Kévin MAUGE
kmauge@etud.u-pem.fr
Pape NDIAYE
pndiaye@etud.u-pem.fr

Master Informatique
M1-S2
Groupe : 2

Projet
Programmation réseau

Matou - Service de Chat

Manuel développeur
- 01/05/2016 -

Présentation générale du projet

Objectifs

L'objectif du projet est d'implanter un service de chat avec une architecture centralisée (client/serveur).
Le serveur doit être implanté en mode "non-bloquant".
Les connexions privées doivent être directes.
L'envoi et la réception de fichier ne doit pas bloquer l'envoi et la réception des messages.

Fonctionnement

Comportement du serveur

Lorsqu'il est lancé, le serveur surveille un port de connexion et accepte toutes les connexions pendantes sur ce port. Si la limite du maximum de connecté est atteinte, il ferma la connexion aussitôt.

Lorsqu'un client envoie une requête au serveur, le serveur lit la requête argument par argument.
Si un client ne respecte pas le protocol, le serveur ferme immédiatement la connexion avec ce client. Ce cas ne doit jamais se produire avec un client réglementaire.
Les erreurs d'utilisation comme l'ouverture d'une connexion privée avec un utilisateur déconnecté ne sont pas considérées comme des erreurs de protocol. Dans ce cas, le serveur envoie un message d'erreur au client.

Comportement du client

Si un pseudo a été indiqué à la fin de la ligne de commande, le client va essayer de se connecter au serveur avec ce pseudo. En cas d'échec, le client s'arrête immédiatement.
Si aucun pseudo n'a été indiqué à la fin de la ligne de commande, l'utilisateur devra indiquer un pseudo manuellement sur le terminal afin que le client essaye de se connecter au serveur avec ce pseudo. En cas d'échec, le client attendra un nouveau pseudo jusqu'à recevoir un pseudo valide.

Une fois connecté et authentifié, le client est identifié de manière unique par ce pseudo durant toute sa session.
Le pseudo d'un client est libéré au moment de sa déconnexion.

Architecture

Packages

Le service dispose de plusieurs packages

- fr.upem.matou.client : package nécessaire au client
- fr.upem.matou.client.network : contient l'implémentation réseau du client
- fr.upem.matou.client.ui : contient l'implémentation de l'interface graphique du client
- fr.upem.matou.server : package nécessaire au serveur
- fr.upem.matou.server.network : contient l'implémentation réseau du serveur
- fr.upem.matou.shared : package nécessaire au client et au serveur
- fr.upem.matou.shared.logger : contient le logger
- fr.upem.matou.shared.network : contient la base du protocole de communication réseau
- fr.upem.matou.shared.utils : regroupe des classes et méthodes utilitaires

Responsabilité des classes

Voici la liste des différentes classes du service ainsi que leurs propriétés respectives :

- package "client"
  - ClientMatou : C'est le point d'entrée du client.
- package "client.network"
  - ClientCommunication : Classe composée de méthodes statiques utilisées par le client pour garantir le respect du Protocol de communication.
  - ClientCore : C'est la classe responsable d'un client de chat sur un serveur. Cet objet est lié à une seule interface utilisateur et un seul serveur, mais il peut être utilisé pour lancer plusieurs instances de chat.
  - ClientInstance : C'est la classe responsable d'une session de chat active avec une serveur.
  - ClientSession : C'est la classe responsable de gérer les connexions avec l'extérieur pour une session de chat active.
  - ClientEvent : C'est une interface permettant d'exécuter une action sur un objet ClientSession.
  - Message : Représente un message reçu.
  - SourceConnectionData : Représente les informations de connexion privée d'un autre client lorsque l'on est en mode "réception" d'une connexion privée.
  - DestinationConnectionData : Représente les informations de connexion privée d'un autre client lorsque l'on est en mode "émission" d'une connexion privée.
- package "client.ui"
  - UserInterface : C'est une interface responsable de gérer les interactions entre l'utilisateur et le programme. Les implémentations de cette interface doivent être transparentes et thread-safe.

    - ShellInterface : C'est une implémentation de UserInterface qui utilise le terminal.
    - ShellCommand : Classe qui regroupe des méthodes statiques permettant de parser une commande de ShellInterface en un ClientEvent.
- package "server"
    - ServerMatou : C'est le point d'entrée du serveur.
- package "server.network"
    - ServerCommunication : Classe composée de méthodes statiques utilisées par le serveur pour garantir le respect du protocole de communication.
    - ServerCore : C'est la classe responsable d'un serveur de chat.
    - ServerDataBase : C'est une classe unique pour une instance du serveur de chat qui regroupe les informations communes à tous les clients connectés.
    - ServerSession : C'est une classe unique pour chaque client connecté au serveur qui regroupe les informations relatives à l'état actuel de ce client.
    - SelectorLogger : C'est le logger du sélecteur.
- package "shared.network"
    - NetworkCommunication : Classe composée de méthodes statiques utilisées par le client et le serveur pour garantir le respect du Protocol de communication. C'est dans cette classe que sont définies la plupart des règles à respecter.
    - NetworkProtocol : Enumération des différents types de requête. Chaque type de requête est identifié par un entier unique.
    - ErrorType : Enumération des différentes erreurs d'utilisation que le serveur peut envoyer au client.
    - Username : Représente un nom d'utilisateur. Un nom d'utilisateur est une chaîne de caractère identifiée par sa représentation en minuscule.
- package "shared.logger"
    - Logger : C'est le logger d'événements du service. Il n'y a qu'une instance de cet objet à l'exécution, il est manipulé avec des méthodes statiques.
    - Colorator : Classe qui regroupe des méthodes statiques permettant de formater des chaines de caractère afin qu'elles s'affichent en couleur sur le terminal.
- package "shared.utils"
    - ByteBuffers : Classe qui regroupe des méthodes statiques sur les objets de type "ByteBuffer".
    - Configuration : Classe qui regroupe des méthodes statiques pour lire un fichier de configuration.

Choix de développement

Exécution personnalisée

Le programme peut être lancé dans plusieurs configurations différentes en fonction des besoins de l'utilisateur.

Le client et le serveur disposent de 2 options en ligne de commande.
Le client peut également indiquer un pseudo facultatif à la fin de la ligne de commande afin d'abréger le processus de connexion.

En outre, le client et le serveur disposent également d'un fichier de configuration. Cette alternative permet de régler certains paramètres sans avoir à modifier le code source ni à saisir une ligne de commande à rallonge.
Le fichier de configuration du client possède 6 champs modifiables par un booléen.
Le fichier de configuration du serveur possède 7 champs modifiables par un booléen.

Gestion des erreurs et des exceptions

Les problèmes qui peuvent survenir durant l'exécution du serveur ou du client sont signalés à l'utilisateur via un objet "Logger".

Le logger dispose de deux sorties : une sortie pour les messages de base et une sortie pour les exceptions.
La sortie de base est utilisée pour logger les événements suivants, du plus grave au plus bénin : erreur, avertissement, information, message de debug.
La sortie d'exception est utilisée pour logger les exceptions critiques.
Il est possible de rediriger chacun de ces flux de sortie vers un fichier pour garder des traces d'une exécution. Pour cela, il faut indiquer une option sur la ligne de commande du programme.

Le logger dispose de deux niveaux de détail d'affichage :
- Un affichage détaillé indiquant le niveau de gravité de l'événement, la date et l'heure où il a été inscrit, le thread ayant appelé le logger ainsi que le message associé à l'événement. Pour les interactions réseau, il y a également des informations complémentaires sur la SocketChannel.
- Un affichage épuré indiquant uniquement le niveau de gravité de l'événement ainsi que le message associé.
Le niveau de détail peut être modifié dans les fichiers de configuration (HEADER).

Le logger dispose également d'un mode "couleur" où chaque type d'événement est coloré de manière différente sur les terminaux compatibles.
Cette option peut être activée ou désactivée dans les fichiers de configuration (COLORATOR).

Implémentation du serveur

Le serveur est implémenté en mode TCP non bloquant.

Le serveur alloue un unique buffer d'écriture global puis deux buffers par client (un pour la lecture et un pour l'écriture).
La taille du buffer de lecture d'un client est générée à partir de la taille maximum d'un argument d'une requête de protocole que le serveur peut recevoir. Ce buffer ne peut pas déborder.
La taille du buffer d'écriture d'un client est générée à partir de la taille maximum d'une requête de Protocol que le serveur peut envoyer. Elle est multipliée par un certain coefficient afin d'avoir une marge suffisante dans le cas où le buffer se remplit plus vite qu'il ne se vide. Si l'ajout d'une requête à ce buffer n'est pas possible (débordement du buffer), la requête n'est pas ajoutée au buffer (elle est perdue pour ce client).
La taille du buffer d'écriture global est générée à partir de la taille maximum d'une requête de protocole que le serveur peut envoyer. Ce buffer est vidé immédiatement après avoir été rempli, il ne peut donc pas déborder.
Cette implémentation était selon nous la plus judicieuse afin de ne pas avoir à multiplier les allocations ni à gérer une liste de messages en attente qui peut augmenter indéfiniment.

Le serveur lit les requêtes argument par argument dans le buffer de lecture, en positionnant la limite de celui-ci à la taille attendue pour les données à lire.
Le serveur ne traite l'argument que lorsqu'il a complètement été reçu.
Le serveur ne traite la requête que lorsque tous ses arguments ont été reçus.
Le contrôle de validité des arguments est effectué pendant et après la lecture de la requête.

Le serveur dispose également d'une limite de connexion : au delà de cette limite, les nouvelles de connexions sont fermées tout de suite après avoir été acceptées.
Cette fonctionnalité permet d'éviter que le serveur ne plante parce qu'il n'a plus de mémoire.
A noter que les connexions sont quand même acceptées pour ne pas accumuler les connexions pendantes. Si les connexions pendantes n'étaient pas traitées, le serveur risquerait de devoir traiter plein d'anciennes demandes de connexion même si elles ne sont plus actives.

Le serveur envoi une notification à tous les clients authentifiés dès qu'une personne se connecte ou se déconnecte du chat public.

Implémentation du client

Le client est implémenté en mode TCP concurrent.

Le client démarre :
- 1 seul thread global pour l'écriture des requêtes
- 1 seul thread pour la lecture des requêtes du serveur
- 1 thread par connexion privée pour la lecture des messages privés
- 1 thread par connexion privée pour la lecture des fichiers

D'autres threads intermédiaires sont parfois créés pour une tâche spécifique (par exemple : l'envoi d'un fichier).

Si le client reçoit une requête invalide ou si la connexion est fermée, une IOException est levée et tue le thread chargé de recevoir les requêtes pour le canal concerné.

L'exécution du client se termine lorsque la session de chat publique prend fin.
La fermeture de la connexion publique implique la fermeture de toutes les connexions privées.

Gestion des pseudos

Les pseudos ne sont pas représentés par un objet String mais par un objet Username qui redéfinit les méthodes equals et hashCode afin d'ignorer la casse.
Cette implémentation permet de garantir l'unicité du pseudo d'un utilisateur connecté tout en conservant sa typographie.
Toutes les comparaisons sur les pseudos sont faites à partir de cet objet.

De plus, un pseudo ne peut contenir que des caractères alphanumériques et sa taille encodée ne doit pas dépasser 32 octets.

Gestion des messages

Un message est représenté par un objet String par le serveur et par un objet Message par le client au moment de la réception.

De plus, un message ne peut contenir aucun caractère de contrôle et sa taille encodée ne doit pas dépasser 512 octets.

Connexion privée

Les connexions privées fonctionnent avec des demandes de connexion. Il existe deux types de demandes :
- La demande d'ouverture : Le client SOURCE demande une connexion privée à un autre client DESTINATION. Le client DESTINATION recevra une notification indiquant que SOURCE souhaite se connecter à lui.
- La demande d'acceptation : Le client DESTINATION accepte la demande de connexion privée au client SOURCE. Le client SOURCE doit avoir envoyé une demande d'ouverture précédemment, sans quoi cette demande d'acceptation ne peut aboutir.

Lorsque la connexion privée est validée, l'utilisateur SOURCE devient le serveur de la connexion privée et l'utilisateur DESTINATION devient le client de la connexion privée.
Le serveur envoi les informations nécessaires aux deux partis.
Le client SOURCE vérifie l'identité du client DESTINATION à partir de son adresse IP.

Les connexions privées sont prises en charge pour les adresses IPv4 et les adresses IPv6.

La connexion privée est établie sur deux ports : un port pour la réception des messages et un port pour la réception des fichiers.
La fermeture de la connexion privée engendre la fermeture des deux canaux associés à la connexion. Si un problème survient lors de la communication privée, les deux canaux sont fermés indistinctement du canal à l'origine du problème.

Lorsqu'un fichier est reçu par connexion privé, il est enregistré dans le répertoire "files" du répertoire courant. Si ce répertoire n'existe pas, la réception du fichier échoue et la connexion privée est interrompue. En dehors de cela, il n'est pas possible de refuser la réception d'un fichier.
Lorsqu'un fichier a été intégralement reçu, l'utilisateur en est notifié par l'interface utilisateur avec le chemin menant au fichier. Cette notification est la seule garantie que le fichier a été reçu dans son intégralité.
Le nom d'un fichier reçu est généré à partir du pseudo de l'utilisateur ayant envoyé le fichier, d'un identifiant et du nom d'origine du fichier.
Le fichier est envoyé et reçu par tronçon de 4 Kio. La taille du fichier est encodée sur 8 octets, il est donc possible d'échanger des fichiers très volumineux.

Conclusion

Objectifs réalisés

Notre programme répond aux objectifs demandés par le sujet et respecte les consignes imposées.

Points forts

Voici les principaux avantages de notre projet :

- Le nombre d'allocation mémoire côté serveur est réduit.
- Il est possible de lancer plusieurs sessions de chat avec un même objet ClientCore.
- Le projet ne dépend pas d'une seule interface utilisateur. Il suffit pour un objet d'implémenter l'interface UserInterface afin d'être utilisée comme interface utilisateur pour le client.
- Le programme arrête proprement les threads lors de l'arrêt d'une session de chat.
- Les pseudos sont insensibles à la casse.
- Forte abstraction des buffers côté client grâce à l'encapsulation des ByteBuffer dans des méthodes readInt/writeInt, readString/writeString...

Points faibles

Voici les principaux défauts de notre projet :

- L'architecture est compliquée, autant côté serveur (avec 2 gros objets qui ont une dépendance circulaire) que côté client (où une session nécessite 2 objets).
- Le client est gourmand en ressources (nombreux threads et nombreuses allocations dues à l'abstraction mise en place vis à vis des buffers).

Difficultés rencontrés

Voici les principales difficultés que nous avons rencontré durant ce projet :

- Gestion des buffers côté serveur
  - Contrairement au client qui peut abstraire l'utilisation des buffers, le serveur repose intégralement sur la pérennité de ces buffers lors d'une même session.
- Gestion des événements du client
  - Nous avons eu beaucoup de mal à implémenter le système d'événements pour faire communiquer l'interface utilisateur et le cœur du programme. N'ayant eu aucun cours d'interface graphique dans notre formation universitaire, nous avons improvisé une architecture basée sur des objets "événements" qui prennent en paramètre le contexte sur lequel ils doivent être exécutés. Cette implémentation détournée du Design Pattern "Visiteur" est la meilleure que nous ayons trouvée pour le moment.

Dysfonctionnements possibles :

Nous avons effectué plusieurs tests sur notre programme. Voici les fonctionnements anormaux que nous avons remarqué et que nous n'avons pas pu pleinement corriger :

- Si un client A est connecté sur la même machine X que le serveur et qu'un autre client B connecté sur une autre machine Y essaye d'établir une connexion privée avec A, la connexion privée ne pourra pas être toujours être établie.
  - Ce dysfonctionnement intervient uniquement lorsque le client A se connecte au serveur en utilisant l'adresse de rebouclage du serveur (localhost / 127.0.0.1 par exemple). Le serveur va identifier ce client A avec cette même adresse et l'enverra au client B. Mais comme cette adresse est spécifique à la machine X du client A, le client B ne pourra pas s'y connecter depuis sa machine Y.
- Un client A est connecté depuis une machine X et un client B est connecté depuis une machine Y. Le client A veut ouvrir une connexion privée avec le client B. Si un autre client C connecté depuis la machine Y connait les ports de connexion privée de A, alors il pourra se connecter à A en se faisant passer pour B.
  - En effet, les clients B et C sont sur la même machine donc ils seront potentiellement identifiés par la même adresse IP. C'est une problématique de sécurité, le mécanisme de vérification d'identité se limite aux adresses IP et à la transmission des numéros de port.
- Lorsqu'une instance de chat est interrompue, le programme ne s'arrête pas toujours immédiatement. Il faut parfois que l'utilisateur tape quelque chose sur le terminal pour que l'exécution s'arrête.
  - Ce dysfonctionnement est dû au fait que ShellInterface utilise un objet Scanner pour lire l'entrée standard. Lorsque le thread responsable d'envoyer les événements est bloqué sur l'attente d'un nouvel événement, le Scanner attend de lire une ligne sur l'entrée standard.

Malheureusement cette méthode ne peut pas être interrompue, c'est pourquoi le thread continue son exécution.

- Lorsqu'un ferme le flux d'entrée standard du programme, toutes les instances de chat d'un ClientCore sont interrompues.
    - Bien que ça ressemble à un dysfonctionnement, c'est une situation normale. Si plusieurs ClientInstance sont lancées sur un même objet ClientCore, chaque session de chat utilisera la même interface utilisateur et donc le même terminal. Si le flux du terminal a été fermé brusquement, il est normal que toutes les sessions associées ne puissent poursuivre leur tâche.
- Les pseudos ne sont pas toujours insensibles à la casse.
    - Ce problème n'apparait qu'avec des lettres où les majuscules et les minuscules ne sont pas équivalentes (par exemple, l'alphabet géorgien). Il est dû au fait que les pseudos sont comparés entre eux par leur représentation en minuscule.

Perspectives d'améliorations

Voici quelques perspectives d'améliorations et d'optimisations possibles pour le projet. Il s'agit généralement d'idées que nous avions eu et que nous n'avons pas eu le temps de mettre en place ou bien d'améliorations qui nous sont venus trop tard dans la réalisation du projet :

- Pouvoir déconnecter du serveur tous les clients qui sont inactifs depuis trop longtemps. Le temps d'inactivité serait différent en fonction de l'état "Authentifié" ou "Non authentifié" du client.
- Pouvoir refuser la réception d'un fichier.
- Pouvoir annuler une demande d'ouverture de connexion privée avant que celle-ci ne soit acceptée.
- Développer une interface graphique.

Annexes

Références

Pour concevoir l'objet "Colorator", nous nous sommes inspirés de :

- **Wikipedia** : https://en.wikipedia.org/wiki/ANSI_escape_code
- **Stack Overflow** : http://stackoverflow.com/questions/5762491/how-to-print-color-in-console-using-system-out-println