ADEDJOUMON Adam
GAMEIRO COSTA Bruce

M1 Informatique
2020 / 2021

ChatOS : le **Rapport**

Evolution depuis la soutenance

Comme demandé, les tokens sont maintenant des nombres aléatoires et non plus des nombres pouvant être déterminé à partir des pseudos. *(< 20 minutes)*

De plus voici quelques améliorations non demandées :

- À la connexion et la déconnexion d'un client, les autres clients reçoivent un message. *(< 5 minutes)*
- Correction d'un léger bogue qui provoquait l'acceptation forcée d'une connexion privée si un client envoi deux fois la même demande avant que l'autre client n'ait eu le temps d'accepter/de refuser. *(~10 minutes)*
- Bloque la lecture de la socket quand le client doit répondre à un requête de connexion privée. *(< 2 minute)*

Fonctionnalités

Ce qui est fonctionnel :

- L'envoi et la réception des différents types de messages (généraux et privées)
- La vérification de l'existence d'un client lorsqu'il est demandé (par une connexion privée ou par un message privé)
- L'établissement d'une connexion privée entre deux clients
- L'authentification par un pseudo n'étant pas déjà utilisé
- La requête (HTTP) de ressource textuelle via une connexion privée (qui s'affiche donc dans le terminal)
- La requête (HTTP) de ressource via une connexion privée (qui est sauvegardé dans l'espace de travail)
- L'envoi d'une réponse (HTTP) avec un code d'erreur si la ressource demandée n'existe pas
- La réutilisation d'une connexion privée déjà établie entre 2 clients pour une nouvelle requête HTTP
- La résiliation automatique des connexions privées impliquant un client qui s'est déconnecté
- La déconnexion d'un client ne fait pas planter le serveur
- La fermeture du serveur ne fait pas planter les clients
- L'échappement des caractères '@' et '/' en début de ligne
- La possibilité de récupérer une erreur (voir la section appropriée)

Ce qui ne fonctionne pas :

- Jusqu'ici tous les bogues ont été corrigés.

Choix d'implémentation

Par des contraintes de temps par rapport aux autres projets, les suggestions que vous nous avez faites (patrons *Visiteur/Observer*) n'ont pas été implémentées.

Les contextes

Ce projet contient beaucoup (5) de contextes différents.
Ayant beaucoup de fonctionnalités proches (voir identiques parfois) il a été choisi de regrouper leur code en commun dans une classe abstraite `AbstractContext`.

Ceci nous permet donc de n'avoir qu'un exemplaire des fonctions : `doRead`, `doWrite`, `doConnect`, `updateInterestOps`, `processOut` et `queueMessage`.
Avec bien sûr quelques `Override` pour ajouter des fonctionnalités qui auraient pu être remplacées par des observers si ça avait été vraiment nécessaire, mais étant donné qu'il y a en tout 4 `Override` de ces fonctions parmi les 5 sous-classes cela aurait été *overkill* à mon point de vue. (Mais dans un vrai projet qui aurait été en évolution sans fin définitive, effectivement il aurait mieux fallu utiliser des observers)

Les paquets

Le principal élément de ce projet sont les paquets. Il y a au moins 2 façons de les représenter :

- Un **paquet** ⇔ un **classe/record**. Ça a pour avantage de pouvoir mieux séparer les paquets et de pouvoir implémenter le patron *Visiteur*.
  Mais ça a pour défaut de rajouter 10 classes rien que pour tous les paquets sans parler des visiteurs ni des interfaces.
- Une classe qui regroupe tous les paquets.

Les avantages sont que tout est au même endroit et qu'il suffit de trouver la bonne branche d'un `switch` pour savoir ce qu'on fait et ça fait beaucoup moins de classes.

Les désavantages sont que si on doit ajouter un paquet il faut retrouver tous les endroits où on utilise le type du paquet pour ajouter le nouveau (donc on va en rater plusieurs fois) et que si on avait beaucoup plus de paquets les fonctions feraient trop de lignes pour être lisibles.

Donc comme dit au début de cette section c'est la deuxième solution qui a été mise en place pour la raison que pour un petit projet comme celui-ci la première solution est plus coûteuse en temps et en lisibilité que la seconde.
Et surtout ça aurait nécessité un changement de structure trop important pour être sûr de rendre un projet propre et fonctionnel à temps.

Les lecteurs (`Reader`)

Les lecteurs sont utilisés par tout le programme pour lire les octets.
Il en existe plusieurs qui dérivent tous de l'interface Reader :

- Un lecteur de chaine de caractère (`StringReader`) qui permet de lire un entier représentant la longueur de la chaine de caractère qui le suit, encodé en UTF-8.
- Un lecteur de paquet (`PacketReader`) qui lit un paquet. Donc un octet puis en fonction de la valeur de cet octet lit différentes choses. (Pour plus de détail lire l'annexe de la [RFC](Protocol.txt)).
- Un lecteur de ligne HTTP (`HTTPLineReader`) qui lit des octets jusqu'à lire `\r\n` (qui indique une fin de ligne en HTTP).
- Un lecteur de paquet HTTP (`HTTPReader`) qui lit un message HTTP pouvant être une requête ou une réponse.
- Un lecteur de rejet qui contrairement aux autres rejette les octets lu jusqu'à lire une récupération d'erreurs

Pour les mêmes raisons qu'il n'y a pas une classe pour chaque paquet, il n'y pas un lecteur pour chaque paquet.

Autres choix

Il y a aussi quelques choix personnels ayant été fait au début du projet et n'ayant pas été abandonnés bien qu'ils soient très probablement inutiles.

La récupération d'erreur

La récupération d'erreur était principalement utilisée pour faire en sorte de ne pas déconnecter un client ayant envoyé un paquet erroné au serveur.
Un paquet erroné est un paquet où il y a :

- soit un code invalide (donc ne figurant pas dans la [RFC](Protocol.txt)) autant pour le type du paquet que pour le type de l'erreur d'un paquet d'erreur.
- soit une longueur invalide (donc inférieure à 1 ou supérieur à 1024)

Si un de ces cas survenait, le serveur se mettait en mode 'rejet' avec ce client et rejetais tous les octets jusqu'à lire un paquet de récupération c'est-à-dire un paquet composé du type `ERR` puis du code d'erreur `ERROR_RECOVER`. À la suite de ça il se remettait en mode 'normal' et recommençait à lire correctement.

Dans l'ensemble c'est une bonne idée mais elle est inutile puisque si le projet est bien fait, ce cas n'arrive jamais.
Et dans le cas où le client est mal fait, le serveur va souvent se mettre en défaut puisque l'erreur ne pouvant pas venir du protocole TCP (pas de perte de paquet pas d'échange, ...), l'erreur vient forcément du client. Donc si elle se produit une fois, elle se reproduira !
Donc il vaut mieux fermer la connexion.

Cependant j'avais déjà prévu cette façon de faire au tout début et je trouvais ça élégant... Inutile mais élégant.

La gestion des gros fichiers

Encore une fois c'est une idée qui est séduisante mais inutilement compliqué... Le problème est que lors d'une connexion privée si le client demande un fichier trop volumineux (donc supérieur à la limite que j'ai mise qui est de 32'768 octets - l'entête), on ne peut pas
mettre tout dans le buffer.

La solution évidente est donc d'envoyer le fichier en plusieurs morceaux en calculant la taille au début et en la mettant dans le header. De cette façon le client recevant le paquet sait quelle taille de fichier il attend et peut le gérer plus facilement.

La solution mise en place est de remplir le buffer avec le fichier envoyer ce paquet (en-tête + contenu) puis avec ce qu'il reste du fichier recommencer et ça jusqu'à ce qu'il ne reste rien.
Donc pour dire ça autrement on divise le fichier en morceaux de 31'768 (taille maximale - en-tête) et chaque morceau sera envoyé dans un paquet HTTP ayant un header avec un champ `'Content-Size'` avec pour valeur maximale 31'768.
Ensuite l'assemblage se fait par ajout à la fin du même fichier.

Si on compare les deux méthodes, elles ont toutes les deux des avantages et c'est pour ça que je garde ma solution.

Dans la première solution si le fichier fait 1 Giga, il faudra soit alloué un buffer de 1 Giga, soit découper le contenu en plusieurs buffers donc ça ressemble un peu.
Mais on peut s'en sortir puisque on peut identifier le début et la fin du fichier.

Dans la seconde solution, les buffers ont une taille maximale donc la mémoire ne peut pas exploser.
Le souci est que pour chaque paquet on perd 1000 octets d'en-tête et qu'il est impossible de déterminer
à coup sûr la fin de la réception d'un fichier.

La génération de token

Au début, la génération de token se faisait en combinant le `hashCode` des deux pseudos par une opération symétrique.
Cela permettait d'avoir un code facilement calculable à partir des pseudos sans prendre en

compte l'ordre dans lequel ils sont fourni.

Cependant suite à la soutenance on m'a fait remarquer que ce n'est pas sécurisé puisque les deux mêmes pseudos donneront toujours le même token mais que les pseudos n'appartiennent pas forcément toujours à la même personne.
Donc j'ai modifié la façon de générer les pseudos en ajouter une `Map` qui contient les couples de pseudos et leur token attribué.
Le token est généré la première fois qu'un client fait une demande de connexion privée et est détruit lorsque la connexion est refusée ou lorsqu'elle est interrompue.

Parties

Voici la liste des différentes classes de ce projet avec une rapide description de ses fonctionnalités et des choix fait.

`Context`

Interface commune à tous les contexts.
Tous les contexts doivent définir 2 méthodes : `doRead` et `doWrite`.
La fonction `doConnect` est nécessaire pour les contextes du client mais pas pour le serveur.
Cependant pour plus d'unicité cette fonction est aussi dans l'interface (cela évite d'avoir 2 interfaces avec une fonction de différence)

`AbstractContext`
Classe abstraite permettant de factoriser les méthodes qui étaient en commun dans tous les contextes. Il y a déjà eu une catégorie parlant des choix de cette implémentation.

`Client`

Le client est la partie utilisateur de ce projet, tout ce qui a été fait l'a été en pensant à l'ergonomie et ce qui pouvait être le plus agréable par rapport à l'utilisateur.

`ClientChatOS`
Classe regroupant toutes les fonctionnalités disponibles pour le client.
Permet de se souvenir des connexions privées actives ainsi que celles en attente.
Se découpe en 2 threads : un pour la console qui transmet ce que l'utilisateur entre dans `System.in` vers le contexte principal. Et un pour traiter les sockets actives (faire les lectures, les écritures et tous les traitements).

Elle conserve les connexions privées établies et celles qui sont en attente.
Ce qui permet de ne pas renvoyer plusieurs fois la même demande tant qu'on n'a pas reçu de réponse et de réutiliser les connexions déjà établies.

`MainContext`

Le contexte principal du client. Il lui permet d'analyser les messages entré dans la console et de recevoir des messages venant du serveur.

Choix :
Ici il a été fait le choix de traiter les messages à envoyer dans le contexte plutôt que dans le thread de la console pour renforcer l'encapsulation.
Si on avait fait ça dans la console il aurait été plus difficile de différencier si le message était prévu pour être le pseudo lors de l'authentification ou bien un message normal ou encore un refus ou une acceptation de connexion privée.

Tout cela aurait pu être fait mais avec un risque de modification concurrente et en rajoutant des getters/setters (donc en fragilisant l'encapsulation).

`PrivateConnectionContext`

Ce contexte permet de gérer une connexion privée entre le client actuel et un autre client (B).
Il permet donc de demander une ressource à B et de fournir une ressource demandée par B uniquement si elle existe dans l'espace de travail.
Si la ressource reçue est de type `text/plain` alors elle sera affichée, sinon elle sera sauvegardée dans l'espace de travail en créant le fichier si manquant ou en ajoutant à la fin du fichier si le fichier existait déjà.

Serveur

Le serveur permet d'héberger les clients et de transmettre les différents paquets entre les clients.

ServerChatOS

Le serveur est composé d'un socket permettant d'accepter les clients qui tente de se connecter.
Quand le client est accepté il est mis en "attente d'authentification" jusqu'à ce que le client choisisse un pseudo valide.

Le serveur se doit de stocker les pseudos déjà utilisé pour permettre aux clients de choisir un pseudo unique et pour vérifier si le destinataire des messages privés ou des demandes de connexions privées existe bien.

Le serveur garde en mémoire les connexions privées établies ainsi que les demandes de connexions privées et les tokens déjà utilisés.

`ConnectionContext`

C'est donc ici que le client est mis quand le serveur attends qu'il s'authentifie.
Il y a deux façons de s'authentifier :

- En envoyant un pseudo correct (non pris).
- En envoyant un token valide.

Dans le premier cas on a donc un client "classique" qui se connecte au serveur en utilisant un pseudo. Ce contexte sera donc remplacé par un contexte de client.
Dans le second cas, le client se connecte en utilisant un token donc c'est une extrémité d'une connexion privée ayant déjà été accepté au préalable.
Si le serveur ne connait pas ce token, la demande de connexion sera refusée.

`ClientContext`

Ce contexte représente une connexion dite "classique".
Le client connecté a donc accès aux fonctionnalités suivantes :

- Envoyer un message général. Il sera donc transmis à tous les autres clients connecté en précisant qui l'a envoyé.
- Envoyer un message privé à un autre client. Le message sera envoyé à l'autre client que si celui-ci existe.
- Demander/Accepter/Refuser une connexion privée.

Si une connexion privée est demandée, alors la couple de pseudo sera enregistré comme ayant fait une demande et pourra être validé ou invalidé par la réponse de l'autre client.
Si la connexion est refusée par l'autre client alors la demande sera supprimé des demandes en attente. Et si elle est acceptée alors les deux clients recevront un token permettant de s'identifier à partir d'une nouvelle socket.

`PrivateConnection`

Cette classe permet de synchroniser les deux extrémités d'une connexion privée.
Cette classe est paramétrée par un token et attend de recevoir deux connexions privées avant de les démarrer. Lorsqu'une des deux connexions est fermées, cette classe s'occupe de supprimer tout ce qui concerne la connexion privée.

`PrivateConnectionContext`

Ce contexte est utilisé lorsqu'une connexion se fait à l'aide d'un token.
Elle est mise en attente jusqu'à ce qu'elle puisse être connectée avec l'autre extrémité de la connexion privée. Une fois fait tous les octets lus seront envoyé au contexte associé à l'autre extrémité.

Reader

Les lecteurs sont une partie très importante de ce projet.
Ce sont eux qui permettent d'interpréter les suites d'octets envoyé par le serveur et le client.
Il en existe plusieurs types permettant de lire différentes entrées.
Mais chaque lecteur possède une méthode permettant de lire des données à partir d'un buffer, une méthode qui permet de récupérer l'élément créé et une méthode permettant de remettre à zéro le lecteur.

`StringReader`

Ce lecteur permet de lire une chaine de caractère encodé en UTF-8 précédé d'un entier indiquant sa longueur (entre 1 et 1'024).
Ce lecteur est utilisé dans tout le projet étant donné qu'il permet de lire toutes les chaines de caractères (les pseudos et les messages).

`PacketReader`

Ce lecteur permet de lire un paquet. Un paquet est composé d'un code indiquant son type. À partir de ce type, il est possible de
déterminer ce qui doit être lu par la suite.
Pour plus d'information veuillez consulter la [RFC](Protocol.txt).

`HTTPLineReader`

Ce lecteur permet de lire une ligne dans un paquet HTTP.
Une ligne en HTTP est une suite d'octet encodé en ASCII et finissant par `\r\n`.
La ligne renvoyée par ce lecteur ne contient pas ces deux derniers caractères.

`HTTPReader`

Permet de lire un paquet HTTP pouvant être une requête, auquel cas il n'est composé que d'une ligne contenant "GET" suivit de la ressource demandée.
Ou bien une réponse qui contient plusieurs lignes d'en-tête suivit d'un contenu potentiel si la réponse a le code 200 (OK).

`RejectReader`

Ce lecteur permet de lire des octets jusqu'à la lecture d'un paquet composé du type `ERR` suivit du code d'erreur `ERROR_RECOVER`.
Ce lecteur est un peu particulier puisqu'il ignore les données au lieu de les enregistrer.
De plus, il se remet tout seul à zéro une fois qu'il a fini de lire les une récupération d'erreur.
Enfin contrairement aux autres lecteurs il ne lève jamais d'exceptions.

Packet

Les paquets sont le point fondamentale de ce projet. Les choix d'implémentations ont déjà été précisé plus haut et leur fonctionnement est décrit de façon détaillé dans la [RFC](Protocol.txt).

`PacketFactory`

Cette classe permet d'offrir une façon de créer des paquets plus agréable.
Elle fournit des méthodes pour créer chaque type de paquets en ne donnant que les arguments nécessaires.

`HTTPPacket`

Les paquets HTTP sont le moyen de communication des connexions privées. Il en existe 3 sortes :

- les requêtes qui contiennent la ressource que l'autre client nous demande.
- les réponses incorrectes qui indiquent que la ressource demandée n'existe pas chez l'autre client.
- les réponses correctes qui contiennent la ressource demandée.

`Display`

Ce groupe de classes ne sert qu'à afficher les informations sur la console du serveur et du client.

`AnsiColors`
Définie quelques couleurs utilisables dans un terminal ainsi que des méthodes pour colorer du texte et générer de nouvelles couleurs.

`ClientMessageDisplay`
Permet d'afficher les paquets que le client reçoit avec de la couleur afin que ce soit plus agréable.

`ServerMessageDisplay`
Permet d'afficher les paquets que le serveur reçoit. Normalement l'affichage n'est utilisé que lors de phase de début ou pour modérer ce qu'il se passe sur le serveur.

Les patrons utilisés

Il y a 2 patrons qui ont été utilisé dans ce projet :

- le patron statique *Factory method*. Utilisé pour créer les différents paquets utilisé dans le projet.
- le patron *Strategy*. Utilisé lors de la lecture des paquets qui permet de fabriquer différents paquets une fois le traitement fini