CHADOW

Projet de programmation Réseaux M1
Protocole, Application client/server sur TCP par

Antoine Bastos

Valentin Sebbah

Université Gustave Eiffel 2023-2024

Le client peut être testé avec l'instance publique du serveur qu'on a mis en place. Elle sera ouverte pendant 1 mois, à l'adresse **217.182.68.48** sur le port **7777**.
La mise en production est déclenchée par une pipeline ci/cd lorsque qu'on commit sur la branche main.

Après coup, nous n'avons pas pris en compte qu'il fallait pouvoir nat le port d'écoute du client sur le routeur. Il faudrait forward le port manuellement ou utiliser le protocole UPNP/NAT-PMP pour le faire automatiquement.

Fonctionnalités

Toutes les fonctionnalités essentielles du projet sont opérationnelles :

- Connexion des clients avec des pseudonymes uniques.
- Chat global et messages privés entre les clients.
- Partage, recherche et téléchargement de codex.
- Téléchargement parallèle pour accélérer le processus.
- Utilisation de proxys pour le téléchargement en mode fermé.
- Possibilité d'utiliser le serveur comme proxy dans des conditions spécifiques.

Statistiques indicatives du téléchargement d'un codex

Ces tests ont été réalisés avec des tailles de chunks de 128kb, taille par défaut du client qui peut être modifié en paramètre.

Mode ouvert

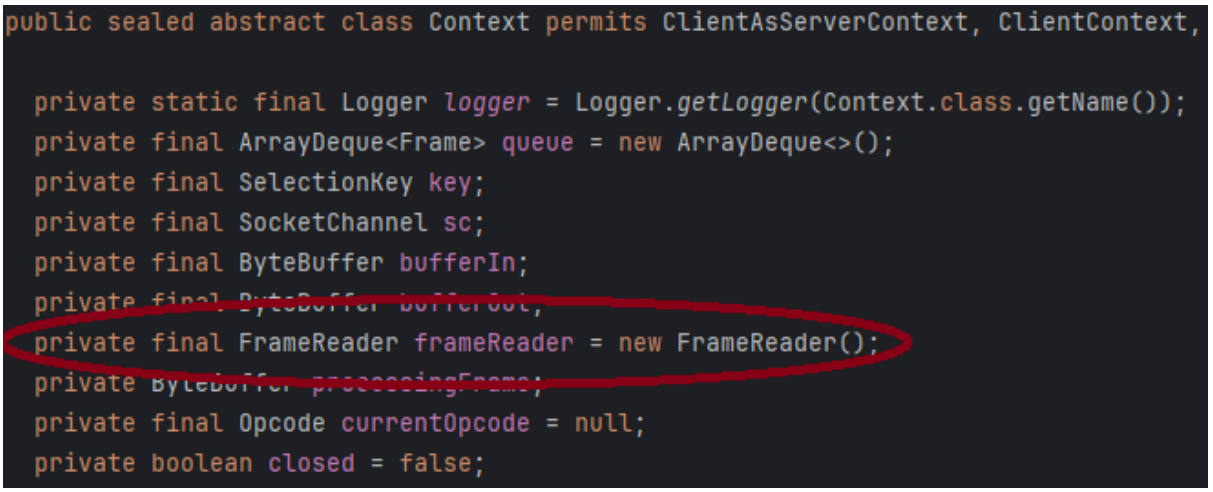
Fichier / dossier	Téléchargement vers un client (local)	Téléchargement vers plusieurs client (locaux)
300Mo	✓ 15 sec	✓ 12 sec (3 sharers)
3Go	✓ 2min30	✓
5Go	✓ 6min35	✓ 6min05 (3 sharers)

Mode fermé

Fichier / dossier	Téléchargement vers un client (local)	Téléchargement vers plusieurs client (locaux)
300Mo	✓ 5min40	✓
3Go	✓ 50min	✓

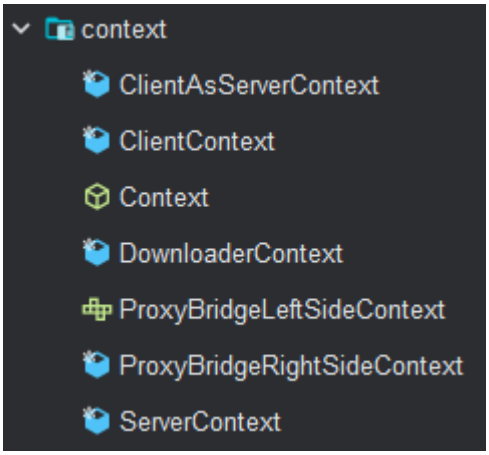
Intégration des Retours de la Soutenance Bêta

Lors de la soutenance bêta, plusieurs remarques ont été formulées. Notamment, il a été relevé que nous n'avions pas implémenté de lecteur (reader) de Frame à ce stade du développement. Cette absence rendait le code relativement lourd au sein des Contextes. En réponse à cette observation, nous avons introduit le FrameReader (décrit dans la partie FreamReader), afin de simplifier la gestion des lectures dans les Contextes en ne nécessitant qu'un seul lecteur.



Unique Reader utilisé par la class Context, le FrameReader

Une demande spécifique lors de la soutenance bêta concernait l'élimination de la duplication de code au sein des Contextes, dans le but de factoriser le code et d'améliorer sa maintenabilité. Pour résoudre ce problème, nous avons introduit une structure de classe plus modulaire. Ainsi, nous avons créé la classe Context en tant que classe mère, accompagnée de classes filles spécifiques utilisées pour les clients et le serveur.



Class mere Context avec ses classes filles

Cette refonte nous a permis de regrouper le code commun au sein de la classe Context, tandis que les classes filles ont été dédiées à la gestion des fonctionnalités spécifiques des clients et du serveur.

Évaluation des Contributions du binôme

Antoine : 60%
Valentin : 40%

Points saillants

L'interface à été conçue pour correspondre à ce que l'on attend d'un chat IRC. La présentation et les couleurs rendent le client en ligne de commande attrayant et visuellement compréhensible.

L'interface utilisateur

La difficulté majeure est d'avoir une interface qui se met à jour et pouvant prendre en compte ce que l'utilisateur entre au clavier. Ce n'était facile, car le terminal est bufferisé et quand l'affichage est redessiné il efface aussi ce qui a été entré par l'utilisateur avant qu'il puisse appuyer sur 'entrée'.
Il aurait fallu passer en mode "raw", ce qui est plateforme dépendant, et pour ne pas trop réinventer la roue il aurait été préférable d'utiliser une bibliothèque tierce ; ce qui n'est pas autorisé. Nous avons fait un compromis, un thread affiche et un autre récupère les entrées du clavier. Par défaut le thread d'affichage est prioritaire, et quand l'utilisateur appuie sur

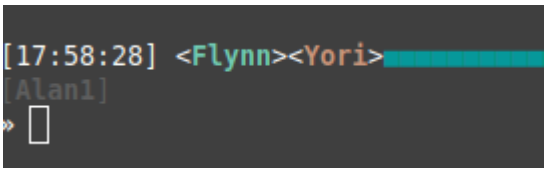
entrée, ça flush le buffer et à ce moment on dit que c'est le thread de l'entrée clavier qui prend la main et l'affichage est mis sur pause.

Nous avons fait au mieux pour que l'interface soit dynamique et puisse être redimensionnée (avec la commande :r), mais aussi paginer l'affichage, en le rendant scrollable ou permettant la sélection d'éléments dans une liste.

Remarque: On ne pouvait pas rendre le redimensionnement automatique de l'affichage car c'est dépendant de la plateforme. On a besoin de connaître la taille du terminal à tout instant, ce qui est possible via des instructions spécifiques.

Les messages privés

Une fonctionnalité intéressante est que l'utilisateur possède une barre de notifications (au niveau de l'affichage de l'heure en temps réel, au-dessus de son input). Quand il reçoit un message privé d'un autre utilisateur, la conversation associée est en non-lu, et une notification apparaît.



Nous n'avons pas eu le temps d'ajouter d'autres types de notifications (téléchargement complété, info diverse, erreurs, …) mais c'est possible.

Si l'utilisateur échange avec un autre par message privé, et que l'un se déconnecte, la conversation est gardée chez l'autre, de sorte que les messages puissent être encore visibles. L'username du correspondant est modifié pour indiquer qu'il n'existe plus.

Pagination et filtres de la recherche de fichiers partagés

La commande (:f ou :find) permet de rechercher les fichiers qui sont en partage, en requêtant le serveur. Afin d'éviter de demander un très grand nombre de résultats nous avons mis en place un système de pagination, qui les récupère au fur-et-à-mesure que l'utilisateur les parcourt.

Mise en partage de dossiers

Plutôt que de se contenter des partage des fichiers un à un (ce qui reste possible), nous avons fait en sorte de pouvoir partager des dossiers complets. L'arborescence est recréée à l'identique au moment du téléchargement.

Partage de gros fichiers

Dans l'implémentation de notre client, les chunks sont d'une taille de 128 Kb ce qui permet d'avoir un téléchargement rapide même de fichier de taille conséquente. Nous avons testé 5Go sans problème. Le bon fonctionnement du client et du chat n'est pas impacté car les trames sont mises dans une file avant d'être envoyées, donc les autres trames (messages, recherche de codex, etc..) s'intercalent entre deux chunks.

Implémentations du serveur

Gestion des proxies

Gestion des Requêtes Client et des Détails des Proxys

1. ClientRequest (Requête Client) :
   - Cette classe interne représente une requête client avec le contexte du serveur associé et l'identifiant du codex demandé par le client.

2. ProxiesDetails (Détails des Proxys) :
   - Cette classe interne représente les détails des proxys associés à une requête client, y compris le nombre de proxies, le nombre de partageurs, et les détails de chaque chaîne de proxys.

3. ChainDetails (Détails de la Chaîne) :
   - Cette classe interne représente les détails d'une chaîne de proxys, y compris la liste des proxys contactés et les proxys qui ont confirmé la chaîne.

Fonctionnalités Principales

1. Initialisation des Requêtes :
   - La méthode `initRequest` initialise une nouvelle requête client en calculant d'abord les partageurs possibles pour le codex demandé. Ensuite, elle crée les différentes chaînes de proxys et contacte les proxys pour chaque chaîne.

2. Contact des Proxys pour une Chaîne :
   - La méthode `contactingProxiesForAChain` gère le processus de contact des proxys pour une chaîne donnée, en commençant par le dernier proxy de la chaîne et en remontant vers le premier proxy.

3. Sélection du Meilleur Proxy :
   - La méthode `selectBestProxy` sélectionne le meilleur proxy parmi les proxys disponibles en fonction de certains critères, tels que l'exclusion du client et des partageurs, ainsi que des proxys déjà contactés.

4. Gestion de la Confirmation des Proxys :
   - La méthode `proxyConfirmed` gère la confirmation d'un proxy pour une chaîne spécifique et gère l'état de la requête associée. Si tous les proxys de la chaîne ont confirmé la requête, une réponse est envoyée au client.

5. Autres Fonctionnalités :
   - La classe `ProxyHandler` comprend également des méthodes pour la gestion générale des requêtes et des proxys, telles que la suppression de toutes les instances d'un client, la génération d'un entier unique, le calcul du nombre possible de partageurs pour un codex donné, et la sauvegarde du serveur en tant que proxy pour une chaîne spécifique.

Utilisation dans le Système

- Le `ProxyHandler` est intégré au serveur pour gérer les requêtes des clients et la communication avec les proxys.
- Il joue un rôle central dans la coordination des actions entre les clients, les serveurs et les proxys, garantissant une distribution efficace et sécurisée des ressources partagées.

Implémentations des readers

Le GlobalReader

Dans notre projet, nous avons cherché à maximiser la réutilisabilité du code en évitant de créer un lecteur spécifique pour chaque type de trame (`Frame`). Puisque les trames sont des records, nous avons conçu la classe `GlobalReader` pour gérer la lecture de n'importe quel type de trame.

Le but principal de la classe `GlobalReader` est de fournir une solution générale pour la lecture de trames, en permettant de traiter dynamiquement différents types de données. Voici comment elle fonctionne :

1. Traitement des Paramètres
   - Pour chaque paramètre d'une trame, le `GlobalReader` utilise un lecteur (`Reader`) approprié en fonction du type de données.
   - Trois cas peuvent se présenter :
     - Si le paramètre est de type primitif, le lecteur correspondant est utilisé.
     - Si le paramètre est une autre trame (`Frame`), un nouveau `GlobalReader` est instancié pour ce sous-type.
     - Si le paramètre est un tableau (`Array`), un lecteur de tableau (`ArrayReader`) est utilisé.

2. Construction de la Trame

- Une fois que tous les paramètres de la trame ont été lus, le constructeur de la trame est appelé avec les valeurs récupérées.
- La trame ainsi construite est renvoyée.

L'utilisation de `GlobalReader` présente plusieurs avantages :
- Réutilisabilité : Élimine la nécessité de créer un lecteur spécifique pour chaque type de trame.
- Flexibilité : Capable de traiter différents types de données dynamiquement.
- Modularité : Facilite l'extension du projet avec de nouveaux types de trames sans modifier le code existant.

Le ArrayReader

Le `ArrayReader` a pour fonction de lire n'importe quel sous-type d'un record qui est sous forme d'un tableau (`T[]`). Pour ce faire, il utilise un `GlobalReader` qui lit tous les éléments du tableau.

Le but principal du `ArrayReader` est de fournir une solution générique pour la lecture de tableaux d'éléments de type `T`. Voici comment il fonctionne :

1. Traitement des éléments du tableau
   - Le `ArrayReader` utilise un `GlobalReader` pour lire chaque élément du tableau.
   - Le `GlobalReader` est capable de lire dynamiquement différents types de données, ce qui permet au `ArrayReader` de traiter des tableaux contenant des éléments de divers types.

2. Construction du Tableau
   - Une fois que tous les éléments du tableau ont été lus, le tableau est construit avec les valeurs récupérées.
   - Le tableau ainsi construit est retourné.

Il possède les mêmes avantages que le GlobalReader à savoir la réutilisabilité, la flexibilité et la modularité.

Le FrameReader

Le `FrameReader` joue le rôle de lecteur générique pour les trames échangées entre les clients et le serveur dans notre système. Chaque type de trame est associé à un `GlobalReader` dédié.

Le principal objectif du `FrameReader` est de gérer la lecture des différentes trames utilisées dans la communication entre les clients et le serveur. Voici ses fonctionnalités principales :

1. Traitement des Trames
   - Le `FrameReader` est conçu pour traiter les données brutes et les convertir en trames compréhensibles par le système.
   - Il utilise des `GlobalReader`s spécifiques pour chaque type de trame afin d'interpréter correctement les données reçues.

2. Gestion des Opcodes
   - Le `FrameReader` identifie l'opcode de chaque trame reçue pour déterminer le type de trame.
   - En fonction de l'opcode, le `FrameReader` utilise le `GlobalReader` associé pour lire et interpréter le contenu de la trame.

Points d’Améliorations

- Pas de sauvegarde de l’état du client ni du serveur. Nous avons manqué de temps pour implanter un mécanisme de sérialisation/désérialisation, ou tout simplement un fichier de sauvegarde décrivant les codex téléchargés, leur état et leur emplacement.

- Pas d’interface de commandes pour le serveur, c’est du fire-and-forget.

- La création d’un codex n’est pas asynchrone, donc l’interface bloque et ça peut être long pour les partages de plus de 2Go (< 1 min).

L’architecture suffisamment bien faite pour qu’on puisse régler tout ça facilement et ajouter plein d’autres fonctionnalités.