IFT1025 Automne 2009 — Devoir 4

Miklós Csűrös

26 novembre 2009

À remettre avant 11 : 59 au matin du lundi le 7 décembre par courrier éléctronique. Ce travail est destiné à des équipes de deux ou trois étudiants. Le but du TP est de se familiariser avec l'implantation d'une application client-serveur sur TCP/IP.

1 Application clavardage

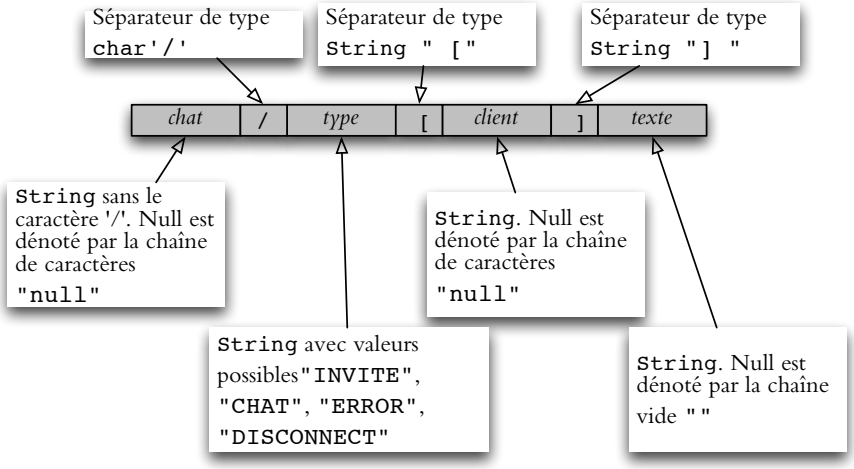
Dans ce TP, vous devez développer une application de clavardage sur l'Internet. La fonctionnalité essentielle de l'application était développée en TP3. Vous devez amender ou modifier le code pour assurer les fonctionnalités suivantes.

⋆ Établissement de connexions sur TCP/IP à l'aide de sockets.

⋆ Possibilité de joindre plus que deux usagers à un clavardoir.

2 Communication TCP/IP

La communication entre serveur et les clients se fait à l'aide de sockets sur TCP/IP. Vous pouvez utiliser un port quelconque (vérifiez si libre, voir `http://www.iana.org/assignments/port-numbers`) : le port 10250 est recommandé. Il faut définir le syntaxe des messages échangés. Un message (*type*, *chat*, *client*, *texte*) est encodé en une ligne de texte selon le format suivant.



Ce format est implanté dans la méthode **toString** d'un objet `Message`. Exemple :

```
#1024/CHAT [armand] Bienvenue mon ami!
```

Le serveur et les clients doivent ignorer toutes données transmises si elles ne conforment pas au syntaxe correct (n'affichez pas un message d'erreur).

En utilisant le socket, les messages sont envoyés à l'aide d'un `PrintWriter`. Utilisez la méthode **println**(`String` *txt*) pour envoyer un message en une ligne, où *txt* est la valeur retournée par **toString** d'un objet `Message`. N'oubliez pas de vider le tampon par **flush**(). Vous devez implanter le décodage du message : une méthode qui prend un `String` comme argument et retourne l'objet `Message` qui y correspond (ou `null` si le texte ne respecte pas le format). Ainsi, on peut lire les messsages par **readLine**() d'un `BufferedReader` sur le flux d'entrée du socket. Attention : le procéssus léger sera suspendu en exécutant **readLine**() jusqu'à ce qu'on recoive un message. Si le socket est fermé par **close** pendant l'attente, alors une `SocketException` est lancée (extension de `IOException`).

Peut-être le plus simple est de définir des extensions de la classe `Connection` qui sont spécifiques au client et au serveur. Par exemple, au côté client, on doit instancier la connexion avec un socket établi. La classe doit redéfinir les méthodes **messageToServer** et **getMessageFromServer**. (On peut même redéfinir **messageToClient** et **getMessageFromClient** pour lancer une `UnsupportedOperationException`.)

2.1 Client

Au côté client, la connexion s'établit en spécifiant le serveur, un port TCP/IP, et un pseudonym.

1. Client construit le socket avec le serveur et le port spécifiés. Si la connexion est refusée (exception lors d'instanciation de `Socket`), le logiciel quitte sans rien faire après avoir affiché un message d'erreur.
2. Quand la connexion est acceptée, le client envoie le message

   ```
   null/INVITE [pseudonym]
   ```

   Simultanément, l'interface graphique est affiché, mais avec l'interaction usager (boutons «Inviter» et «Déconnecter», champ de texte) désactivée. Le procéssus de `SwingWorker` est lancé pour recevoir la réponse du serveur à notre demande de «login».
3. Le serveur vérifie si le pseudonym est unique. Si le pseudonym n'est pas unique, alors un message d'erreur est reçu

   ```
   null/ERROR [pseudonym]
   ```

   À la réception de ce message, le client doit fermer le socket, et arrêter le `SwingWorker` qui écoute au socket. Un message d'erreur est affiché (`JOptionPane`).
4. Si le pseudonym est unique, le serveur envoie un identificateur unique dans un message :

   ```
   null/INVITE [ident]
   ```

   À la réception de ce message, l'identificateur est stocké au client pour les messages ultérieures. L'interactions (boutons et champ de texte) sont activés (utiliser **setEnabled**).

Avant la fermeture de l'application graphique du client, il faut fermer le socket (après avoir envoyé un message de déconnexion, si nécessaire), arrêter le procéssus `SwingWorker` et désactiver les boutons.

Implantez l'exécutable au côté client par la classe `chat.NetClient`. La méthode **main** prend trois arguments fournis à la ligne de commande :

```
% java chat.NetClient serveur port pseudonym
```

2.2 Serveur

Le serveur est lancé en spécifiant un port (10250). L'application doit instancier un `SocketServer` et accepter les connexions en une boucle infinie. À chaque nouvelle connexion, utilisez le nouveau `Socket` pour lancer un `ClientAssistant` avec un client anonyme (identificateur et pesudonyme `null`). Le premier message reçu doit être le «login» du client : voir étape 2 en §2.1. Si le pseudonym est unique, alors le `ClientAssistant` associé doit créer un identificateur unique (p.e., une variable d'un objet `Server` qui est un entier incrémenté à chaque connexion d'un nouveau client). Cet identificateur est le pseudonym sont enregistrés localement, et l'identificateur est envoyé au client : voir étape 4 en §2.1. Si le pseudonym n'est pas unique, alors un message d'erreur est envoyé au client (étape 3 en §2.1), la connexion est fermée, et le procéssus de `ClientAssistant` arrête son exécution. À la déconnexion du client, il faut également fermer le socket est arrêter le procéssus `ClientAssistant`. Attention : on peut perdre le socket (p.e., la connexion internet du client est perdue) — quand cela arrive, il faut agir comme si on avait reçu un message de déconnexion explicite. Le serveur ne doit pas planter sous aucune condition. Considérez les possibilités de rupture de connexion ou d'un client «malicieux» qui ne respecte pas les règles de communication. L'exécution du serveur n'arrête que quand le `SocketServer` ne fonctionne plus, ou le système d'exploitation arrête le procéssus (p.e., `kill` sur la ligne de commande).

3 Participants ultiples

Dans cette version de l'application, il est possible de rajouter des clients additionnels à un clavardoir existant. Spécifiquement, chaque participant a le droit d'inviter n'importe quel autre usager, en envoyant le message

```
chat/INVITE [pseudonym]
```

au serveur. Ici, `chat` est l'identificateur d'un clavardoir où on participe. Le serveur doit agir comme à la réception d'un message `"null/INVITE [pseudonym] "`, sauf qu'il n'est pas nécessaire de créer un nouveau clavardoir. Donc, le serveur envoie le message `"chat/INVITE [ami] "`, où `ami` est le pseudonym de l'usager qui a initié l'invitation. Cette fonctionnalité est implantée à l'aide d'une modification de l'interface graphique du client : il faut ajouter un bouton d'invitation à chaque fenêtre de clavardoir, et un champ de texte pour spécifier le pseudonym de l'usager invité. (On garde aussi le paire existant de bouton-champ de texte dans la barre d'outils, pour l'établissement d'un nouveau clavardoir.) À l'ajout d'un nouveau participant à un clavardoir existant, il ne faut pas ouvrir une fenêtre separée : toute la communication du clavardoir s'affiche dans la même fenêtre.

Implantez l'exécutable au côté client par la classe `chat.NetClient`. La méthode **main** prend un argument fourni à la ligne de commande :

```
% java chat.NetServer port
```

4 Travail et soumission

Comme prototype, vous pouvez utilisez votre propre code du Devoir 3, ou ma solution affichée sur le site Web du cours. Le code peut être modifié et restructuré comme nécessaire, il faut juste assurer que la spécification de la fonctionnalité (format de messages échangés), et celle de l'interface graphique soient satisfaites. N'importe quelle implantation de `NetClient` devrait fonctionner avec

n'importe quelle implantation de `NetServer`. En fait, je vous encourage de tester l'interaction entre vos applications et celles d'autres équipes.

Vous pouvez tester votre code sur la même machine pendant le dévéloppement.

```
% java chat.NetServer 10250 &
% java chat.NetClient localhost 10250 Alceste &
% java chat.NetClient localhost 10250 Philinte &
% java chat.NetClient localhost 10250 Celimene &
```

La connexion serveur-client devrait être fonctionnelle aussi à travers d'ordinateurs différents sur un réseau, mais faites attention : il est possible que le port 10250 n'est pas accissible à travers d'un pare-feu (*firewall*).

Travail estimé : 200–300 lignes de code additionnel (incluant commentaires et lignes vides pour lisibilité).

Compilez un archive jar nommé `Chat.jar` qui contient les classes du package `chat`. L'archive doit inclure les sources `.java` et les classes compilées `.class`. Dans chaque fichier de source, indiquez le membre de l'équipe qui l'a développé, en javadoc (sous `@author` dans la documentation de la classe). Dans votre courrier éléctronique, donnez le nom de tous les membres de l'équipe.

Envoyez le fichier comme attachement dans un courriel à `dift1025@iro.umontreal...`. Votre programme doit être fonctionnel, clair, commenté et original (plagiat=0).