TP nº 2

Consignes les exercices ou questions marqués d'un ⋆ devront être d'abord rédigés sur papier (afin de se préparer aux épreuves écrites du partiel et de l'examen). En particulier, il est recommendé d'être dans les même conditions qu'en examen : pas de document autres que le cours ni de calculatrice. Tous les TPs se font sous Linux.

1 Préambule

Le but TP est le suivant :
— S'initier à la programmation Client/Serveur en utilisant le protocole UDP
— Illustrer les limites de ce protocole
— Rappeler certains concepts de base de la programmation Java (principalement l'héritage et la création de *threads*).
Le projet Eclipse contenant le code à compléter est récupérable sur la page du cours.

2 Serveur et Client Echo

Le but de cet exercice est d'implémenter un serveur et un client remplissant la fonctionalité `Echo` :
— Le serveur attends des datagrammes UDP contenant de simples messages texte, sur le port 12345.
— Sur reception d'un datagramme, le serveur affiche le contenu de celui-ci dans la console.
— Le client se connecte à un serveur sur le port UDP 12345 d'un serveur.
— L'utilisateur ayant lancé le client peut écrire des messages sur l'entrée standard (dans la console) qui sont envoyés au serveur.
Un tel couple Client/Serveur peut être vu comme un squelette d'application de messagerie instantanée.

2.1 Serveur

Le code du serveur de base est dans la classe `UDPEchoServer`.

1. compléter la méthode `.start(int port)` du serveur. On prendra bien garde à créer l'objet `DatagramSocket` en passant `null` à son constructeur.
2. compléter la méthode `.handlePacket(DatagramPacket packet)` pour appeler `incrMessageCount()` et ainsi compter le nombre de datagrammes reçus par le serveur.

2.2 Client

Le code du client de base est dans la classe `UDPEchoClient`.

1. compléter la méthode `.start(String address, int port)` du client pour qu'il établisse une connexion à l'adresse et au port UDP indiqué dans les paramètres
2. compléter la méthod `.send(String message)` pour que la chaine de caractères passée en argument soit envoyée sur la *socket* reliant le client au serveur.
3. Lancer maintenant une instance du serveur depuis Eclipse et une instance du client dans le terminal. Pour lancer le code du client, se placer dans le répertoire `bin` se trouvant à la racine du projet Eclipse et exécuter

```
java clientserveur.tp02.UDPEchoClient
```

Dans la fenêtre du client, entrer plusieurs messages (valider à chaque fois avec la touche [Entrée]). Après cela, sélectionner la console où est lancé le serveur dans l'interface d'Eclipse et quitter le serveur en appuyant sur [q] (ou n'importe quelle autre touche). Vérifier que le nombre de messages reçus correspond bien au nombre de message envoyés.

2.3 Serveur (bis)

1. compléter maintenant le code de la méthode `.handlePacket(DatagramPacket packet)` pour afficher sur la sortie standard l'adresse et le port du client ayant envoyé le datagramme ainsi que le contenu de ce dernier.

2.4 Client (bis)

1. Recommencer la manipulation de la section 2.2, question 3 et vérifier que les messages sont effectivement affichés dans la console du serveur.
2. Envoyer depuis le client un message très long (de longueur supérieure à `PACKET_LENGTH` octets. Que constate t'on ?
3. ⋆ De quel côté (client ou serveur) peut on corriger le problème précédent ? Proposer un solution en pseudo-code.

> **Réponse:** Le serveur ne pouvant rien recevoir de plus long que `PACKET_LENGTH`, le client doit découper ses messages plus longs que cette limite en plusieurs messages. (Voir le code Java du corrigé pour du pseudo code).

4. Implémenter la solution précédente.

3 Client *flood*

On souhaite implémenter maintenant un autre client dont le seul but est d'envoyer des messages (courts) en raffale au serveur.

— ⋆ Lire attentivement le code de la classe `UDPEchoClientFlood`. Pourquoi cette dernière n'implémente-t-elle que les méthodes `.mainLoop()` et `.main()` ?

> **Réponse:** Cette classe étend la classe `UDPEchoClient`. Elle aura le même comportement que la classe mère pour les méthodes non-redéfinies.

— Compléter le corps de la méthode `.mainLoop()` pour qu'elle envoie `NUM_MESSAGES` messages au serveur (chacun consistant d'un entier croissant entre 0 et `NUM_MESSAGES`.

— Tester ce client avec votre serveur. Que constatez vous sur les messages reçus ?

— ⋆ Proposer une explication pour l'observation précédente.

> **Réponse:** Des paquets sont perdus car le client les envoie plus vite que le serveur ne peut les traiter. UDP étant un protocole simple, il n'assure pas le buffering de tous les paquets ni une notification en cas de paquet perdu.

— Dans la méthode `.start()` du serveur, augmenter la taille du *buffer* interne pour la *socket* en effectuant :

```
size = 2* size;
serverSocket.setReceiveBufferSize(size);
```

juste avant l'appel à `.bind()`. Cela règle-t-il le problème ?

— Augmenter maintenant le nombre de messages envoyés dans la classe `UDPEchoClientFlood` (en changeant la valeur de la constante `NUM_MESSAGES`). La solution précédente fonctionne-t-elle toujours ?

Réponse: L'augmentation de la taille du buffer interne ne suffit plus si le nombre de messages envoyés par le client *flood* augmente aussi.

4 Serveur *multi-thread*

On souhaite enfin écrire une version *multi-thread* du serveur. On souhaite réutiliser un maximum de code du serveur. On a donc défini une classe `UDPEchoServeurMulti` étendant la classe `UDPEchoServeur`. Le but est d'effectuer le même traitement que la classe `UDPEchoServeur` mais en utilisant un *thread* pour chaque nouveau message reçu (le traitement de ce message s'effectuera donc en parallèle du reste du code). Chaque *thread* créé est stocké dans un tableau global afin de pouvoir *attendre*, dans la méthode `.stop()` que tous les *threads* se sont bien arrêtés.

— ⋆ Expliquer ce que fait la méthode `start()` de la classe `UDPEchoServeurMulti`.

Réponse: La méthode initialise le vecteur `threads` et appelle la méthode `start()` de la classe parente.

— ⋆ Expliquer ce que fait la méthode `stop()` de la classe `UDPEchoServeurMulti`.

Réponse: La méthode attends que tous les *threads* de traitement soient terminés et appelle ensuite la méthode `stop()` de la classe parente.

— ⋆ Lire attentivement le code de la méthode `handlePacket` de la classe `UDPEchoServeurMulti`, et en particulier la méthode `run()` de la classe anonyme implémentant `Thread`. À quoi fait référence
`UDPEchoServerMulti.super.handlePacket(packet);`

Réponse: Utiliser `this` ou `super` dans une classe anonyme fait référence à l'Objet lui même. En préfixant `this` ou `super` avec le nom de la classe externe, on fait référence à l'objet englobant (ou son parent). Ici, on appelle donc la méthode `handlePacket` de la classe `UDPEchoServeur`.

— Dans la classe anonyme de la méthode `handlePacket`, initialiser pour l'instant l'attribut `packet` avec la variable `fpacket`.
— Lancer le serveur *multi-threadé* ainsi qu'un client *flood*. Quittez le serveur. Que constatez vous sur les messages ?
— ⋆ (difficile) proposer une explication pour l'observation précédente.

Réponse: Si on regarde le code de la méthode `handlePacket()` de la classe `UDPEchoServeur` on remarque que c'est *le même* tableau `buffer[]` qui est utilisé pour tous les datagrammes. Cela ne pose pas de problème en mode *single-thread* car les datagrammes sont traités les uns après les autres. Dans le serveur *multi-threadé*, le *thread* principal (qui exécute `receive()` peut écraser le contenu du tableau alors qu'un autre *thread* le lit.

— Corriger l'initialisation de la classe anonyme pour corriger le problème.
— L'utilisation du *multi-threading* a-t-elle améliorée sensiblement la situation vis-à-vis de la perte des messages ?
— Simulez maintenant le fait que le serveur peut effectuer un traitement couteux pour chaque message en rajoutant un appel à
`Thread.sleep(30);  //pause de 30ms`
dans la méthode `handlePacket` de la classe `UDPEchoServeur`. Relancer le client *flood* une fois avec le serveur simple et deuxième fois avec le serveur *multi-threadé* et comparez le temps où chaque message a été traité par le serveur.
— ⋆ Quel est l'avantage du serveur *multi-threadé*.

Réponse: Le serveur *multi-threadé* n'est pas obligé d'attendre le traitement du message précédent pour traiter un message reçu. Il améliore donc la latence. Par contre il ne permet toujours

pas de traiter les messages suffisamment rapidement pour suivre le rythme du client *flood*.