Programmation Orientée Objet
M2 CCI 2008–2009
hoenen@icps.u-strasbg.fr

Université de Strasbourg
Département d'informatique
13 février 2009

TP7 : réseau et threads

1 Connexion au réseau

Les protocoles Deux protocoles de communication sont utilisés dans Java : TCP (*Transmission Control Protocol*) et UDP (*User Datagram Protocol*). Le protocole TCP est basé sur une connexion entre deux ordinateurs qui fournit un tunnel par lequel les données transitent de manière fiable. Le protocole UDP ne provoque pas de connexion directe entre deux ordinateurs, les paquets de données sont envoyés de manière indépendante et sans aucune garantie concernant leur arrivée effective. Le protocole TCP est à privilégier lorsque l'on souhaite garantir la transmission de toutes les données, tandis que le protocole UDP permet une transmission de données plus rapide mais sans contrôle.

Les ports Les ports sont des dispositifs permettant d'acheminer les données venant du réseau (donc d'une unique connexion physique) vers différentes applications. Ces données sont toujours accompagnées d'informations concernant l'adresse de la machine (adresse IP, sur 32 bits pour IPv4) et le port de destination sur cette machine (sur 16 bits). Le numéro de port est utilisé par le protocole de communication (TCP ou UDP) pour déterminer à quel processus associer les données. Dans le cas de TCP, un socket est rattaché à un port et les données transitent via ce socket. Dans le cas d'UDP, chaque paquet contient le numéro du port de destination.

Les ports étant codés sur 16 bits, un numéro de port a toujours une valeur comprise entre 0 et 65535. Les numéros 0 à 1023 sont réservés à des services bien connus (http, ftp, ssh...) et ne doivent pas être utilisés autrement par une application.

L'API Toutes les classes utiles pour gérer une connexion au réseau dans un programme Java se trouvent dans le paquetage `java.net`. Ce paquetage contient notamment les classes `URL`, `URLConnection`, `Socket` et `ServerSocket` pour le protocole TCP, et les classes `DatagramSocket` et `DatagramPacket` pour le protocole UDP.

Dans ce TP on se concentrera uniquement sur le protocole TCP. On va commencer par implanter un mécanisme de communications client–serveur le plus basique possible.

1. Le `Serveur` initialise un `ServerSocket` sur un port donné et attend une connexion d'un client sur ce port (méthode `Socket accept()`). Une fois la connexion établie, il récupère les flux d'entrée et de sortie (`get(In/Out)putStream`) et les enveloppe dans des flux de caractères. On récupère alors la chaîne de caractère envoyée par le client et on lui la renvoie en y ajoutant un message du serveur.
2. Le `Client` ouvre un `Socket` vers le serveur (adresse + port). On récupère les flux d'entrée et de sortie de ce socket en les enveloppant dans deux flux de caractères (par exemple un `PrintStream` pour la sortie et un `InputStreamReader` bufferisé en sortie). On écrit un message à destination du serveur sur le flux de sortie et on lit et affiche à l'écran le message reçu en retour sur le flux d'entrée.

2 Le chat

On va essayer d'écrire un petit programme de chat (ou t'chat...) en se basant sur le mécanisme client – serveur vu précédemment. L'idée est rigoureusement identique sauf que les messages envoyés sont tapés au clavier au fur et à mesure. On va gérer pour l'instant uniquement les discussions entre deux utilisateurs distants, dont l'un est le serveur et l'autre le client.

1. Reprendre les classes `Client` et `Serveur` implantées à l'instant pour créer des classes `ChatClient` et `ChatServeur`. Cette fois client et serveur contiennent une boucle dans laquelle ils lisent une ligne rentrée au clavier, l'envoie vers le correspondant puis reçoivent une ligne de leur correspondant et l'affiche.
2. Ajouter un test pour permettre de sortir de la boucle, par exemple lors de la réception d'un message spécifique (ex : " !quit").
3. Le nom du serveur, son port de connexion ainsi que le pseudo de l'utlisateur doivent être passés en paramètre du programme client. Le programme serveur reçoit lui en paramètre uniquement le port et le pseudo. Modifier l'envoi de message (client et serveur) pour y inclure le pseudo.
4. Quelles sont les limitations de cette approche pour la réalisation d'un chat ?

3 Threads et retour sur le chat

Multitâche Un thread, ou fil d'exécution, représente une partie d'un programme qui fonctionne de manière parallèle avec d'autre threads. En réalité, si on ne dispose que d'un processeur mono-cœur, chaque thread se voit attribué du temps CPU durant lequel son exécution se poursuit, avant d'être endormi pour laisser d'autres threads s'exécuter. Le système se charge d'attribuer le temps CPU pour les différents threads qui du coup, semblent s'exécuter simultanément.

Les threads Java Tout ce qu'il faut pour utiliser des threads supplémentaires (chaque `main` est déjà associé par défaut à un thread d'exécution) se trouve dans le package `java.lang`, on a donc pas besoin d'importer spécifiquement un paquetage. Il existe deux manières d'utiliser des threads :

- en implémentant l'interface `Runnable` et en définissant la méthode `void run()`,
- en héritant de la classe `Thread` et en redéfinissant la méthode `void run()`.

Les traitements à effectuer par le thread sont à mettre dans cette méthode `run`. Cette méthode est exécutée tout de suite après la création du thread par un appel à la méthode `start()`. Cette méthode `start` est appelée directement depuis une instance, dans le cas d'une classe qui hérite de `Thread` :

```
class MaClasse extends Thread {
  public void run() {
    ...
  }

  public static void main(String[] args) {
    MaClasse monObjetThread = new MaClasse();
    monObjetThread.start();
    ...
  }
}
```

ou depuis un objet de type `Thread` associé à une instance, dans le cas d'une classe qui implémente `Runnable` :

```
class MaClasse implements Runnable {
  public void run() {
    ...
  }

  public static void main(String[] args) {
    Thread monThread = new thread(new MaClasse());
    monThread.start();
    ...
  }
}
```

On va maintenant reprendre le code du chat, et y ajouter des threads pour le rendre plus réactif. Il faut que l'envoie et la réception des messages ne correspondent pas à un ordre pré-établi et ne soient pas bloquants.

1. Créer une classe `ChatEnv` et une classe `ChatRec` pour gérer respectivement l'envoi et la réception des messages par un thread. Autrement dit, il suffit de déclarer ces deux classes comme implémentant l'interface `Runnable`, et de placer le code correspondant à l'écriture au clavier et l'envoi d'un message (resp. la réception d'un message et son affichage à l'écran) dans la méthode `run`.
2. Modifier les classes `ChatClient` et `ChatServeur` pour qu'elles utilisent désormais deux threads associés aux classes `ChatRec` et `ChatEnv`.
3. Modifier ce chat pour autoriser la sauvegarde de l'historique d'une conversation.
4. Quelles autres modifications apporter à ce chat pour en améliorer la praticité ?