

TP n° 3

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 recommandé d'être dans les mêmes 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 TCP
- Illustrer les différences entre TCP et UDP
- Mise en place d'un protocole application rudimentaire au dessus de TCP/IP
- Rappeler certains concepts de base de la programmation Java (création et manipulation de *threads*).
- **Servir d'entraînement au premier TP noté qui sera similaire.**

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 fonctionnalité Echo.

Pseudo code du serveur :

- le serveur initialise une socket TCP sur le port 12345 et se place en attente de connexion sur cette socket.
- sur réception d'une connexion, le serveur crée un nouveau *thread de gestion de connexion*. Ce *thread* récupère les flux d'entrée et de sortie associés à la socket et le reliant au client se trouvant à l'autre et se met à suivre le protocole de l'application.
- une fois le *thread* créé, et s'exécutant en parallèle, le serveur se remet à l'écoute d'une connexion sur la socket
- si aucune connexion n'a lieu dans un délais de 1000ms, le serveur vérifie si l'utilisateur a écrit sur l'entrée standard. Si c'est le cas, le serveur se termine, sinon il retourne en attente de connexion.

Pseudo code du client :

- Le client initialise une socket TCP et la connecte au port 12345 de l'adresse du serveur (localhost par défaut). Le client récupère les flux d'entrée et de sortie associés à la socket et le reliant au serveur.
- Une fois la connexion établie, le client attend que l'utilisateur écrive une commande sur l'entrée standard. Celle-ci est envoyée au serveur et la réponse de ce dernier est affiché dans la console.
- Le client termine quand l'utilisateur envoie la commande de déconnexion au serveur.

Protocole de l'application : Tous les messages échangés entre le client et le serveur se terminent par un retour à la ligne ("`\n`").

- Une fois la connexion établie, le client envoie la chaîne de caractères "`CONNECT\n`" au serveur.
- Le serveur renvoie la réponse "`CONNECT OK\n`".
- le client peut envoyer au serveur la commande "`ECHO ... \n`" où « ... » représente un texte libre mais sans retour à la ligne.
- Pour chaque ligne "`ECHO ... \n`" reçue, le serveur affiche la ligne sur sa sortie standard et renvoie la réponse "`RECEIVED l\n`", où « *l* » est la longueur du message reçu (sans compter la partie "`ECHO` " ni le retour à la ligne).

- le client peut envoyer au serveur la commande "DISCONNECT\n".
- le serveur renvoie en réponse le message "BYE\n" puis coupe sa connexion avec le client.
- sur réception du message "BYE\n", le client se termine.
- tout d'un message *m* par le client autre que ceux décrit ci-dessus provoque la réponse "INVALID *m*\n" de la part du serveur.

Organisation du code : le code du serveur est réparti en deux classes. Une classe principale, TCPEchoServer qui crée le socket serveur et attends les connexions sur le port TCP 12345 et une classe TCPEchoConnection qui étend la classe Thread (ce qui permet à son code de s'exécuter en parallèle du reste de l'application). Le code du client ne possède qu'une seule classe, TCPEchoClient.

2.1 Questions

1. Compléter la méthode start(String host, int port) du serveur.
2. Compléter la méthode mainLoop() du serveur. Cette dernière doit : (i) attendre une connexion, (ii) créer un objet TCPEchoConnection en passant la socket représentant la connexion établie comme paramètre au constructeur, (iii) ajouter cet objet au vecteur clients et enfin (iv) invoquer la méthode start() de cet objet pour qu'il exécute sa méthode run() en tâche de fond.
3. Compléter la méthode run() de la classe TCPEchoConnection pour implémenter le protocole décrit précédemment. Il faut prendre garde à toujours appeler la méthode .flush() du flux de sortie pour s'assurer que le message est bien transmis.
4. Compléter les méthodes start() et mainLoop() du client selon les spécifications données.

On cherche maintenant à établir à quel endroit de l'application les évènements suivants peuvent se produire :

5. ★ A quel moment le client a-t-il la certitude que la connexion avec le serveur est bien établie au niveau réseau? et au niveau application (c'est à dire le serveur est effectivement prêt à traiter des commandes envoyées par le client)? De telles garanties sont-elles possibles avec UDP? (justifier)

Réponse: Au moment où l'appel à .connect(. . .) fait par le client termine (sans exception) on sait que la communication est établie. Cela correspond simultanément à la terminaison de l'appel à .accept() côté serveur. Une fois que le client reçoit CONNECT OK, il sait que le serveur est prêt à recevoir des commandes. De telles garanties sont impossibles avec UDP : les messages peuvent se perdre, ou arriver dans le désordre. Le client pourrait donc attendre indéfiniment une réponse perdue.

6. ★ Y a-t-il une limite sur la taille des messages? Tester en envoyant au serveur des commandes ECHO suivies de messages très longs. Quelle différence y a-t-il avec UDP?

Réponse: Il n'y a pas de limite sur la taille des messages, TCP gère automatiquement l'envoi et la reconstitution des messages longs.

7. ★ Que se passe-t-il en cas de terminaison abrupte du client ou du serveur? Tester les différents cas de figure et dites ce qui peut se produire.

Réponse: Le client est bloqué en attente de lecture sur son entrée standard. Le serveur est en lecture bloquante le flux d'entrée de sa socket. Si on coupe le client abruptement, côté serveur, la lecture sur la socket termine (appel à read() renvoie -1, l'appel à readLine() renvoie null). Le serveur peut tester cette condition pour savoir que le client est déconnecté.

Si on coupe abruptement le serveur, côté client, il y a toujours une lecture bloquante sur l'entrée standard. Une fois que celle-ci termine (quand l'utilisateur écrit quelque chose), le client essaie d'écrire. Le socket ayant été fermé, l'écriture peut provoquer une IOException (mais ce n'est pas obligatoire). Par contre, lors de la lecture de la réponse envoyée par le serveur, cette dernière vaut null (lecture du client sur sa socket avec readLine()).