Good Duck Transfert Client and Server

Étienne Marais - Benjamin Viau

Sommaire

Protocole

Le répertoire suivant contient le code qui implémente le protocole GDPT pour le côté serveur et pour le côté client. Le fichier contenant le protocole se trouve ici : docs/rfc.txt.

Prérequis

Votre système doit posséder les outils suivant installés :

- Java 11 ~openjdk11
- Make

Informations sur la lecture du manuel

Dans le manuel, nous allons distinguer le terminal de l'invite de commandes du client. Nous utiliserons les symboles suivants pour faire la distinction entre les deux :

Pour le shell :

```
$ <command>
```

Pour l'invite de commande du client :

```
>> <command>
```

Compilation

Pour compiler, il faut se placer à la racine du dossier, là où se trouve le **Makefile**.

Serveur

Pour compiler et lancer le serveur sur le port **1027**, il faut exécuter la commande suivante :

```
$ make server
```

Pour le lancer sur un autre port, il faut faire comme suit :

```
$ make server SERVER_PORT=<port>
```

Client

Pour compiler et lancer le client sur le port **1027** et sur l'adresse **127.0.0.1** :

```
$ make client
```

Pour lancer le client sur une autre adresse et sur un autre port, il faut lancer avec :

```
$ make client GTDP_addr=<addr> GDTP_port=<port>
```

Par défaut, le client n'affiche pas les paquets reçus pour permettre une meilleure lisibilité à l'utilisateur. Il est cependant possible de les voir grâce au paramètre suivant :

```
$ make client DEBUG=yes <options>
```

Nous avons aussi mis en ligne un serveur accessible de n'importe où qui stocke des annonces depuis le 2 Novembre. Pour vous y connectez, vous devez taper la commande suivante :

```
$ make client GDTP_addr=psi.maiste.fr
```

Nettoyer le répertoire

Afin d'éliminer les **.class* il est possible d'utiliser la commande suivante :

```
$ make clean
```

Fonctionnement

Serveur

Une fois lancé, le serveur fonctionne de façon autonome et affiche les logs de son exécution. Il peut être arrêté grâce à un **CTRL+C**.

Client

Interface L'interface client se compose de deux onglets:

- Le terminal
- Le chat

Vous pouvez naviguer entre les deux onglets grâce aux flèches directionnelles et à la touche Tab. Toutes les commandes doivent être rentrées dans la partie *Terminal*. Pour la partie *chat*, une fois que vous avez essayé de communiquer avec un pair via la commande talk, il est possible de voir la liste des pairs avec la flèche en haut à droite. Le pair indiqué est celui avec qui vous êtes en train de discuter.

Aide Pour afficher l'aide dans l'interpréteur, il faut utiliser la commande suivante :

```
>> help
```

Connexion Le client fonctionne en ligne de commandes une fois lancé. Il faut d'abord se connecter avec un identifiant pour pouvoir effectuer sa première connexion au serveur en utilisant la commande suivante dans l'interpréteur du client :

```
>> connect [USERNAME]
```

Pour les connexions suivantes, vous pouvez simplement effectuer la commande suivante:

```
>> connect
```

En effet, un token est créé et ajouté au répertoire *$HOME* lors de la première connexion dans **$HOME/.config/gdtp/token**. Si vous voulez changer d'utilisateur, il faut refaire la manipulation avec le connect [USERNAME].

Quitter Pour quitter le logiciel et vous déconnecter, vous pouvez utiliser la commande suivante :

```
>> exit
```

Domaines Pour afficher les domaines disponibles sur le serveur, il faut faire la commande suivante :

```
>> domains
```

Annonces Pour obtenir toutes les annonces d'un domaine, il faut utiliser la commande suivante :

```
>> ancs [DOMAINE]
```

Propres annonces Pour obtenir l'ensemble des annonces que vous avez postées, il faut taper la commande suivante :

```
>> own
```

Création Pour poster une nouvelle annonce, il faut lancer l'éditeur intéractif :

```
>> post
```

Le prix s'écrit avec le format suivant : `13.50` correspond à 13.50€.

Mise à jour Pour mettre à jour une annonce, il faut écrire :

```
>> update [ID ANNONCE]
```

Suppression Pour supprimer une annonce sur le serveur qui vous appartient, il faut lancer la commande suivante :

```
 >> delete [ID ANNONCE]
```

Récupération de l'IP Il est possible de récupérer l'ip d'un collaborateur pour le contacter directement. La commande à utiliser est la suivante :

```
>> ip [ID ANNONCE]
```

Ouvrir le chat

Il est possible d'ouvrir une discussion avec quelqu'un qui est connecté sur le serveur de discussion grâce à la commande suivante :

```
  talk [ID ANNONCE]
```

Discuter avec quelqu'un de connecté

Une fois que l'action `talk` a été faite, vous pouvez utiliser l'onglet de chat pour discuter avec la personne que vous voulez contacter. Cependant, si celle-ci se déconnecte, elle ne recevera plus vos messages et ceux- ci seront perdus.

Détail de l'implémentation

Nous allons ici vous parler des détails de notre implémentation et des choix que nous avons faits qui ne sont pas précisés dans le protocole. Nous allons aussi décrire les classes que nous avons définies et leur utilité. Nous avons fait le choix de répartir notre code dans les trois packages suivants :

Parties communes

Nous avons fait le choix dans notre implémentation de ne stocker en mémoire que des chaînes de caractères pour permettre une réponse plus rapide du serveur. Les vérifications sont effectuées à l'insertion des données.

Logs.java

Il s'agit de la la classe la plus simple. Afin de fournir aux classes du client et du serveur un système permettant de débogguer le code facilement, nous avons implémenté cette classe qui repose sur le Logger fournit par Java. Il permet d'avoir un système lisible qui fournit une information sur la position de l'erreur dans le code, une date et une distinction des niveaux de criticité des erreurs.

Message.java

Les messages échangés pour le protocole GDTP sont représentés comme un type de messages avec son tableau d'arguments. Les types des messages sont définis au sein d'une énumération. Il est donc très simple de rajouter un nouveau type d'en-tête reconnu par le serveur. Comme indiqué au dessus, les arguments sont représentés comme de simples chaînes de caractères. Cette classe a vocation à gérer le passage d'une chaîne de caractères produites par TCP en un message compréhensible par le serveur et vice-et-versa.

Domaine.java

Cette classe est là pour représenter les domaines accessibles depuis le serveur. Comme les messages, les domaines sont représentés par une énumération afin d'être facilement extensible. En effet, le serveur se sert de l'énumération pour constuire l'arbre qui stocke les domaines.

Les domaines de chaque serveur ne sont pas définis. Il est libre pour chaque groupe de choisir les domaines que son serveur offre. Dans notre cas, nous avons repris les grands domaines du site leboncoin.fr. Ils sont toujours stockés sous forme de lettres majuscules pour notre serveur afin de ne pas faire de distinction sur la casse.

Annonce.java

Les annonces sont définies par leur domaine, leur titre, leur description et leur prix. Ces quatre arguments sont des chaînes de caractères pour ne pas avoir à effectuer de conversion à l'envoi. En outre, des vérifications sont effectuées sur sur les arguments pour s'assurer qu'aucun n'est vide et que le prix est bien formaté comme défini par Java. Cette classe ne gère que l'invariant qui indique qu'aucun des champs ne doit être à *null*. Les objets ayant une unique signature en mémoire, l'id de l'objet est créé par concaténation de l'utilisateur avec son id d'objet.

Index.java

Il s'agit d'une des classes les plus importantes pour le serveur. En effet, c'est elle qui assure la cohérence entre les utilisateurs. Il s'agit d'une classe où les accès se font en concurrence. Pour cela, chacune des

méthodes appelées est verrouillée par le mot clef *synchronized* afin de limiter l'accès concurrent. Pour permettre aux Threads de l'utiliser, cette classe est conçue comme un singleton et n'existe donc qu'une fois en mémoire. Elle conserve et assure la cohérence des données utilisateurs et vérifie les invariants suivants :

- Un utilisateur n'est connecté qu'avec une ip en même temps.
- Il n'existe qu'un utilisateur avec le même nom.
- Aucun token de connexion n'existe en même temps plusieurs fois dans la mémoire.

Pour venir à bien de sa mission, on trouve dans l'index, 2 structures de données :

- Une table de hachage, users, contenant la liste des utilisateurs qui se sont déjà connectés une fois. Elle associe à chaque utilisateur un token de connexion qui doit nous servir pour assurer la partie sécurité plus tard.
- Une table de hachage, cache, qui contient pour chaque utilisateur l'adresse IP à laquelle il est actuellement connecté. Elle est supprimée lors de la déconnexion de l'utilisateur.

Ainsi la récupération des données courantes se fait en temps constant et pour les autres en temps linéaire sur la taille de la table de hachage.

StorageAnnonce.java

Cette classe s'occupe de sauvegarder en mémoire les données des annonces. Les domaines sont stockés sous forme de noeud d'un arbre et chaque noeud indexe une table de hachage où les annonces sont indexées par id d'annonce. Comme il s'agit d'une classe qui fonctionne en concurrence, elle a les mêmes propriétés que l'index. Ses méthodes empêchent les accès concurrents via *synchronized* et il s'agit d'une classe de type singleton. En outre, cette organisation des données permet de trouver une annonce en temps O(nlog n) et de d'ajouter des données en temps logarithmique. Cette classe vérifie les invariants suivants :

- Chaque domaine possède une table de hachage regroupant ses annonces, celle-ci est potentiellement vide.
- Chaque annonce est insérée dans une table de hachage du domaine qui lui correspond.
- On ne peut pas insérer une annonce qui existe déjà.
- Un utilisateur ne peut que modifier ou supprimer une annonce qui lui appartient.

LetterBox.java

Il s'agit d'une structure de données qui gère tous nos messages reçus et envoyés. Cette structure est résiliante à la concurrence c'est pourquoi toutes ses méthodes sont *synchronized*. Elle contient une liste des messsages reçus par utilisateur ainsi qu'une liste des messages envoyés, indexés par des timestamps. Lorsque l'on reçoit un message de la part d'un utilisateur si c'est un acquittement, le message associé est retiré de la liste des messages à envoyer. Sinon, on l'ajoute à la liste des messages reçus.

PeerList.java

Elle contient les adresses IPs associées aux utilisateurs que l'on a déjà contactés. Comme pour les autres structures, elle possède toutes ses méthodes en *synchronized*. À son initialisation, elle lance un *GarbageCollector* qui supprime les adresses que l'on a pas rencontrées depuis une heure afin d'alléger la mémoire.

Client

L'implémentation du client s'articule autour de 4 modules.

GDTService.java

Il s'agit d'une classe qui s'occupe de communiquer directement avec le serveur. Sa fonction principale est `askFor`, elle demande un message en entrée et renvoie un autre message (la réponse) de manière

asynchrone. En effet, le message reçu sera encapsulé dans un objet Future. Un future est un objet qui contient la promesse d'une valeur. On lui fournira alors une lambda (Un Consumer) qui implémentera que faire une fois la valeur promise calculée. De cette façon, on s'affranchit de gérer nous-mêmes les threads.

DataProvider.java

Cette classe s'occupe de traiter les réponses reçues (par exemple afficher les réponses). De la même manière que HTTP, le client n'attend du serveur que des réponses à ses requêtes. D'où l'existence de la fonction `basicRequest` qui permet de synthétiser l'idée du protocole. Elle prend un message d'envoi en argument, ainsi que des fonctions codants les comportements attendus selon si la réponse est positive ou négative. Par exemple, on appelle cette fonction avec le message CONNECT ainsi que 2 fonctions. Une qui va afficher 'Success !' si l'on recoit comme réponse 'CONNECT_OK' et une autre qui va afficher 'Error' si on recoit un message 'CONNECT_KO'.

Cette classe s'occupe également de conserver les identifiants de l'utilisateur sur le disque.

Client.gui

C'est le package qui s'occupe de toute l'interface utilisateur. On utilise ici la librairie `Lanterna` servant à exploiter d'avantage les possibilités du terminal et permettre notamment de scinder celui-ci en deux. Pour l'affichage des messages du chat, un threads va vérifer l'arrivé de nouveaux messages dans la boite aux lettres, puis va mettre à jour le texte du Panel de droite. Il est possible d'ouvrir plusieurs conversations grâce à une hashmap qui retient l'historique de chaque conversation en fonction du correspondant. Pour ce qui est du Panel de droite (le terminal), le parsing et l'exécution des commandes sont déclenchés à l'aide d'un `Listener` de l'API. Son rôle est de traiter les instructions de l'utilisateur et d'appeller les fonctions nécessaires dans l'objet DataProvider.

ProductViewer.java

ProductViewer quant à elle, fournit des fonctions pour afficher de manière élégante les tableaux de produits.

PeerService.java

Il s'agit du service qui s'occupe de recevoir et d'envoyer les nouveaux messages sur la socket UDP.

Serveur

Notre serveur a été conçu pour stocker les données dans la RAM comme une base de données de type REDIS. Lorsque l'on coupe le serveur, les données sont effacées. Cela permet d'avoir un serveur très rapide mais qui n'est pas conçu pour la persistence des données. Il est composé de deux classes :

Server.java

Il s'agit de la classe qui écoute les requêtes entrante. À chaque nouvelle connexion sur la socket d'écoute, elle crée une nouvelle socket qui est traitée dans une nouvelle Thread de type Handler.

Handler.java

Cette classe s'occupe de gérer les échanges entre le serveur et le client une fois la connexion établie. Elle lit les requêtes ligne par ligne jusqu'à voir un point. Quand elle voit celui-ci, elle transforme la requête en message et la traite via un switch pour déterminer l'action à executér. Toutes les actions néccessitent d'abord de se connecter en suivant le protocole. Sinon, le message NOT_CONNECTED est envoyé. Aussi, le protocole a été écrit de telle façon que, si la requête est inconnue, nous pouvons le spécifier au client avec lequel nous sommes connectés. Il est donc très facile d'étendre le protocole avec de nouvels en-têtes: il suffit d'écrire une nouvelle méthode est de l'ajouter au switch.

Chaque “handler” possède un timeout de 12H pour couper la connexion en cas d’absence de message pendant cette durée. Dans le cas où le client se déconnecte, le serveur ferme automatiquement la connexion et retire l’utilisateur de la liste des IP en cours de connexion.

Sécurité

Intégrité

Pour s’assurer de l’identité de l’émetteur et du récepteur nous pouvons utiliser le système de HMAC. Cela consiste à prendre une empreinte du message que l’on va envoyer avec un token qui nous identifie. Le token est présent de chacun des côtés de la connexion et permet de signer les messages. Lorsque l’on reçoit le message, il suffit de calculer l’empreinte à nouveau et de s’assurer que c’est la même. Ainsi, si elle diffère, nous savons que soit le message a été modifié, soit l’utilisateur n’est pas le bon. Le “.” dans le protocole permet de faire la distinction entre la partie HMAC et le reste.

Protocole Client-Serveur

Pour sécuriser le protocole Client-Serveur et assurer la confidentialité, nous pouvons utiliser les sockets sécurisées fournies par TCP à travers le protocole cela nous assure la confidentialité, et HMAC l’intégrité et l’authenticité. Nous sommes donc bien sécurisés.

Protocole Client-Client

Le protocole Pair-à-Pair requiert le passage par un serveur tierce pour faire l’échange des clefs. Lorsque l’on demande l’ip, on fait une demande au serveur, qui contacte les deux paires pour leur indiquer les clefs qui vont être utilisées lors de l’échange. Une fois cela fait, les deux pairs peuvent chiffrer leurs messages grâce à ces clefs. Cela nous donne la confidentialité et HMAC nous donne à nouveau l’intégrité et l’authenticité. Nous sommes donc sécurisés.

Choix du protocole

Pour le protocole, nous avons choisi d’utiliser TCP pour la partie Client-Server car cela permet d’assurer l’intégrité de l’échange et de l’ordre dans lesquels les paquets arrivent. Même si cela peut ralentir le temps de récéption des informations, cela ne pose pas de soucis car il ne s’agit pas d’informations en temps réel.

Pour la partie Client-Client, nous avons décidé d’utiliser le protocole UDP afin d’alléger le client. En effet, dans le cas de TCP nous aurions dû utiliser autant de connexions qu’il y a de pairs. En outre, UDP nous permet d’être dans un système pair-à-pair. Enfin, la dernière raison qui nous a poussées à écrire le protocole en UDP est le fait qu’il s’agit d’un cours et non d’un projet professionel. C’était donc l’occasion de tester les technologies dans un cadre adapté.