Projet de réseaux
Simulation d'un aquarium de poissons

Équipe G4-2
13 mai 2021

Rémi DEHENNE
Sébastien DELPEUCH
Aymeric FERRON
Tom MOËNNE-LOCCOZ
Aurélien MOINEL

Résumé Ce document présente l'implémentation d'une application d'aquarium en réseau, selon le modèle client-serveur. Le projet est organisé à l'aide d'outils de gestion de projet agile, notamment Taiga. La majorité des objets métier sont implémentés dans le serveur écrit en C, afin de gérer l'ensemble de la modélisation, tandis que le client Java n'implémente que les structures de données qui lui sont utiles pour l'affichage ou l'interaction avec le serveur. Le client et le serveur communiquent via le protocole TCP, en utilisant l'interface *socket* : le client peut ainsi envoyer des commandes automatiques ou saisies par l'utilisateur, que le serveur exécute après s'être assuré de leur validité. Dans cette version, la majorité des commandes et des fonctionnalités demandées sont implémentées, même si les envois périodiques ou continus de poissons ne sont pas fonctionnels. Enfin, même si le serveur gère en parallèle les requêtes de plusieurs clients, le code actuel ne protège pas le serveur des comportements incohérents causés par des accès concurrents.

Introduction

Dans le cadre du projet de réseau de semestre 8 à l'ENSEIRB-MATMECA, nous sommes amenés à implémenter une application d'aquarium selon le modèle client-serveur. D'une part, la gestion de l'aquarium est centralisée au niveau du serveur (ou « contrôleur »), tandis que différentes instances de clients permettent d'interagir avec l'aquarium et d'afficher son contenu de manière graphique.

L'objectif de ce document est de présenter, à mi-parcours, l'état d'avancement de l'application. Nous détaillons d'abord l'organisation adoptée au sein de l'équipe projet, avant de décrire les structures de données choisies pour modéliser l'aquarium et ses composants. Par la suite, nous discutons du fonctionnement général des applications serveur et clientes, en particulier en terme de communications en réseau. Enfin, nous précisons les fonctionnalités restant à implémenter avant la fin du projet.

1 Organisation du projet

Le projet est développé en utilisant certains outils de la méthodologie agile Scrum. Cette méthode permet de développer un produit de manière itérative et incrémentale, afin d'obtenir des retours réguliers du client et des utilisateurs finaux, pour identifier au mieux leurs besoins.

Dans le cadre de ce projet, nous utilisons essentiellement les outils de la méthode Scrum afin d'organiser le développement, et découper les exigences du sujet en *stories*. Nous avons ainsi recours à la plateforme Taiga afin de découper et répartir les tâches au sein du groupe, ainsi que pour suivre leur avancement.

Concrètement, Taiga est une application web *open source* qui permet de gérer un projet agile. En particulier, les parties prenantes d'un projet peuvent créer des *stories* et les ajouter au *backlog* produit. Lors de la planification d'un sprint, les *stories* à réaliser sont déplacées dans le *backlog* de sprint, et découpées en tâches « atomiques ». Chaque développeur peut alors s'attribuer des tâches et modifier leur statut : « Nouveau », « Prêt », « En cours », « À tester » et « Terminé ». Cet outil s'est avéré très commode afin de se coordonner lors du travail à distance.

Par ailleurs, comme le projet est composé de différents modules relativement indépendants, les membres du projet ont été répartis sur chacun des modules. Tom a ainsi implémenté les structures de données du serveur (ou « modèle »), en fournissant au code qui l'utilise une couche d'abstraction via des fonctions d'interface. En parallèle, Aymeric et Sébastien ont mis en place le *parser* et l'interprétation de commandes côté serveur, tandis que Rémi a implémenté l'architecture du serveur, notamment la communication réseau avec les clients connectés. Enfin, Aurélien a développé le client Java, notamment l'interface graphique et la communication avec le serveur.

Néanmoins, cette approche a nécessité un travail de coordination et d'intégration important à la fin du sprint 1. Malgré l'importance de la conception dès le début du projet et la communication constante entre les membres de l'équipe, certains modules ont dû être adaptés pour s'intégrer au reste du code, en particulier côté serveur.

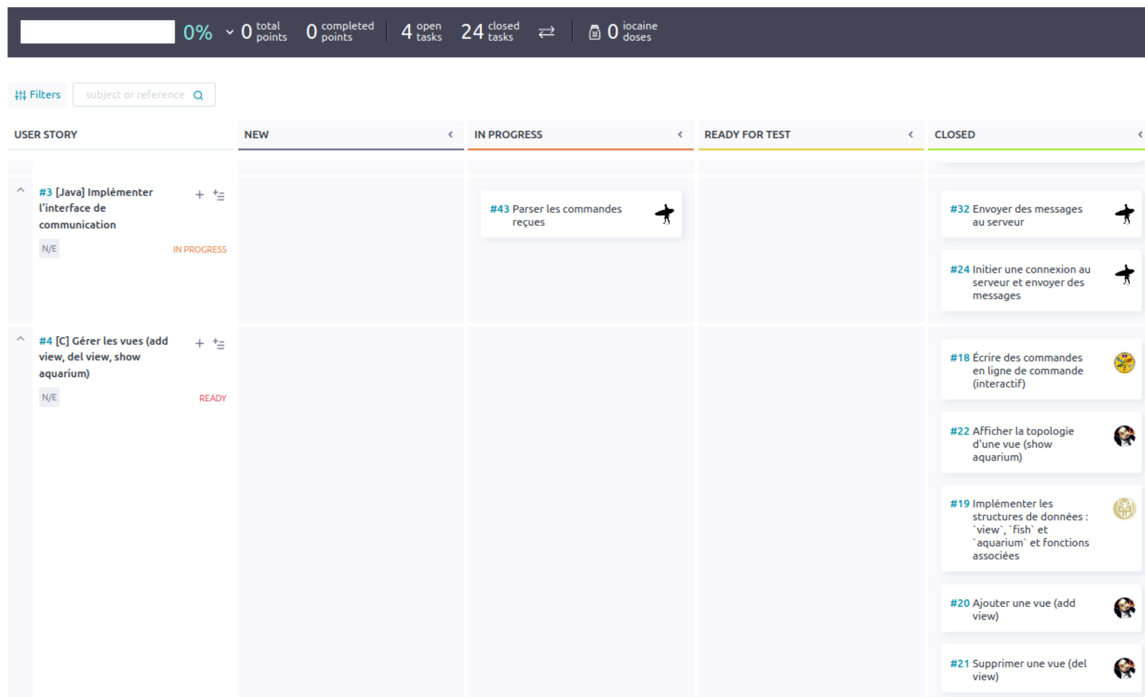

FIGURE 1: Interface d'un backlog de sprint Taiga

2 Modèle et structures de données

Dans un premier temps, nous détaillons les structures de données qui permettent de représenter les objets métier de l'aquarium : ces structures sont regroupées sous l'appellation « modèle » au sens du patron de conception Modèle - Vue - Contrôleur (MVC). En particulier, l'architecture client-serveur et multi-langage du projet impose de développer de manière indépendante les structures utilisées par le client et par le serveur, leurs besoins étant différents.

2.1 Structures de données côté serveur

Du côté du serveur, écrit en C, il est nécessaire de maintenir un ensemble vaste de données de plusieurs natures pour pouvoir fournir des informations cohérentes et correctes aux différents clients.

Pour répondre à ces besoins, des structures représentent les différents objets métier de l'application que le serveur doit manipuler pendant son exécution (poissons, vues, aquarium, etc). Ces structures sont pour la plupart *opaques* pour le code qui les utilise : elles sont seulement *déclarées* dans les fichiers de *header*, et non *définies*. Par conséquent, ces objets ne peuvent être manipulés que par des fonctions d'interface. Cette approche permet d'abstraire les détails d'implémentation et de s'assurer que le code « client » interagit correctement avec les objets métier, notamment à des fins de sécurité et d'indépendance entre les modules.

Tout d'abord, les structures principales sont l'`aquarium_t` pour l'aquarium, les `fish_t` pour les poissons et les `view_t` pour les vues et sont présentées en Figure 2. L'aquarium permet tout d'abord de réunir les différentes vues et poissons, et détient une taille qui sert de référence pour les tailles des poissons et des vues. Ces vues représentent à la fois une région de l'aquarium, de coordonnées et de taille fixées, mais peuvent être également reliées avec une connexion avec un client : en effet, chaque client connecté et identifié par le serveur est rattaché à une vue, qui ne doit être occupée que par au plus un client.

Enfin, les poissons sont les éléments mobiles au sein de l'aquarium. Pour se déplacer, ils disposent de deux structures de données : une mobilité (`mobility_t`) et un modèle de mobilité (`mob_model_t`). D'une part, une mobilité représente le cheminement prédéterminé d'un poisson. Pour le moment, il s'agit simplement de la destination actuelle du poisson, et le temps à l'issue duquel il atteindra cette destination. Cette sous-structure est cependant vouée à être améliorée lors de l'implémentation de commandes permettant au client de connaître périodiquement les destinations des poissons, dont certaines nécessitent de connaître à l'avance les déplacements suivants des poissons.

Les modèles de mobilité `mob_model_t` sont quant à eux des structures permettant aux poissons de mettre à jour leurs informations de mobilité, et, plus précisément, de choisir une nouvelle destination lorsque la précédente est atteinte. Par exemple, elles peuvent implémenter le comportement de *RandomWayPoint* qui décide

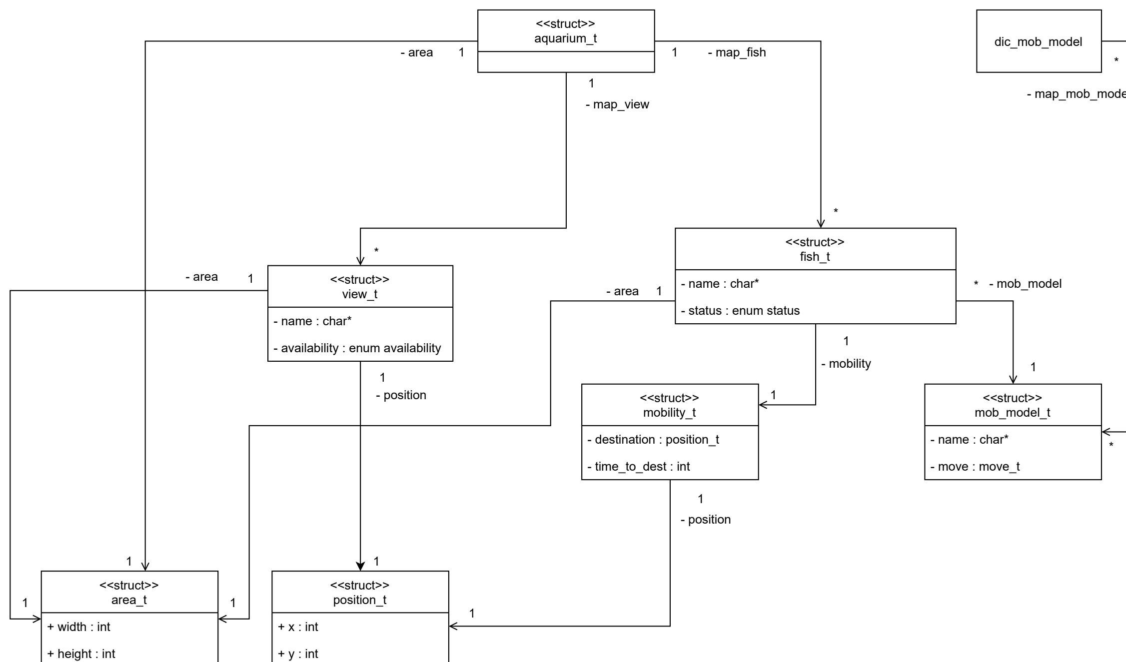

FIGURE 2: Diagramme de classes des structures de données du serveur

aléatoirement d'une nouvelle destination. Cette structure regroupe simplement le nom de modèle du mobilité, en chaîne de caractères, et un pointeur de fonction d'alias de type `move_t`, qui modifie une mobilité `mobility_t` par effet de bord pour changer la destination du poisson. Chaque poisson doit être initialisé avec un tel modèle de mobilité, et ne peut pas en changer durant son cycle de vie.

L'ensemble des modèles de mobilité utilisables est conservé dans une variable globale `dic_mob_model`, qui conserve toutes les stratégies de déplacement. À l'initialisation des poissons par la commande `addFish`, il est donc nécessaire que le nom du modèle de mobilité renseigné par l'utilisateur corresponde à un modèle de mobilité contenu dans `dic_mob_model`. En effet, cette étape permet de retrouver l'adresse de la fonction `move_t` à exécuter par le poisson, à partir de la chaîne de caractères saisie par l'utilisateur.

L'aquarium et le `dic_mob_model` sont implémentés à l'aide de tables de hachage, ou dictionnaires, qui donnent accès en temps constant aux valeurs contenues à partir d'un identifiant choisi. Dans notre cadre, cela permet de pouvoir accéder à tous ces éléments à travers une chaîne de caractères, en l'occurrence leur nom. Cela correspond tout à fait au besoin de la majorité des commandes, et, contrairement à un tableau, permet de trouver l'élément recherché en temps constant, tout en conservant une complexité spatiale toujours linéaire en fonction du nombre d'éléments de la collection.

2.2 Structures de données du client

Le client Java est quant à lui développé selon le patron MVC, adapté pour satisfaire les exigences du projet. Ce patron consiste à séparer dans différents modules les objets métier de l'application (modèle), la gestion de l'affichage (vue) et le traitement des interactions utilisateur (contrôleur).

Contrairement au serveur, les structures de données implémentées dans le client, aussi appelées « modèle », se limitent aux poissons et à certaines de leurs propriétés représentées en Figure 3. En effet, même si d'un point de vue du serveur, chaque client connecté est une vue (`view_t`) de l'aquarium, les clients n'ont besoin de représenter que les poissons qui leur sont envoyés. À l'inverse, là où le serveur ne calcule que certaines positions futures des poissons, le client doit être en mesure d'interpoler le déplacement entre sa position actuelle et la position future, pour que le poisson se déplace progressivement dans l'interface graphique. Dans le cas contraire, les poissons donneraient l'impression à l'utilisateur de se téléporter d'un point a à un point b, ce qui ne serait pas satisfaisant.

La classe `FishProperties` permet de convertir en pixels les positions et la taille du poisson, qui sont reçues du serveur en pourcentage de la taille de la fenêtre. Par ailleurs, la classe `Fish` contient l'ensemble des données liées au visuel des poissons eux-mêmes, ainsi que les informations de déplacement des poissons au sein d'instances de la classe `FishProperties`.

Munis de ces structures de données, nous décrivons dans les sections suivantes le fonctionnement général du serveur et du client.

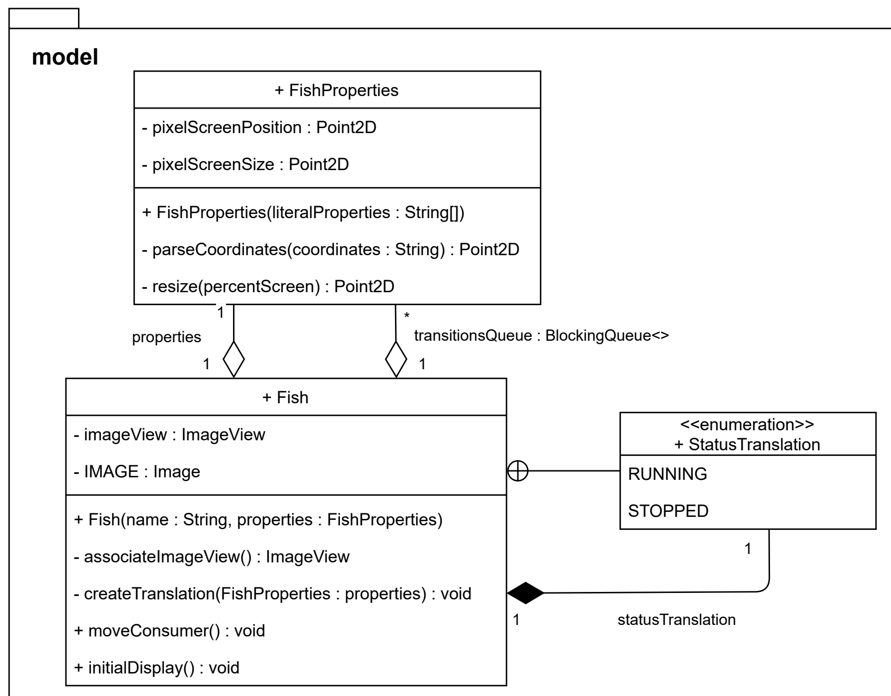

FIGURE 3: Diagramme de classes des structures de données du client

3 Architecture du serveur

Dans le modèle client-serveur, le serveur a pour rôle d'attendre des connexions de clients afin de répondre à leurs requêtes. Dans un tel modèle, la logique métier et les vérifications sont généralement réalisées côté serveur, à des fins de sécurité. En effet, si les vérifications avaient lieu côté client, un client malicieux pourrait alors modifier librement les données du serveur. Le serveur a ainsi pour rôles d'attendre les requêtes des clients, de s'assurer de leur validité, et, le cas échéant, de les traiter, avant de renvoyer une réponse.

3.1 Attente et communication avec les clients

Le serveur implémenté dans le cadre du projet est écrit en C, et communique avec les clients en utilisant le protocole TCP, via l'interface `socket`. Le protocole de couche transport TCP a été retenu, car il assure un transport *fiable*, notamment grâce à des mécanismes d'acquittement et de gestion d'erreurs. En effet, dans cette application, il est impératif que le transport entre machines soit fiable afin d'éviter des incohérences entre les clients et le serveur, même si cela entraîne un surcoût lors des échanges. Par ailleurs, les données n'ont pas de contrainte particulière de « temps réel », contrairement à des applications de téléphonie ou de visioconférence. Dans un tel cas, nous pourrions utiliser le protocole UDP ou le protocole applicatif RTP (*Real-time Transport Protocol*).

Il est généralement souhaitable qu'un serveur puisse gérer plusieurs connexions de clients en parallèle. Après l'initialisation des structures de données et la création de la socket, le serveur se met en écoute passive sur un port donné (port 9009 par défaut), et attend des connexions entrantes via l'appel système `listen`.

Dès qu'un client établit une connexion avec le serveur, cette connexion est ajoutée automatiquement à la file des connexions en attente gérée par `listen`. Le code du serveur se contente de boucler indéfiniment sur un appel système `accept`, qui récupère et supprime la première connexion insérée dans la file. Si la file est vide, l'appel d'`accept` est bloquant jusqu'à ce qu'un client se connecte. Lorsque l'appel à `accept` termine, un descripteur de fichier vers une nouvelle socket est renvoyé : cette socket permet de communiquer avec le client associé à la connexion, par exemple avec les primitives `read` et `write`.

La socket de connexion est alors transmise à un nouveau thread, qui ne communique qu'avec un seul client via la fonction `handle_connection`. Cette solution permet de gérer plusieurs communications en parallèle sur des architectures multi-cœurs, mais également de s'assurer que les appels bloquants à `accept`, `read` ou `write` n'endorment que le thread courant, et ne bloquent pas la communication avec les autres clients. Chaque thread ainsi créé lit le contenu de la socket envoyé par le client, vérifie si le texte reçu est une commande valide et l'exécute le cas échéant : ce processus est répété jusqu'à ce que la connexion soit fermée par le client. La Figure 4 résume le fonctionnement général du serveur.

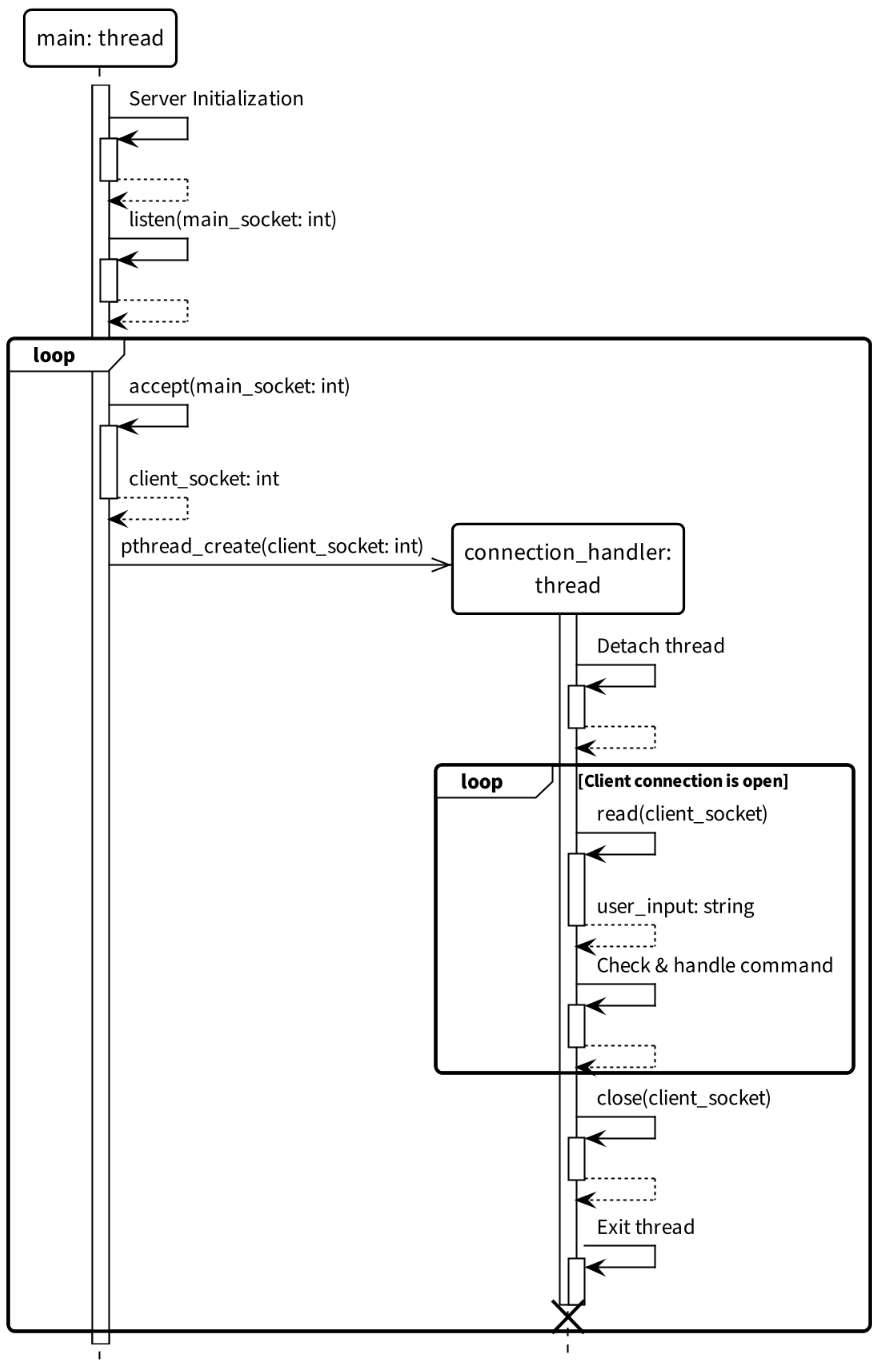

FIGURE 4: Diagramme de séquence du fonctionnement général du serveur

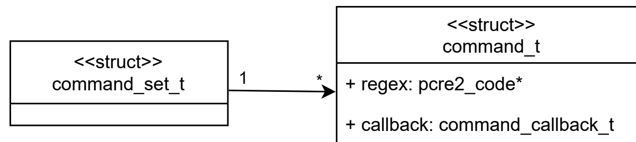

FIGURE 5: Diagramme de classes des structures de données des commandes côté serveur

3.2 Interprétation des commandes et fichiers

Dans un second temps, les commandes reçues par le serveur sont analysées, et exécutées si elles sont valides. Le serveur peut à la fois recevoir certaines commandes des clients via le réseau, mais également des commandes spécifiques depuis un interpréteur lancé à l'exécution du serveur, dans un autre thread.

Tout d'abord, la syntaxe des commandes est analysée à l'aide de la bibliothèque d'expressions régulières PCRE2 (*Perl Compatible Regular Expressions*), plus complète que la bibliothèque POSIX.2 `regex` [1]. L'analyseur essaie de trouver une correspondance entre l'entrée utilisateur et chacune des expressions régulières qui définit la syntaxe des commandes. Il s'arrête dès que la première correspondance est trouvée et appelle la fonction de rappel, ou *callback*, associée à la commande. Si l'entrée ne valide aucune expression régulière, un message d'erreur est renvoyé.

Cette solution permet de factoriser efficacement le code : pour implémenter une commande, il suffit d'ajouter une commande `command_t` à l'ensemble de commandes `command_set_t` qui lui correspond. On sépare ainsi l'ensemble des commandes autorisées pour les clients, de celles autorisées pour l'invite de commandes du serveur. Une `command_t`, représentée en Figure 5, est composée d'une expression régulière, ainsi que de la fonction de rappel à appeler si une correspondance est trouvée.

Par ailleurs, les expressions régulières ne se limitent pas à vérifier la syntaxe des commandes, mais permettent aussi de *parser* les arguments et les options passées : il suffit pour cela d'utiliser des parenthèses capturantes dans l'expression régulière. Ainsi, l'expression régulière présentée en Code 1 décrit la syntaxe de la commande d'ajout de vue, et capture dans des groupes successifs l'identifiant de la vue, ses positions `x` et `y`, et enfin, sa largeur et sa hauteur.

```
^add view (\S+) (\d+)x(\d+)\+(\d+)\+(\d+)$
// -> Matches "add view <view id> <x>x<y>+<width>+<height>"
```

Code 1: Expression régulière décrivant la syntaxe de la commande `add view` et permettant d'extraire ses arguments

Les arguments capturés sont alors transmis à la fonction de rappel sous forme d'un tableau de chaînes de caractères, qui peut les utiliser pour interagir avec l'aquarium ou renvoyer une réponse au client. Ainsi, la fonction de rappel `add_view` se sert de ses arguments pour ajouter la vue à l'aquarium, selon les paramètres spécifiés par l'utilisateur. Elle peut renvoyer une réponse au client ou à l'interpréteur de commandes du serveur à l'aide des flux `out` et `err` passés en paramètres. Néanmoins, lors d'une communication avec le client, les flux `out` et `err` sont identiques : ils envoient tous deux le texte écrit via la *socket*. Cette implémentation permet d'ajouter facilement de nouvelles commandes, sans avoir à modifier le code existant ou réimplémenter un *parser* d'arguments pour chaque commande.

4 Fonctionnement du client

Une fois le serveur en écoute sur un port donné, le client Java peut s'y connecter en TCP afin de lui envoyer des requêtes. Il peut alors envoyer des commandes pour consulter l'état de l'aquarium afin d'afficher les résultats à l'utilisateur, mais également pour ajouter, supprimer ou modifier l'état des poissons de l'aquarium.

4.1 Communication avec le serveur

Tout d'abord, la communication réseau avec le serveur est assurée par la classe `AquariumSocket`, représentée en Figure 6 et instanciée dans le module contrôleur. Comme dans le cas du serveur, la connexion en TCP est ouverte à l'aide de l'interface socket, en spécifiant l'adresse IP du serveur et son port d'écoute. Le contrôleur client démarre ensuite un thread chargé de communiquer via la socket avec le serveur. Pour cela, le client envoie une demande d'identification `hello`, telle que définie dans le protocole de communication dans [Ahm21]. Si le serveur accepte la demande de connexion, le client démarre alors trois nouveaux threads : un premier reçoit les messages du serveur envoyé via la socket, un deuxième récupère les entrées de l'utilisateur

1. Contrairement à la norme POSIX.2, la spécification PCRE2 propose notamment un mécanisme de parenthèses non capturantes `(?:)`. Elle propose également une option pour supporter les caractères Unicode dans les classes de caractères.

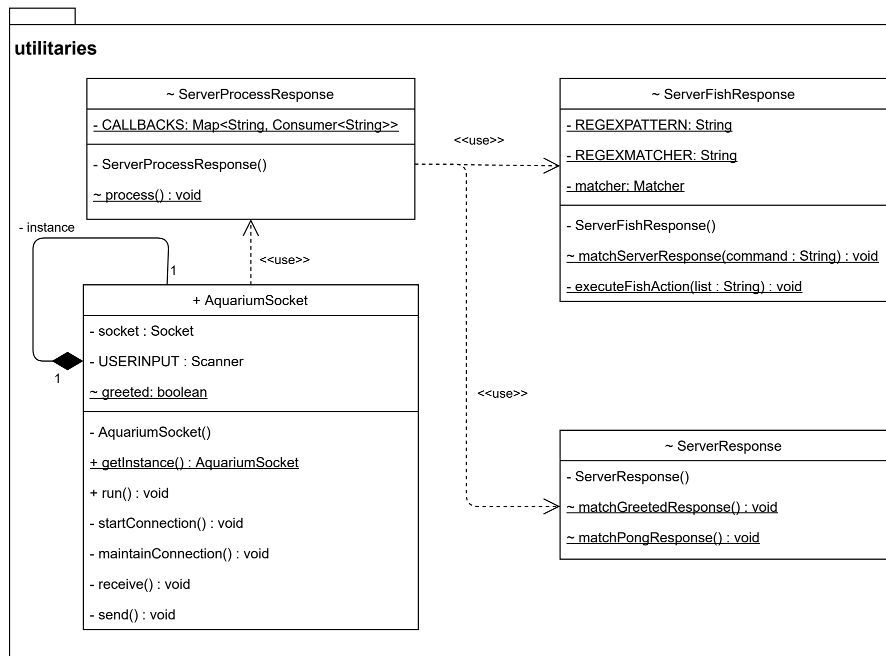

FIGURE 6: Diagramme de classes du module de communication et d'interprétation des réponses du client

écrites dans le terminal et les envoie au serveur, et le dernier maintient la connexion avec le serveur par l'envoi régulier de commandes `ping`.

4.2 Interprétation des réponses du serveur

L'étape suivante de la communication consiste à interpréter les réponses du serveur, renvoyées suite aux requêtes du client. Ces requêtes client peuvent être envoyées automatiquement, comme avec un `ping` ou une demande de poissons `getFishesContinuously`, ou peuvent avoir été saisies par l'utilisateur, dans l'invite de commandes du client. En plus de permettre l'envoi de requêtes avec la méthode `sendCommand`, la classe `AquariumSocket` permet de recevoir les réponses reçues et de les interpréter. Plus précisément, lorsque le thread chargé de recevoir les réponses appelle la méthode `receive` d'`AquariumSocket`, le contenu de la socket est lu et transmis en argument à la méthode de classe `process` de la classe `ServerProcessResponse`, représentée en Figure 6.

Cette méthode `process` vérifie si la réponse reçue du serveur valide une des expressions régulières `"^list.*"`, `"^greeted.*"` ou `"^pong.*"`, stockées en tant que clés dans le tableau associatif `CALLBACKS`. Puis, lors d'une correspondance, le *callback* associé à l'expression régulière dans la `Map` est appelé : les expressions régulières `"^list.*"`, `"^greeted.*"` et `"^pong.*"` sont respectivement associées aux méthodes `matchServerResponse` de `ServerFishResponse`, et `matchGreetedResponse` et `matchPongResponse` de `Server-Response`. Ainsi, chaque réponse valide du serveur est associée à une action à effectuer par le client, qui peut « parser » plus en détails les réponses reçues. Par exemple, les réponses `list` sont « parsées » dans la méthode `matchServerResponse` de `ServerFishResponse`, afin de récupérer les informations sur les poissons et les mettre à jour.

4.3 Ajout des poissons et des transitions

Lorsqu'une liste de poissons est reçue par le client, deux cas de figures se présentent. Si le poisson n'est pas connu du client, le contrôleur crée une nouvelle instance de poisson et stocke sa destination. Sinon, le contrôleur ajoute une nouvelle position à atteindre pour les poissons, en créant une nouvelle instance de la classe `FishProperties`. Cette instance est enfilée dans la `BlockingQueue` du poisson, qui représente la suite des positions auxquelles le poisson doit se déplacer par la suite.

Plus précisément, cette `BlockingQueue` permet de répondre au problème du « producteur-consommateur », un problème de synchronisation des ressources. D'un côté, le contrôleur joue le rôle du producteur en remplissant la `BlockingQueue` du poisson d'instances de `FishProperties`. De l'autre, la classe `Fish` joue le rôle de consommateur, en réalisant les transitions une à une. Plus précisément, nous créons un thread par poisson, chargé de réaliser les transitions au fur et à mesure. Ainsi, tant que la `BlockingQueue` du poisson n'est pas vide, le consommateur exécute chaque transition de celle-ci. Lorsqu'elle est vide, l'implémentation Java de `BlockingQueue` permet d'attendre automatiquement qu'un élément soit disponible avant de continuer son exécution.

Cependant, lorsqu'une transition est lancée, celle-ci est exécutée en parallèle du thread qui a lu dans la `BlockingQueue`. De ce fait, si notre consommateur exécutait l'ensemble des transitions les unes à la suite des autres, notre poisson réaliserait l'ensemble de ses transitions en parallèle, et se déplacerait donc de manière non-déterministe. Afin d'éviter ce cas de figure, nous avons mis en place un système de verrous. Plus précisément lorsque la transition la plus récente d'une instance de la classe `Fish` est en cours d'exécution, celle-ci écrit la valeur `RUNNING` dans son attribut `status`. De cette manière, le thread chargé de déclencher les différentes transitions du poisson entre dans une attente passive, du fait de la valeur de ce statut. Lorsque la transition en cours termine, le thread en attente peut alors exécuter la prochaine transition en tête de la `BlockingQueue`.

Même si la bibliothèque JavaFX propose un objet `SequentialTransition` qui permet d'exécuter séquentiellement une liste de transitions, celui-ci ne permet pas d'ajouter de nouvelles transitions lorsque qu'une transition est déjà en cours d'exécution. La solution que nous avons implémentée permet de résoudre cette limitation, tout en garantissant l'exécution séquentielle des transitions du poisson. Ainsi, le client est non seulement capable d'envoyer des requêtes au serveur, mais également d'interpréter les réponses, notamment pour afficher et déplacer les poissons dans l'interface graphique.

Conclusion

À l'issue de la sixième séance de projet, la majorité des modules et des fonctionnalités demandées sont implémentées. Certaines commandes doivent encore être développées, notamment pour les envois périodiques et continus de poissons, ou la déconnexion des clients inactifs. De plus, la version actuelle ne permet pas de sauvegarder les logs du serveur et du client dans un fichier, ou de contrôler leur verbosité.

Pour le moment, la modification des paramètres du serveur impose de modifier des constantes de préprocesseur et de recompiler le programme : par la suite, la configuration devra être chargée depuis un fichier pour en faciliter la modification. Enfin, même si le traitement des requêtes des différents clients par le serveur a lieu en parallèle, la version actuelle ne garantit aucune protection face aux *data races*, ce qui pourrait, dans certains cas, provoquer des incohérences dans les structures de données serveur ou client.

Références

[Ahm21] Toufik AHMED. *Projet Réseaux RE203 : "Simulation d'un aquarium de poissons"*. 2021. URL : https://docs.google.com/document/d/1MzZG0qDfVr8U50v8X79yvaPfToj8Yz4DwiGg3swl1kk/edit (visité le 29/04/2021).