment des ping afin que le serveur ne déconnecte pas le client pour être passif.

Ensuite , le client envoie une commande "hello" ou "hello as in " selon le fichier de configuration pour récupérer le nom de la vue. Le client également crée des taches pour javaFx pour créer correctement la plateforme et ces paramètre, Récuperer toutes les informations relatives au poisson et leurs chemin.Finalement, à l'aide du module **Image view** et **pathTransition**, alimenté par les paramètres reçus du thread parsing, permet d'avoir une visualisation assez fluide

3.3 commande du client

Les commandes qu'un client peut envoyer sont les suivantes :

— **status** : la réponse du serveur pour ce message est le nombre de poisson dans le vue et une liste détaillant l'état de des poissons. Pour construire cette réponse le contrôleur parcours la liste des poisson inclue dans la vue du client (même partialement) et écrit directement dans le buffer de réponse les information obtenus.

— **addFish** : le serveur commence par vérifier que le nom du poisson existe déjà dans l'aquarium , si c'est bien le cas, il se contente de retourner un message d'erreur au client, sinon il ajoute début de liste le nouveau poisson.

— **delFish** : à l'issue de cette commande, le contrôleur vérifie que le poisson existe déjà dans l'aquarium, si c est le cas il supprime le poisson de la liste sinon il envoie un message d'erreur au client.

— **startFish** : cette commande déclenche principalement le remplissage d'un tableau qui représentera le chemin que le poisson va suivre.

— **log out** : ce commande marque le status de la connexion comme étant déconnecté.

4 Intégration contrôleur - client

4.1 Commandes du protocole

Les commandes protocole sont les commandes de communication client-serveur via une connexion socket .

— **hello in as** : le controleur test d'abord si le client voulant avoir un nom est déjà associé à une vue, si c'est le cas le contrôleur refuse de l'attribuer un nouveau nom. Sinon, on cherche si le nouveau nom est déjà utilisé par un autre client , dan ce cas le contrôleur attribue au client un nom disponible dans la topologie. Si le nom n'est pas déjà pris le serveur peut lui attribuer le nouveau nom.

— **hello** : le comportement de cette commande est pareil à hello as in, sauf c'est le contrôleur qui s'occupe de trouver un nom pour la view.

— **ping** : à l'issu de cette commande, le contrôleur se contente de renvoyer un message "port" avec le nom du port.

— **getFishes** : face à cette commande, le contrôleur parcours toutes les les poissons dans l'aquarium et affiche les informations de ceux qui appartiennent à la vue du client en utilisant des cordonnées relatives à la vue .

— **ls** : le but de serveur face à cette commande est de retourner les 5 prochaine position qui sont inclus dans la view.

— **getfishescontinuously** : cette commande reste la plus importante dans notre projet. Afin de traiter cette commande, le contrôleur commence par créer un nouveau thread, le rôle de ce thread est de parcourir la liste des poissons et d'envoyer les informations des poissons qui vérifie les conditions suivantes :

  ☐ le client est connecté.

  ☐ le poisson est mode actif (is_started == 1).

  ☐ le poisson est inclue partiellement ou totalement dans la vue.

4.2 Résultat

Voici un exemple d'exécution de notre programme :

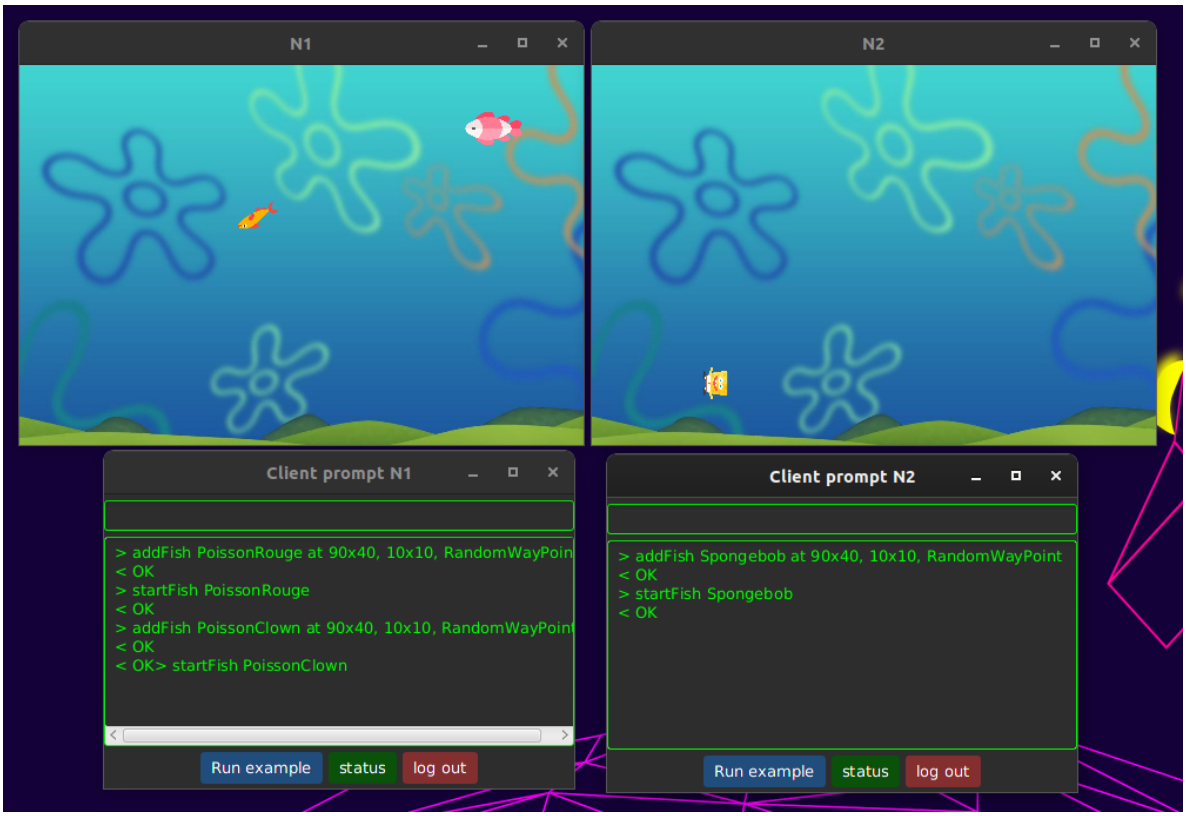


Figure 4: Exemple d'affichage

La figure 4 présente une capture d'écran de notre application en marche, coté client. Les deux vues sont positionnées de sorte à représenter l'aquarium au complet.

4.3 Gestion des logs

Le contrôleur permet la gestion des logs que ce soit ceux engendrés par le serveur ou par la partie du client. En effet, l'affichage des logs du serveur est fait grâce à la fonction `DEBUG_OUT(...)` qui sont dirigés à la sortie d'erreurs, or on redirige la sortie vers un fichier ou les logs sont ainsi sauvegardés comme vu lors de son inspection dans la figure qui suit.

```
#define DEBUG_OUT(...)
    do
    {
        fprintf(stderr, "DEBUG: "__VA_ARGS__);
    } while (0)
```

```
DEBUG: generated position is : 950x400,50x100
DEBUG: generated position is : 950x400,50x100
DEBUG: f->position = 0
DEBUG: PoissonRouge in N1 ?
DEBUG: f->position = 1
DEBUG: PoissonRouge in N1 ?
DEBUG: connection ended, proceeding to free
DEBUG: connection freed, thread will now exit
```

Pour la partie contrôleur responsable des interactions avec le client, l'ajout d'un attribut de type `Logger` à la classe `Connection` est fait ce qui nous permettra de bien tracer et afficher les logs chacun selon son type (SEVERE, FINE, FINER, INFO) sur la console comme suit :

```
INFO: > startFish PoissonRouge
mai 23, 2021 4:28:51 PM aquarium.parse.ParseServerIncoming run
INFO: < OK
mai 23, 2021 4:28:55 PM aquarium.parse.ParseServerIncoming run
INFO: < list  [PoissonRouge at 96x40,10x10,5]
mai 23, 2021 4:29:00 PM aquarium.parse.ParseServerIncoming run
INFO: < list
mai 23, 2021 4:29:03 PM aquarium.Connection sendCommand
INFO: > log out
mai 23, 2021 4:29:03 PM aquarium.parse.ParseServerIncoming run
INFO: < bye
mai 23, 2021 4:29:03 PM aquarium.Connection sendCommand
INFO: > log out
```

5 Pistes d'amélioration possibles

Nous avons réussi à implémenter une grande partie du programme. Cependant, plusieurs améliorations peuvent être envisageables :

- Actuellement, un seul type de déplacement est possible, à savoir un déplacement un déplacement aléatoire. Il est possible d'en ajouter d'autres (verticaux, horizontaux...).
- Le passage des poisons d'un écran à un autre n'est pas synchronisé, il prend un certain temps (1 seconde), donc c'est un point essentiel à améliorer.
- la commande `ls` est partiellement implémentée dans le code mais n'est pas utilisée dans le protocole. On pourrait réfléchir à une façon de l'utiliser.

6 Organisation du travail

Lors du projet, nous avons essayé de garder une méthode agile dans la mesure du possible. Le diagramme de Gantt de la figure 5 montre comment nous avons organisé notre travail au cours du temps. Nous nous sommes de plus séparés en deux équipes de 2 et 3 personnes, une équipe travaillant sur le contrôleur et l'autre équipe sur le client.

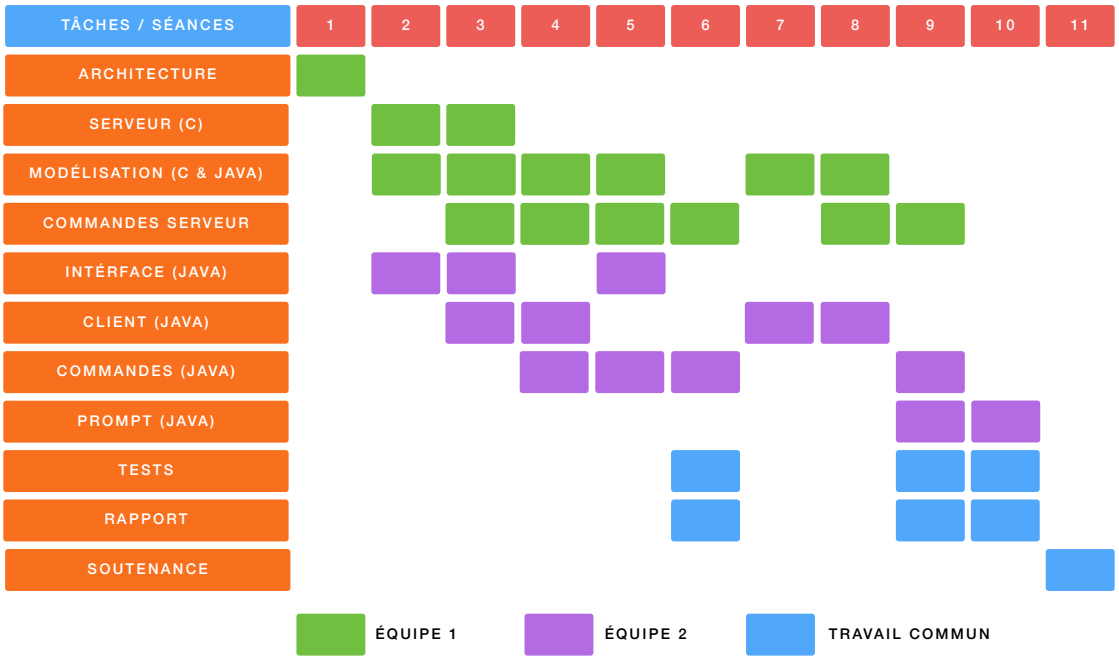


Figure 5: Diagramme de Gantt de notre projet