ChatHack
Client et serveur

Samy WADAN – Armand LIEGEY

Serveur

Serveur

Création et initialisation

- On instancie un serveur avec **ChatHackServer(int port)**
  - Connecte le serveur au port, le configure en non bloquant et initialise le sélecteur.
  - Se connecte au serveur de base de données sur le port 7777 (en dur).
- On démarre le serveur avec **ChatHackServer.launch()**
  - La socketChannel s'enregistre dans le sélecteur en OP_ACCEPT.
  - La boucle du serveur se lance tant que le thread est en vie.
- On le stop avec **ChatHackServer.stop().**
  - L'état des paramètres n'est pas réinitialisé.
  - Déconnecte tous les clients.
  - Il faudra le relancer avec ChatHackServer.launch().

Serveur

Handler des évènements (treatKey)

- La boucle consiste à sélectionner les clés prêtes pour une opération surveillée pour exécuter une action.
  - doAccept(), doConnect() (pour la BDD), doWrite(), doRead().

- Lorsqu'un client se connecte, le serveur l'accepte dans **doAccept()**.
  - On enregistre la socketChannel du client dans le sélecteur en *OP_READ*.
  - On attache à cette clé un nouvel objet de type **ServerContext**.

- Seul les clés correspondant aux clients et à la base de données peuvent être surveillées en *OP_READ* et *OP_WRITE*
  - Les méthodes **doRead()** et **doWrite()** se font dans le **ServerContext**.

Serveur

Gestion des exceptions

- Si une **IOException** est levée depuis l'objet ServerContext
  - On ferme la connexion avec le client associé ou la base de données.
- Sinon
  - On ferme le serveur

Serveur

Interaction utilisateur

- Un autre thread est créé pour la saisie utilisateur.
  - Une boucle infinie sur la lecture de l'entrée standard.

- Certaines commandes permettent d'agir sur le serveur.
  - **Info** : Informe sur le nombre de clients connectés.
  - **Shutdown** : Empêche des nouveaux clients de se connecter.
  - **Shutdownnow** : Stop le serveur et déconnecte tous les clients

Contexts

Contexts

Généralités

- Il y a 2 types de Context :
  - Quand on se situe sur le serveur, on attribue un **ServerContext** à un client qui se connecte, et à la base de données.
  - Quand on se situe sur le client, on attribue un **ClientContext** au serveur « publique » et aux clients « privés ».
- Ces deux types de Context ont une base commune : **AbstractContext.**
  - Cette classe abstraite dont hérite ClientContext et ServerContext est le seul endroit qui manipule les buffeurs d'entrée/sortie. Elle possède notamment la méthode **processIn()** qui déchiffre une trame et exécute son code, et la méthode **queueMessage()** qui envoie une trame au socketChannel relié.
- Les ClientContext et ServerContext possèdent de plus les méthodes propres à l'interaction avec le client/le serveur
  - Pour le client : Connexion à un autre client, affichage des messages, envoi de fichiers…
  - Pour le serveur : Authentification vers la base de données, diffusion des trames…

Réception d'un message

doRead()
→ Récupère les données du buffeur d'entrée

ProcessIn()
→ FrameReader.process()
→ Si DONE alors
→ Frame frame = FrameReader.Get()
→ Exécuter le code associé à la frame
→ Sinon
→ Continuer de lire le buffeur

updateInterestOps()
→ Met à jour les opérations à surveiller

Méthode publique

QueueMessage(Frame frame)
→ Ajoute frame à la Queue

doWrite()
→ Envoie les données du buffeur de sortie

ProcessOut()
→ Remplis le buffeur de sortie depuis la Queue de message

Frames et Readers

Frames et Readers

Généralités

- Chaque trame est représenté par un objet **Frame**
- A chaque frame lui est associé son **Reader**
  - Chaque Context possède un **FrameReader**
- Chaque frame débute par un opcode, ainsi il suffit de lire le 1er octet de la trame pour savoir quel Reader utiliser.
  - C'est le rôle du FrameReader, il lit le 1er octet puis appel un autre reader correspondant à la trame.
  - Ceci est faisable car les opcodes sont uniques. Sauf pour les trames du serveur de la base de données, ainsi le Context associé à la base de données possède son Reader spécifique : **BddReader**.

Frames et Readers

Frames

- Une frame implémente l'interface Frame et possède les 3 méthodes suivantes :
  - Size() → Renvoie la taille de la trame en octets
  - getBytes() → Renvoie le contenue de la trame sous la forme d'un tableau d'octets
  - Accept() → Exécute le code associé à la frame
- Cette méthode accept() est surchargée, elle prend soit un ClientContext un ServerContext en paramètre.
  - Ceci permet à l'objet Frame d'interagir sur le client ou le serveur. Cette méthode est appelée dans le processIn() du Context.
- De plus, chaque frame est datée par sa date d'instanciation.
  - Ceci permet de déterminer entre 2 frames laquelle a été crée en premier, utilise pour la gestion des priorités, cf prochains slides.

Client

Client

Création et initialisation

- On instancie un client avec **ChatHackClient(InetSocketAddress server, String login, String password)**.
  - Ouverture du socketChannel, création des buffeurs, du sélecteur, de la Queue et du FrameReader.

- On le lance avec **ChatHackClient.launch()**.
  - (Ré)initialise les paramètres (buffeurs, queue, FrameReader).
  - Place le socketChannel dans le sélecteur en *OP_CONNECT*.
  - Lance un nouveau thread exécutant la boucle de selection.
  - Appeler launch() quand le client tourne déjà n'a pas d'effet.

- On le stop avec **ChatHackClient.stop().**
  - L'état des paramètres n'est pas réinitialisé.
  - Déconnecte les clients "privés".
  - Il faudra le relancer avec ChatHackClient.launch().

Client

Handler des évènements

- Lorsque la connexion avec le serveur est possible, **doConnect()** termine la connexion et envoie une trame d'authentification avec ses identifiants.
  - Cela se fait en appelant queueMessage() de la même façon qu'avec le serveur.
  - updateInterestOps() est ensuite appelé, retirant *OP_CONNECT* et ajoutant *OP_WRITE* et *OP_READ*.

- Le client reste en écoute du serveur, et passe en écriture lorsqu'une saisie a lieu, c'est à dire lorsque l'utilisateur souhaite envoyer une trame.
  - Plutôt que de réveiller le selecteur avec Selector.wakeUp(), la selection se fait au plus tard toutes les 100ms.

Client

Interaction utilisateur

- L'utilisateur peut envoyer des trames avec le clavier. Il faut pour cela utiliser l'objet **UserInput(ChatHackClient client)** qui prend le client en paramètre.

- L'interaction se fait avec **UserInput.interact()**.
  - Méthode bloquante, elle boucle sur la lecture de l'entrée standard.

- Les chaines de caractères saisies sont parsées et si le parsing à réussis un objet Frame correspondant est créé et envoyé au destinataire.
  - Toujours avec queueMessage().

Client

Serveur privé

- Un client démarre son propre serveur lorsqu'il en a besoin, c'est à dire lorsqu'un autre client demande d'effectuer une communication privée.
  - La clé de ce serveur privé est **inscrit dans le selecteur**. Ainsi le selector gère la clé du serveur publique, du serveur privé, ainsi que les clés des clients privés.

- Voici comment s'exécute l'établissement de la communication privée :
  - A envoie une demande de communication privé à B, en passant par le serveur publique.
  - Si A continue d'envoyer des messages (ou des fichiers) avant que B ne réponde, ils sont placés dans une file d'attente (pour les fichiers, uniquement leur nom est enregistré).
  - Si B refuse, A détruit ces files d'attentes. Sinon : B crée son serveur privé (si pas déjà fait), associe à A un identifiant et le lui envoie ainsi que son port et son adresse IP.
  - A peut maintenant se connecter à B, mais doit lui envoyer une trame d'authentification contenant son nom et son identifiant qui lui a été attribué.
  - B peut maintenant associer un ClientContext à A et l'enregistre en tant que client privé.
  - A enregistre B en tant que client privé et envoie ses messages (et fichiers) mis en attente à B.

Client

Serveur privé

- Comme pour le serveur, si un client envoie des messages/fichiers alors qu'il n'est pas authentifié, ces derniers sont ignorés.

- Vu qu'une connexion directe est établie entre 2 clients, couper le serveur publique n'empêche pas les clients de communiquer entre eux.

Client

Envoi des fichiers

- Lorsqu'un client souhaite envoyer un fichier à un autre client, **un nouveau thread** démarre.
  - L'envoi pouvant durer un certains temps, il ne faut pas bloquer le reste du client.

- Ce thread lit le fichier et envoie les trames au destinataire **au fur et à mesure**. Il prépare en faite 500 trames de chacune 1024 octets de données, et les place dans la file d'envoi lorsqu'elle est vide, pour ne pas la surcharger.
  - On envoie **des salves de 500 trames** et pas une par une car la lecture d'un fichier est couteux, on minimise donc l'accès au disque tout en évitant de surcharger la file également.
  - Pour faire des salves, le thread d'envoi s'interrompt lorsque la file n'est pas vide. Elle est réveillée par un signal depuis la méthode processOut(). C'est pourquoi cette méthode est redéfinie dans le ClientContext.

Client

Envoi des fichiers

- Pendant l'envoi du fichier, la file d'envoi n'est quasiment jamais vide et est toujours alimentée par potentiellement beaucoup de Frames.
  - ➔ Ceci a pour conséquence que si le client souhaite envoyer un message au client téléchargeant le fichier, il devra d'abord attendre que tout le morceau de fichier soit envoyé avant de pouvoir envoyer son message.
- Solution : **Prioriser** les trames messages dans la file.
- La file d'envoi des Frames est implémentée par une **PriorityBlockingQueue**.
  - Elle permet de rendre des trames plus prioritaire que d'autres.
  - Pour cela il faut rendre les Frames comparable entre elles.
  - → Il y a 2 niveaux de priorité : les Frames de fichiers et les autres frames. Une Frame de fichier est moins prioritaire que les autres. Il faut maintenant déterminer la priorité entre 2 Frames de même niveau.
  - → C'est là qu'intervient la date de création des Frame : une Frame plus ancienne est plus prioritaire qu'une frame plus récente. Le datage se fait en nanoseconde, il n'arrive donc jamais que 2 Frames aient la même date.

Client

Réception des fichiers

- Le destinaire doit reconstituer le fichier.
  - Pour cela il faut savoir différentier la première trame, de la dernière et des autres.
- La première trame donne l'information sur la taille du fichier et son nom.
  - Ainsi le client peut créer son fichier (vide)
- Le client le remplis avec les données qu'il reçoit, puis le ferme lorsqu'il a reçu la dernière trame, c'est à dire que la quantité de données qu'il a reçu est égal à la taille du fichier.
- Un objet **FileContext** permet de gérer ce téléchargement.
  - Le FileContext n'a rien à voir avec les autres Context vue précédemment.

Divers

Divers

Gestion des logs

- La plupart des informations, plus ou moins utiles, sont loggées.
- Il y a 2 loggers : Un pour le client et un pour le serveur publique.
- Pour ne pas encombrer la sortie standard, les logs sont redirigés dans un fichier *client_log.log* ou *server_log.log*

Divers

Dépendances

- Le projet contient 4 modules :
  - Les Frames/Readers (Resources)
  - Les Contexts (incluant le FileContext)
  - Le client
  - Le serveur
- Ce découpage permet de pouvoir rajouter/modifier/supprimer des Frames facilement et sans impacter le programme.
  - Un petit soucis se pose quand même car une Frame a besoin d'un Context pour pouvoir agir dessus.
  - Pour éviter des inclusions circulaires, le projet Resources possède une interface Client et Server permettant de faire la liaison entre les deux, sans que Resources n'inclue Context.
- Ainsi il est très facile de changer complètement de protocole en conservant cette architecture.
  - Il suffit de changer les trames dans le projet Resources et éventuellement modifier les fonctions des Contexts pour faire autre chose.

Divers

Avancement du projet

- Un git a été créé pour développer ce projet et est disponible ici : https://github.com/SaladTomatOignon/ChatHack