BORDEAUX INP Enseirb-Matmeca

RE203 : Projet de réseaux

Projet de réseau : Aquarium

Rapport final

Équipe 2 *Responsable :* Toufik AHMED

3 décembre 2024

Table des Matières

1 | Organisation

1.1 Répartition des tâches

Dans le cadre de ce projet, nous avons organisé la répartition des tâches de manière à paralléliser au mieux les fonctionnalités à implémenter pour les 5 membres de l'équipe.

- <1> a été responsable de l'implémentation de l'API socket, du contrôleur, de l'authentification des vues et de la structure « struct set ».
- <2> s'est concentré sur l'implémentation du parseur de fichiers ainsi qu'une partie du serveur.
- <3> a pris en charge l'implémentation de la partie client.
- <4> s'est occupé de l'implémentation de la partie « prompt » .
- <5> s'est chargé d'implémenter des structures de données (Aquarium, Display, Fish) pour le client en C++ et pour le serveur en C.

Cette répartition des tâches nous a permis de progresser efficacement dans le développement du projet tout en répartissant les tâches de manière équitable.

En outre, il a été choisi d'implémenter le client en C++ et non en Java pour mieux correspondre à l'expérience et l'affinité de Nathan et Nolan.

1.2 Gestion des tâches par *issues*

La gestion efficace d'un projet nécessite une organisation méthodique et structurée. Dans le cadre de notre projet, nous avons choisi d'utiliser les issues de GitLab comme outil central pour faciliter notre collaboration et améliorer notre productivité. Les issues sont des fonctionnalités de GitLab qui nous ont permis de suivre, attribuer et résoudre les différentes tâches liées au projet.

Nous avons créé plusieurs issues pour représenter les différentes étapes et objectifs du projet, et avons utilisé les milestones pour regrouper les issues en fonction des jalons ou des grandes étapes du projet. Cela nous a permis d'avoir une vision globale de l'avancement du projet et de prioriser les tâches à accomplir.

1.3 Difficultés d'organisation

Nous avons également été confrontés à certaines difficultés liées à l'organisation du projet. Parmi ces difficultés, le manque de temps et l'incapacité à réaliser certaines tâches en parallèle, notamment la mise en commun des différents éléments, ont été les plus préoccupantes.

2 | Serveur (contrôleur)

2.1 Architecture serveur

2.1.1 Fonctionnement général

Le serveur s'occupant du fonctionnement de l'aquarium et de ses vues possède l'architecture suivante :

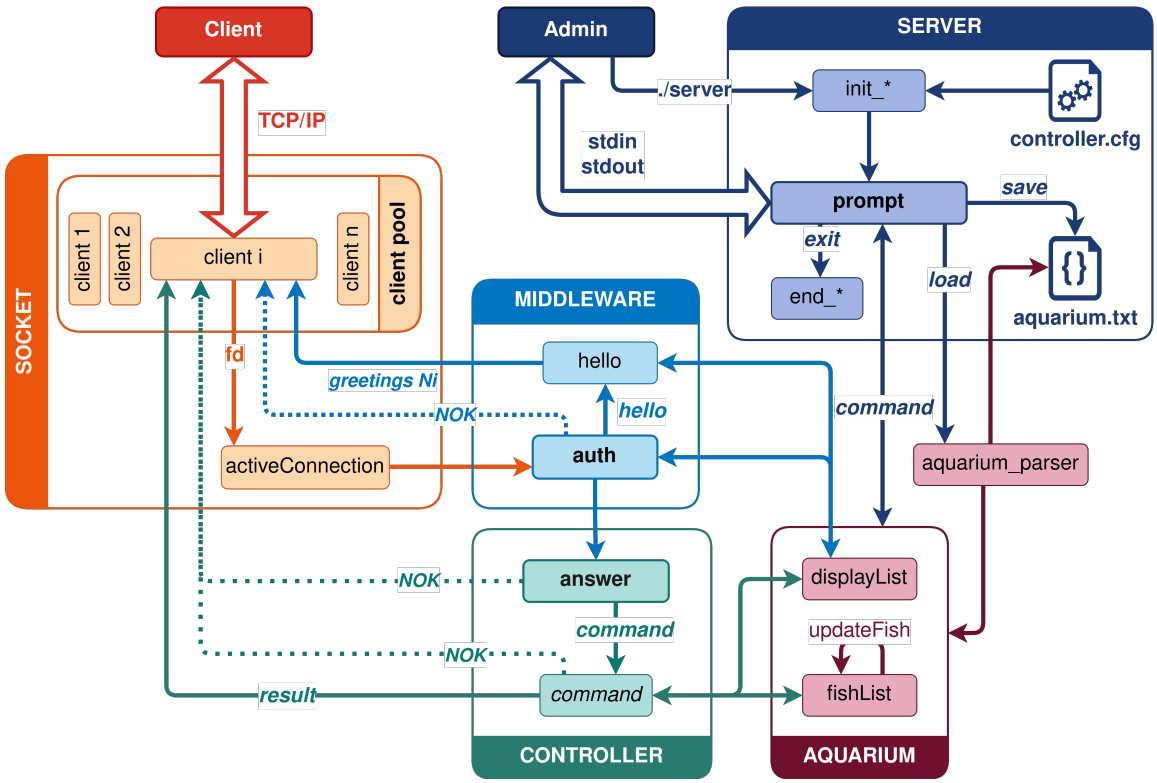

FIGURE 2.1 – Architecture du serveur

Comme décrit dans la figure 2.1, nous avons décomposé notre serveur en différents modules afin d'implémenter les différentes fonctionnalités nécessaires.

Tout d'abord, l'administrateur (personne déployant le serveur) démarre le serveur avec la commande `./serveur`. Ce dernier commence alors une phase d'initialisation du **socket** puis du **contrôleur** à l'aide du fichier de configuration `controller.cfg` avant de proposer un **prompt** à l'administrateur sur l'entrée/sortie standard. Ce dernier lui permet de manipuler l'aquarium en cours d'exécution. La commande `exit` lance la clôture du socket et du contrôleur puis termine l'exécution du serveur. Le socket lancé, les différents clients peuvent se connecter au serveur. À chaque nouvelle connexion, le serveur via la partie *socket* l'accueille dans un pool de clients géré par `epoll` (voir section dédiée à l'API). Lorsque cette connexion est active, on lit le contenu du descripteur de fichier du client puis on l'interprète dans le **middleware** qui vérifie l'authentification de la connexion. Ce module assigne alors ou non un client (son descripteur de fichier) à une vue proposée par le serveur. Si la vue est déjà assignée à une vue, alors la commande du client est passée directement au **contrôleur** qui l'interprétera. En fonction de la validité de celle-ci, le contrôleur effectue les opérations nécessaires sur l'**aquarium** et renverra la réponse demandée au client ou `NOK` si la commande du client est invalide.

2.1.2 Multithreading

La conception du serveur nécessite l'exécution de différents tâches simultanées afin de correctement traiter les différentes opérations demandées, ce qui implique l'utilisation de threads. Plus précisément, nous faisons le choix comme décrit dans la figure 2.2 de dédier le *main* à la gestion du prompt et de multithreader notre contrôleur et notre socket. Ces threads sont lancés au démarrage du serveur à l'aide de fonction `init_*()` et sont terminés proprements grâce à des fonctions `end_*()` lors de la saisie de `exit` par l'administrateur du serveur. Nous avons ainsi :

— Deux threads pour le contrôleur : un pour la mise à jour en continue des poissons et l'autre pour envoyer les poissons en continue aux clients ayant demandé un `getFishesContinuously`.
— $N + 1$ threads pour le socket : un thread dédié aux nouvelles connexions et les N autres pour les connections actives. N peut-être choisi au moment de la compilation est est indépendant du nombre de clients. Dans notre version, nous utilisons $N = 2$ threads.

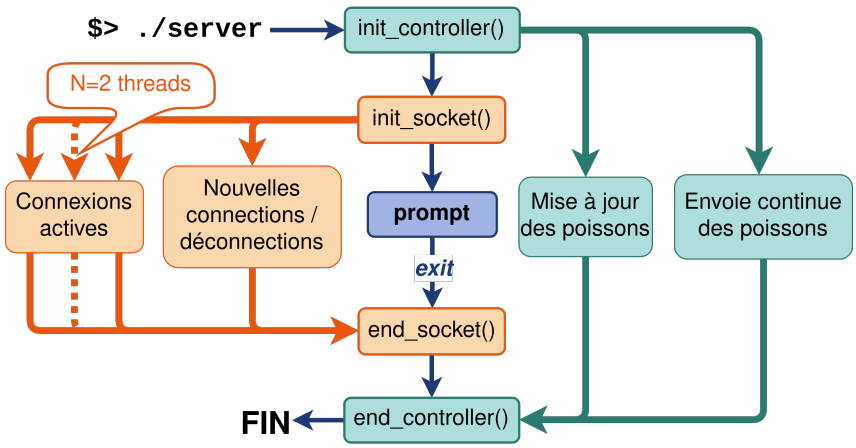

FIGURE 2.2 – Schéma de principe de la création et fermeture des threads serveur

2.2 Outils généraux

2.2.1 Interpréteur générique de commande

Dans le projet, nous avons des commandes spécifiques que l'utilisateur peut nous donner que nous devons être capable d'interpréter afin d'effectuer la commande ou non si celle-ci est incorrecte. Cela doit être fait à plusieurs niveaux dans le programme, puisqu'il est présent aussi bien pour communiquer entre l'utilisateur et le contrôleur pour régler différentes options de l'aquarium mais également pour gérer les messages envoyés entre le client et le serveur. Au lieu de créer des parseurs différents pour chaque niveaux, nous avons décidé d'écrire un interpréteur générique capable d'interpréter les commandes que l'on donne en entrée directement. La figure 2.3 résume le fonctionnement de l'interpréteur.

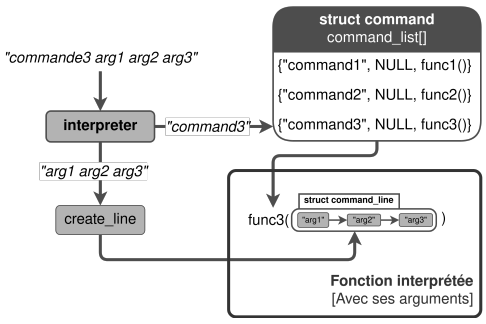

FIGURE 2.3 – Schéma de principe de notre interpréteur de commande

Le parseur `create_line()` prend en entrée la chaîne de caractère à analyser ainsi que le séparateur à utiliser. Ensuite, nous utilisons la fonction `strtok_r` afin d'analyser notre chaîne de caractère en fonction du séparateur défini. Chaque élément ainsi analysé est ensuite mis dans une liste chaîné qui conserve l'ordre de la chaîne de caractère (le premier élément de la liste et le début de la chaîne de caractère).

L'interpréteur complet prend lui en entrée une chaîne de caractère et un séparateur qui sont ensuite envoyé au parseur pour obtenir la liste chaînée, mais aussi qu'un tableau de `struct command` listant des commandes valides. `struct command` contient le nom de la commande sous forme de chaîne de caractère, la fonction associée et une liste d'arguments à appeler. Lorsque la fonction interpréteur a analysé la chaîne de caractère fourni en entrée, elle regarde dans la liste des commandes fourni si un nom de fonction correspond à la commande fourni par l'utilisateur. Si le nom de la commande est trouvé, l'interpréteur renvoie alors la commande sous forme d'un `struct command` contenant la fonction et les arguments interprétés.

Ce module est conçu pour être générique, puisque l'on peut utiliser plusieurs jeu de commandes selon les cas et que l'on peut choisir le caractère séparateur de la commande. Dans notre cas, on utilise trois jeux de `struct command command_list[]` : une pour le middleware, une pour le contrôleur et une pour le prompt.

2.2.2 Ensemble dynamique générique

Une autre outil utilisé à plusieurs reprises par le serveur est une version générique d'un ensemble d'éléments. En effet, nous devons à plusieurs reprises stocker et manipuler plusieurs ensembles de même structure de manière fiable et efficace. Pour cela, nous avons décidé de réutiliser le module *set* implémentée lors du cours d'Atelier de programmation (PG116) dans sa version générique. Ce choix est motivé par trois qualités principales :

— Cette implémentation d'ensemble est suffisamment robuste et contraignante pour garantir l'intégrite des données qu'elle contient tout en étant de taille dynamique. Sa fiabilité est garantie par les nombreux tests effectués lors de sa programmation en PG116 ainsi que par le niveau d'abstraction qu'elle propose.
— Cet ensemble est générique puisque manipulant des types polymorphiques. Nous pouvons ainsi créer des ensembles différents en utilisant le même module.
— Nous maîtrisons finement la complexité de *set*, son implémentation et comportement étant connue et testée.

Dans notre cas, nous avons besoin de 3 ensembles différents pour le serveur : deux ensembles pour l'aquarium afin de stocker dynamiquement les vues disponibles ainsi que les poissons créés dans l'aquarium et l'ensemble des descripteurs de fichiers (clients) ayant demandé des poissons en continue via la commande `getFishesContinuously`.

2.3 Construction des logs

Enfin, il était nécessaire de créer un système de génération de logs afin de suivre en temps réel l'activité de notre serveur. Pour cela, le serveur construit le fichier `server.log` contenant ses logs d'activité avec le module `logger.c` qui permet avec `log_messages` d'avoir quatre niveau de criticité différents (`Critic`, `Warning`, `Success` et `Info`), d'avoir à chaque enregistrement son instant d'émission et l'écriture de messages paramétrés pour chaque enregistrement grâce aux va_list. Les appels systèmes peuvent être gérés par la fonction `exit_if` permettant de renvoyer et enregistrer une erreur critique si l'appel échoue.

2.4 API Socket serveur

Le fichier `socket.c` implémente l'API Socket permettant aux clients d'interagir via TCP/IP avec le serveur. Afin d'optimiser la gestion des connexions, nous avons choisi une parallélisation des opérations en deux types de threads :

— Un thread `connections_handler` dédié à la gestion des nouvelles connexions et des déconnexions. Celui-ci est chargé d'insérer les nouvelles connexions dans le pool de descripteurs de fichiers gérés par le gestionnaire d'événements `epoll`
— Un nombre arbitraire (2 dans notre cas) de threads `active_connection` dédiés à la gestion des connexions actives. À chaque détection d'une activité dans le pool, un de ces threads redirige la lecture du descripteur de fichier concerné vers le middleware `authentificate()` pour prendre en charge le client.

`epoll` utilise un modèle asynchrone qui permet de traiter les événements qu'il gère uniquement lorsqu'ils se produisent, dans notre cas lorsqu'une connexion est active ou qu'il y a une nouvelle connexion/déconnexion. `epoll` peut ainsi gérer un grand nombre de avec une complexité grandement réduite car dépendant uniquement des descripteurs actifs. Le middleware et le contrôleur utilisent ces descripteurs de fichiers gérés par `epoll` pour interpréter les commandes des clients.

2.5 Middleware et Contrôleur

2.5.1 Authentification des clients

Le descripteur de fichier d'une connexion active est donnée par le socket au middleware qui est chargé d'authentifier le client. Il y a alors plusieurs possibilités :

— Soit le client est déjà associé à une vue et alors on redirige son descripteur de fichier au contrôleur
— Soit le client est inconnu. S'il envoie la commande « hello » ou « hello in as <vue> », alors on tente de lui donner une vue libre ou celle qu'il demande si nécessaire. Le descripteur de fichier est alors enregistré dans la vue en question présente dans la liste des vues de l'aquarium.
— Si c'est impossible, si le client a expiré ou que la commande est invalide, alors on écrit `NOK` sur le descripteur de fichier.

L'expiration d'un client est effectuée de manière asynchrone : on enregistre dans la vue du client l'instant du dernier échange valide et on vérifie à chaque nouvelle requête si elle est plus récente qu'un temps d'expiration définir dans la configuration du serveur. Un client qui a expiré se verra donc refusé. Une vue est alors définit comme libre lorsqu'elle n'a pas de descripteur de fichiers ou que sa date de dernière activité est supérieure à la limite configurée.

2.5.2 Contrôleur

La gestion des poissons et des vues en temps réel et l'interprétation des commandes des clients se fait par le contrôleur. Le contrôleur est un programme auxiliaire à l'aquarium qui permet la mise à jour des différents éléments qui le compose en prenant en compte les interactions avec le client. Comme plusieurs instructions sont à exécuter en même temps, nous avons notamment ajouté des threads pour permettre une exécution parallèle des tâches. Nous avons au total deux threads : un permettant la mise à jour des positions des poissons, et un permettant l'envoi régulier de messages (personnalisés) aux clients, tels que `getFishesContinuously`.

La fonction principal du contrôleur est `answer`, qui permet de répondre à chaque client après interprétation de sa demande par le module dédié (cf. sous-section 2.2.1) . Plusieurs autres fonctions sont ensuite appelées, afin de traiter chaque arguments donné et ainsi répondre plus finement à la requête. Pusieurs interactions clients-serveur sont salors possibles. En particulier, un client peut ajouter un poisson dans sa vue, avec le mode de déplacement qu'il souhaite. Cependant, nous avons fait en sorte que deux poissons ne puissent pas avoir le même nom, quelque soit la vue. Cela permet entre autre au serveur de ne pas confondre deux poissons quand ceux-ci se déplacent en dehors de la vue du client qui l'a créé. Un client peut aussi supprimer un poisson qui est dans sa vue, même s'il n'est pas le créateur de ce poisson. Enfin nous avons implémenté les fonctions `getFishes` et `getFishesContinuously` qui

permettent au serveur d'envoyer à chaque client les poissons qui sont dans leur vue (i.e. les poissons que chaque client doit gérer). Nous utilisons pour cela de nouveau la procédure `set__for_each`, qui agira cette fois-ci comme un filtre : la méthode d'affichage des poissons ne sera appliquée que pour les poissons qui sont dans la vue du client.

2.6 Prompt serveur

2.6.1 Interprétation de la ligne de commande

On souhaite que l'utilisateur puisse gérer certain paramètre de l'aquarium sans être disponible du côté client. Pour cela, nous avons une liste de commande à implémenter permettant à l'utilisateur de régler notamment le nombre de vues disponibles dans l'aquarium. Pour faciliter son utilisation, le réglage de l'aquarium doit pouvoir se faire indépendemment des différentes connections effectués par les clients. C'est pourquoi des threads sont utilisés pour la gérer la connexion entre client et serveur.

On utilise `scanf` afin de lire la commande entrée par l'utilisateur puis on utilise l'interpréteur détaillé dans la sous-section 2.2.1 afin de se rediriger vers la fonction permettant d'exécuter la commande souhaiter par l'utilisateur parmi la liste suivante : `load / show / add / del / save / exit`. Pour tout autre commande, un message d'erreur est affiché.

Chacune de ses fonctions prend en entrée l'argument `struct_line` contenant les arguments nécessaire à l'exécution de la fonction.

2.6.2 Exécution des commandes

Voici la liste des différentes commandes disponible pour l'utilisateur côté serveur.

— La fonction `load` prend en argument un nom de fichier ainsi que son extension puis fait appel au parseur de fichier de configuration décrit dans la section 2.9 afin de charger l'aquarium en question. Une fois l'aquarium chargé, un message indique combien de vues ont actuellement disponible dans l'aquarium.
— La fonction `show` affiche simplement dans le terminal la taille de l'aquarium ainsi que les paramètres de ses différentes vues, le tout à l'aide d'une fonction dédié à l'affichage de l'aquarium.
— La fonction `add` permet d'ajouter une vue à l'aquarium, pour cela elle doit respecter une syntaxe bien particulière afin de générer correctement la vue. La syntaxe est la suivante :

  `add view <Nom_de_la_vue> <vue_x>x<vue_y>+<largeur_de_la_vue>+<longueur_de_la_vue>`
— La fonction `del` permet de supprimer une vue de l'aquarium, cela se fait simplement en ajoutant en argument le nom de la vue que l'on souhaite supprimer.
— La fonction `save` permet de sauvegarder l'aquarium afin de conserver les éventuelles modification apportés sur les différentes vues. Pour cela, l'utilisateur doit simplement entrer le nom de son aquarium en argument et n'a pas besoin de préciser l'extension `.txt` étant l'extension utilisé).
— La fonction `exit` permet de quitter l'aquarium, cela met donc également fin à toutes les communications entre le client et le serveur qui est coupé après cette instruction.

2.7 Gestion de l'aquarium

L'une des problématiques principales de ce projet est la gestion d'un aquarium, d'afficheurs et des poissons. Pour cela, nous nous sommes orienté vers de la programmation objet.

Que ce soit du côté du serveur ou du côté du client, la gestion de l'aquarium, des afficheurs et des poissons est identique, seul le langage change : du C++ pour le côté client, du C pour le côté serveur.

Nous avons, dans un premier temps, implémenté le code du client du fait que le C++ est un langage orienté objet, ce qui simplifie l'implémentations des différents objets. Trois classes ont été implémentée :

— La classe *Fish* gérant tout les paramètres liés aux poissons.
— La classe *Display* gérant tout les paramètres liés aux afficheurs.
— La classe *Aquarium* gérant tout les paramètres liés aux aquarium et possèdant un lien « a-un » avec les classes *Fish* et *Display*.

Ainsi pour pouvoir gérer un aquarium du côté client, il suffit d'instancier un objet de classe Aquarium et d'utiliser les méthodes associées à cette classe.

Pour le côté serveur, la gestion de l'aquarium est quasi similaire car nous nous sommes basés sur notre code en C++ pour écrire le code en C.

Cependant, les différences entre le C et le C++ nous ont poussés à faire des modifications dans notre code. Entre autres, toutes nos classes sont gérées par des structures et pour pouvoir gérer les poissons et les afficheurs depuis l'aquarium, nous utilisons une structure permettant d'implémenter des listes chainées.

2.8 Implémentation serveur de l'aquarium en C

Pour implémenter l'aquarium du côté serveur, nous avons mis en place une structure générique qui nous permet de gérer à la fois les poissons et les vues. En effet, notre structure struct aquarium contient plusieurs champs : un nom, sa hauteur et sa largeur, mais aussi les deux structures génériques permettant de représenter la liste des poissons et des vues.

```
typedef struct
{
    char name[256];
    int width;
    int height;
    struct set *fishList;
    struct set *displayList;
    time_t timeout;
} Aquarium;
```

```
struct set
{
    struct link *l;
    int (*cmp)(const void *, const void
    ↪  *);
    void *(*copy)(const void *);
    void (*del)(void *);
};
```

Listing 3 – Structure d'aquarium et structure générique permettant l'implémentation des poissons et des vues.

```
typedef struct
{
  char name[256];
  int x;
  int y;
  int width;
  int height;
  int client_fd;
  time_t last_ping;
} Display;
```

```
typedef struct
{
  char name[256];
  int x;
  int y;
  int vx;
  int vy;
  int width;
  int height;
  int started;
  int delay;
  enum behavior b;
} Fish;
```

Listing 6 – Structure choisie pour les poissons et les vues

Comme nous n'avons besoin que d'un aquarium par serveur, nous avons une variable globale qui n'est autre que l'aquarium. Cela nous permet de simplifier le code et d'éviter plusieurs instanciations d'aquarium qui pourraient être inutiles.

Nous avons ajouté de nombreuses méthodes `getter` et `setter` qui permettent de récupérer et mettre à jour les valeurs des champs de notre aquarium. Ces fonctions de haut niveau permettent de s'abstraire de la structure de données `struct set` et de récupérer facilement un poisson ou une vue en fonction du nom ou du descripteur de fichier entré en paramètre. Nous avons également plusieurs fonctions permettant d'ajouter ou de retirer des éléments dans la liste chaînée, comme par exemple `Aquarium_addFish` qui permet d'ajouter un poisson dans la liste chaînée `fishList`. Enfin, nous avons implémenté plusieurs fonctions qui nous sont très utiles pour la mise à jour de la position des poissons et l'envoi de messages aux clients, comme `Aquarium_isFishInDisplay` qui renvoie un booléen indiquant qu'un poisson est bien dans une vue ou non. Pour cela, nous comparons simplement les coordonnées du poissons à celles de la vue du client. La fonction `Aquarium_maj` met à jour la position de l'ensemble des poissons à l'aide de la procédure `set__for_each` de la structure `struct set`. Cette procédure permet l'appel d'une méthode sur l'ensemble des éléments de la liste chaînée.

2.9 Parsing des fichiers de configuration

Le but de nos parseurs de fichiers de configuration est de pouvoir récupérer les données nécessaires à la création de l'aquarium virtuel de manière fiable. Utiliser des fichiers de configuration permet de simplifier la configuration et la maintenance du système, car toutes les informations de configuration peuvent être stockées dans des fichiers de configuration externes qui peuvent être facilement modifiés sans modifier le code source.

Les parseurs que nous avons codés sont capables de lire des fichiers de configuration avec des options pour l'afficheur et le contrôleur, ainsi que des données pour créer un aquarium virtuel avec des dimensions, des frames (représentant des zones d'affichage). Ainsi, il y a trois fichiers de configuration à lire : un pour le contrôleur, un pour l'afficheur (donc en *C++*), et un autre qui définira la topologie de l'aquarium. Chaque fichier de configuration ayant une structure différente, il nous semblait difficile de faire un parseur générique. Cependant, les fichiers de configuration du contrôleur et de l'afficheur sont au format `cfg` et peuvent donc facilement être interprétés avec la bibliothèque libconfig. Le dernier fichier n'ayant pas de format conventionnel, nous avons codé un parseur spécial pour le lire.

Ainsi, pour les deux premiers fichiers de configuration, nous avons pu utiliser la bibliothèque `libconfig`, car le format des fichiers correspond. Cette bibliothèque C/C++ open-source permet de lire et de manipuler des fichiers de configuration. Nous l'avons utilisé pour simplifier l'implémentation du parseur de fichiers de configuration, car elle offre des fonctions prêtes à l'emploi pour extraire des données de fichiers de configuration de manière fiable et efficace.

Cependant, pour le troisième fichier de configuration, nous avons codé notre propre parseur pour récupérer les données relatives aux frames. Les données sont directement traitées pour créer l'aquarium ainsi que les vues. Nous avons implémenté plusieurs fonctions intermédiaires qui nous permettront de parser le fichier :

- `count_lines` nous permettant de compter le nombre de lignes dans un fichier. Comme une ligne correspond à une vue et qu'une ligne est utilisée pour indiquer la taille de l'aquarium, on sait que le nombre de vues est le nombre de lignes moins un.
- `to_array` qui transforme un fichier en un tableau de chaînes de caractères, où chaque index du tableau correspond au numéro de la ligne dans le fichier.
- `item_from_line` pour parser une ligne en un nouveau tableau de chaînes de caractères. Cette fois-ci, le découpage se fait pour chaque mot. Ainsi, un élément de ce tableau correspond à un mot. On pourra ainsi récupérer le nom de la vue puis convertir les autres éléments en eniters pour avoir son offset et sa taille. Nous utilisons pour cela `strtok` et une liste de délimiteurs : ' ', 'x', '+'.

Enfin, la fonction `parser` parse le fichier et construit l'aquarium ainsi que ses vues en fonction des

données décrites dans le fichier cible.

En plus de la bibliothèque libconfig, nous avons utilisé plusieurs bibliothèques telles que *stdio.h* pour les entrées/sorties, *stdlib.h* pour les allocations de mémoire et *string.h* pour les manipulations de chaînes de caractères.

3 | Client (affichage)

Pour le programme d'affichage nous avons choisi le langage C++ (orienté objet) pour son développement. Le répertoire utilisé est nommé `display`.

3.1 Architecture client

Le programme d'affichage ou client permet d'échanger des données avec un serveur, de les interpréter ainsi que de les restituer à l'utilisateur.

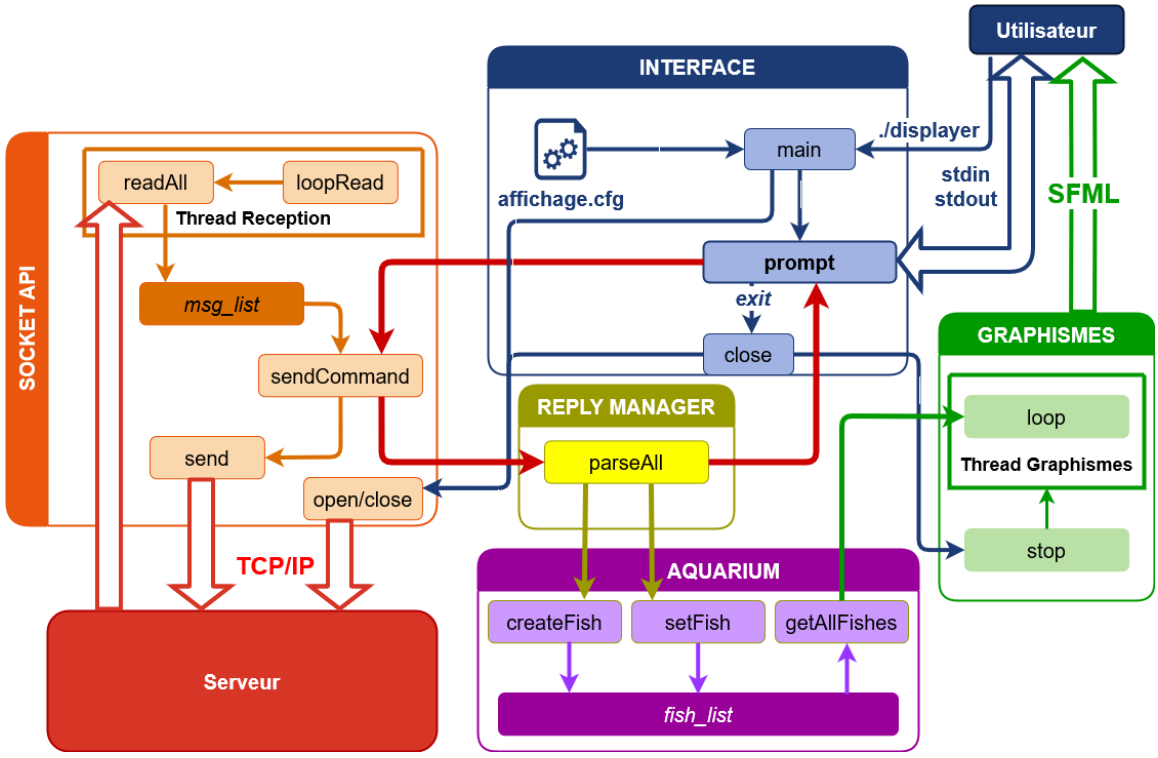

FIGURE 3.1 – Architecture du client

Le schéma figure 3.1 montre l'architecture des modules ainsi que les principaux échanges d'informations du client.

Le programme d'affichage se divise en plusieurs modules ayant chacun un objectif :

— L'interface (`Prompt.hpp`) :
Établie la connexion entre le programme et l'utilisateur.
— L'API Socket client (`SocketAPI.hpp`) :
Permet la communication des données avec le serveur.
— L'aquarium (`Aquarium.hpp`) :
Enregistre et gère les données de l'aquarium (poissons).
— Le gestionnaire de réponses (`ReplyManager.hpp`) :
Interprète les données reçues et agit en conséquence.
— Les graphismes (`Graphics.hpp`) :

Affiche l'aquarium à partir des données en mémoire.

L'implémentation de ces modules est réalisée par l'utilisation des objets C++ et définie dans les fichiers `.cpp` associés.

3.2 Interface utilisateur

La compilation se fait avec la commande `$ make`. Un exécutable nommé `displayer` est ensuite disponible.

L'exécution du programme d'affichage utilisateur est obtenue avec la commande `$ ./displayer <host> <port>` et établie directement une connexion TCP avec les paramètres renseignés.
Dans le cas d'un test du programme isolé, la commande `$ ./displayer debug` permet de lancer le programme en mode hors connexion.
Enfin `$ ./displayer` permet de lancer le programme avec les paramètres lus depuis le fichier de configuration `affichage.cfg`.

Son exécution nécessite la modification de la variable d'environnement `LD_LIBRARY_PATH` pour qu'elle cible le répertoire `SFML/lib`. C'est pourquoi un script `./displayer.sh` prenant les mêmes paramètres que l'exécutable a été mis en place.

Ensuite, l'utilisateur est invité à entrer des commandes via l'entrée standard. Ces commandes sont "parsées" par le programme notamment via le module `Prompt`. Une commande invalide est rejetée, les commandes disponibles sont définies dans le sujet. Une aide est proposée à l'utilisateur ne connaissant pas les commandes.

Le résultat déclenche (ou non) un ou plusieurs appels à des fonctions des autres modules (notamment `L'API Socket client`).

L'utilisateur peut se déconnecter du contrôleur avec la commande `log out`, se qui termine aussi le programme. Il peut aussi le faire brusquement avec des signaux de terminaison.

3.3 Communications avec le contrôleur

3.3.1 API Socket client

`L'API Socket client` est le module s'occupant des communications et de l'interprétation des données reçues. Elle possède quelques méthodes permettant :

- De mettre en place la connexion
- De quitter la connexion
- D'envoyer une chaîne de caractères
- De lire et enregistrer les données reçues
- D'obtenir et consommer les données enregistrées

De plus à chaque fois que le programme échange des données, elles sont copiées et écrites dans un fichier `displayer.log` avec un préfixe (comme dans le sujet) ">" lors de l'envoi et "<" pour une réception.

Les fonctions utilisés pour réaliser la connexion TCP sont celles de l'API Socket en C (pas C++), son utilisation en C++ ne pose pas de problème de compatibilité particulier.

3.3.2 Émission et réception des données

Avec l'API proposée, l'envoi de messages est très simple, la fonction `SocketAPI::send` envoi immédiatement une chaîne de caractères au contrôleur.

C'est en amont dans les modules que les messages à envoyer sont préparés. En effet, les formats des messages envoyés doivent respecter les conventions définies dans le sujet (et L'API n'a pas vocation à s'encombrer pour gérer le formatage).

Pour la réception des messages, il y a quelques difficultés.

Nous avons fait le choix d'utiliser la lecture de données dans les `socket` en mode "bloquant", c'est-à-dire que l'on doit attendre la réception d'au moins un message pour continuer l'exécution.
Cela a l'avantage de laisser du temps au programme pour attendre la réponse du contrôleur pour une commande envoyée.
Par contre, si le serveur envoie au client des messages de manière périodique (comme pour `getFishesContinuously`), alors on peut se retrouver dans des situations problématiques.

Il faudrait par exemple, mettre en place un système de timeout pour éviter d'attendre trop longtemps une réponse.
Mais aussi, il serait judicieux de pourvoir interpréter en temps réel les données reçues.
C'est pourquoi nous utilisons un thread qui s'occupe de lire les données et les les enregistrer en mémoire. On va voir comment cela est géré.

3.3.3 Consommation des messages

Les données reçues sont stockées en mémoire dans l'attribut `msg_list`.
Le thread de réception va donc "boucler" jusqu'à arrêt du programme en attente des messages du serveur.
S'il reçoit une commande à interpréter en temps réel (`list` par exemple), alors le module du gestionnaire de réponse est directement appelé et le message n'est pas enregistré. Sinon les messages restent en attente d'être "consommés".

Le thread principal, une fois qu'il a envoyé une commande, peut attendre une réponse de la part du serveur. Il va donc chercher le contenu des messages reçus pour potentiellement en consommer certains.

Pour éviter les problèmes de concurrence en lecture/écriture des différents threads (que ce soit pour la liste des messages ou les données de l'aquarium), on utilise des structures de `mutex` fournis par la bibliothèque standard (C++).

3.4 Implémentation client de l'aquarium en C++

Afin d'afficher l'ensemble des poissons de l'aquarium, il faut pouvoir les manipuler et les stocker.
C'est pourquoi les deux classes C++ `Fish` et `Aquarium` implémentent efficacement cette objectif.
Ces structures sont assez similaires à celles utilisées pour le serveur.

La différence est que, pour éviter des calculs imprécis dans les mouvements/positions des poissons, on utilise des nombres flottants (que l'on arrondira à l'entier, pour l'affichage) plutôt que des entiers.
En effet, les données envoyées par le contrôleur sont en pourcentage de l'écran, il faut donc les adapter à la position dans l'écran.
De plus, l'utilisation d'entiers provoquent des mouvements imprécis pour certains "angles" de déplacement. Ce constat est nettement visible à l'affichage (le mouvement est saccadé) et il est encore plus important pour l'utilisation des entiers en format "pourcentage".

On retient aussi la position cible du poisson, celle-ci sera actualisée lors de la réception de la commande `list` par le serveur. Elle permet au module graphique d'actualiser la position des poissons.

Ici, l'aquarium fait aussi le rôle "d'interface de virtualisation", et ne renvoie que des copies de poissons pour éviter les collisions issues de la concurrence lecture/écriture des threads. Il s'occupe aussi des conversions entre les formats des positions (flottantes, entières, pourcentage).

3.5 Affichage graphique

Nous avons choisi la bibliothèque SFML pour l'affichage graphique, car c'est une bibliothèque très simple à prendre en main en C++. Mais (attention il y aura un problème de compatibilité sur les machines de l'ENSEIRB qui ont certaines bibliothèques OpenGL et C++ non à jour, il faudra compiler avec `$ make enseirb`).

Pour ne pas gêner le programme principal, tout en affichant l'aquarium, un thread est utilisé pour s'occuper uniquement d'une fonction `Graphics::loop`, qui rafraîchit l'image affichée.

Pour la gestion des graphiques il s'agit pour chaque image, d'afficher correctement les poissons.
Il faut donc donner une image à chaque poisson (elles seront choisies aléatoirement à la création du poisson parmi un ensemble situé dans un répertoire de "ressources").

Ensuite, à chaque nouvelle image (frame), on actualise la position des poissons, puis on les affiche.
Les nouvelles positions sont calculées à partir de plusieurs paramètres dans le but d'être le plus proche possible des informations envoyés par le serveur.
Ces paramètres sont : la position finale, le temps demandé (par le serveur) pour atteindre la cible et la fréquence de rafraîchissement de l'image (FPS).

4 | Utilisation de ChatGPT

ChatGPT ayant été autorisé pour ce projet, nous l'avons utilisé judicieusement sur différentes parties de notre projet.

Parser un fichier étant une tâche difficile en C (du moins, le code peut être assez long), nous avons demandé à ChatGPT des algorithmes de parsing simples ou des bibliothèques qui pourraient aider à notre tâche. C'est alors qu'il nous a conseillé d'utiliser la libraire `libconfig` permettant notamment le parsage de fichier de configuration au format `.cfg`. Nous avons alors utilisé cette librairie et le travail a été plus facile qu'un parsage en partant de zéro, sans librairie. De plus, cette librairie est une librairie C, compatible C++. Ainsi le code de la version C++ du programme de parsing est une transposition du code de parsage de C vers le C++. Malheureusement, le fichier de configuration pour la typologie de l'aquarium n'étant pas au format `.cfg`, nous avons du coder nous même les fonctions de parsage. Cependant, beaucoup de temps a été sauvé grâce à l'utilisation de la librairie `libconfig`, ce qui nous a permis de développer correctement ce second parseur.

ChatGPT a aussi été utilisé lors de l'écriture du socket API multi-client avec l'utilisation d'`epoll`, mais les solutions apportés n'étaient pas forcément satisfaisantes et la documentation en ligne par moteur de recherche était au final plus efficace. En revanche, ChatGPT est très utile pour expliquer rapidement le fonctionnement d'un code, ce que soit pour le comprendre ou pour le présenter dans un rapport, même si cela nécessite souvent une réécriture des résultats pour correspondre aux besoins.