ELBEZ Samuel & JACQUETTE Pierrick
samuel.elbe@gmail.com & pierrick.jacquette@gmail.com
21200353 & 21305551

PROJET

Cahier des Charges :

Réaliser une application de petites annonces. Les fonctionnalités de l'utilisateur doivent être les suivantes :

- Voir toutes les annonces
- Créer une annonce
- Supprimer une annonce et uniquement les siennes
- Contacter le vendeur d'une annonce différent de lui-même

Quand un utilisateur se déconnecte, ses annonces sont supprimés.
Le projet doit fonctionner dans le groupe de 6 personnes.
Un retour pour le client de l'application a lieu toutes les deux semaines.

Dossier de conception :

Tout d'abord on a tous ensemble réfléchi aux différents messages et aux contraintes de l'application. Quelles informations devait être communiquer dans quels messages…., cela nous a permit d'avoir une ébauche de protocole. Le protocole a était rédigé à la suite de cette réunion afin d'avoir que chaque groupe puisse implémenter de son côté mais sans laisser d'ambiguïté possible.

La première étape d'implémentation était la communication entre un client avec le serveur. La seconde étape d'implémentation était la communication entre un client et un autre client.

La date finale de livraison sera le 13 décembre. Nous avons rendu le 29 novembre.

Architecture :

- Advert : objet représentant une annonce qui contient un id, un contenu et l'identifiant du vendeur.

- Client : lance un thread.

- ClientThread : permet de vérifier les demandes de l'utilisateur faites au clavier, les envoyer au serveur, recevoir ce que nous envoie le serveur et de lancer la communication client-client.
  - bufferedReader pour lire l'input de la socket.
  - bufferedReader pour lire ce qu'écrit l'utilisateur.
  - PrintWriter pour écrire dans l'output de la socket.
  - socket sur laquelle le serveur nous a accepté.

- ClientP2P : gère la communication client-client.

- bufferedReader pour lire l'input de la socket avec le client.
- bufferedReader pour lire ce qu'écrit l'utilisateur.
- PrintWriter pour écrire dans l'output de la socket avec le client.
  socket sur laquelle le serveur nous a accepté.

- Message: énumération des différents messages possibles.

- MessageServer : envoie les messages par le serveurs en fonction du protocole

- MyException : définition des différentes erreurs possibles dans les messages reçus.

- Server : objet représentant le serveur avec une socket, une liste d'annonces, une liste d'utilisateur, un compteur pour les id utilisateurs et un pour les id des annonces.

- ServerThread : objet représentant un client qui est connecté au serveur, reçoit les messages, les vérifie et envoi un message au client une réponse.
  - bufferedReader pour lire l'input de la socket.
  - messageServer pour écrire sur l'ouput de la socket.
  - Socket sur laquelle on accepte le client.
  - boolean pour savoir si l'utilisateur se déconnecte.
  - hashMap d'annonce pour avoir toutes les annonces : variable partagé.
  - hashMap d'utilisateur pour avoir toutes les utilisateurs : variable partagé.
  - counter pour savoir l'id d'une nouvelle annonce et nouveau user : variable partagé.
  - idUser : id de l'utilisateur de ce thread.

- User : objet représentant un utilisateur qui contient un id, une adresse, un port et une liste de ses annonces.

Sécurité :

On a mis en place un des fonctions de parsage de message. Quand on reçoit un message on vérifie son contenu afin de ne gérer que les messages non erronés. Par ailleurs, on vérifie ce que l'utilisateur saisi pour envoyer sur le réseau que des messages du protocole.

Notre serveur est toujours disponible sauf si il se plante (comportement testable avec un Ctrl-C), dans ce cas il envoi un message de fin à tous ces clients afin qu'il se termine. Si deux clients sont en communication directe, ils peuvent encore communiquer entre eux et dés qu'il arrête cette communication, ils auront l'information que le serveur est mort.

Pour synchroniser nos objets nous utilisons le mot clé « synchronized ». Cela garantit l'atomicité du bloc de code mais aussi la visibilité des modifications effectuées dans ce bloc. La priorité d'un nouveau d'un nouveau thread client est celle par défaut soit 5 donc tous les clients sont équitables.

ELBEZ Samuel & JACQUETTE Pierrick
samuel.elbe@gmail.com & pierrick.jacquette@gmail.com
21200353 & 21305551

Nous avons choisi le protocole TCP pour les deux communications cela permet d'avoir un service sécurisé sur la remise des paquets. TCP garantit l'ordre et la remise des paquets, il vérifie l'intégrité de l'en-tête des paquets et des données qu'ils contiennent. TCP est responsable de la retransmission des paquets altérés ou perdus par le réseau lors de leur transmission.

Il n'y a aucune garantie de confidentialité pendant les conversations. On utilise des simples sockets avec les messages en clair. Quand un utilisateur se déconnecte ses annonces sont supprimé et l'utilisateur aussi, cela empêche à un client « voleur » de transformer les informations puisqu'elles n'existent plus. On ne peux avoir deux clients sur la même socket et à chaque socket on affecte un identifiant donc le client « voleur » ne peut pas agir quand le bon client est connecté. Il peut voir les messages qui sont envoyés.

Un point de faille possible est dans la situation suivante où A et C des clients et C un mauvais client. A demande au serveur de parler à B, A crée une nouvelle socket, le serveur envoie à B la nouvelle adresse et le port de la nouvelle socket. C écoute les messages entre B et le serveur donc se connecte à A à la place de B. Cette situation est possible.

Dans le futur si le client voudrait une application sécurisé, on pourrait crypté les messages envoyés. Pour éviter un client mal intentionné on pourrait utiliser des certificats (avec keystore) et des sockets SSL.

Par ailleurs, on pourrait avoir un dénis de service. Pour éviter cela on pourrait limiter le nombre de clients présent simultanément sur le serveur. Il faudrait avoir une variable pour connaître le nombre maximal de clients supportables et un compteur pour savoir combien de clients sont connectés. Par ailleurs dans un but malicieux, un seul client pourrait m'envoyer beaucoup de requêtes simultanément. Pour pallier cela, on pourrait savoir combien de requêtes on reçoit de la part de chaque client dans un certain laps de temps. Si ce nombre dépasse une certaine limite alors on déclare le client moins prioritaire.

Remarques :

Le serveur va lancer un thread pour chaque nouveau client avec toutes ses variables qui seront partagées. Chaque client est un thread, pour la communication direct une nouvelle socket est crée.

On prend en compte les Ctrl-C de la part des clients et du serveur.

Nous avons choisi d'utiliser des HashMap pour stocker nos éléments car on peux les supprimer en O(1) (éviter de parcourir la structure à chaque fois).