Texte du TP sur les sockets
Yvon Kermarrec

Les programmes sont disponibles depuis
http://formations.telecom-bretagne.eu/syst/TPDIST/socket/

1ère partie : les sockets avec UDP

- récupérez et installez dans un répertoire les 2 fichiers Example1Receiver.java et Example1Sender.java. Ces 2 programmes mettent en œuvre une communication avec UDP
- consultez dans un premier temps le code et regardez comment les sockets sont gérées. Regardez en particulier les différents appels à l'API Java.
- compilez ce code puis dans 2 fenêtres, lancez ces programmes en respectant l'ordre avec les commandes **java Example1Receiver 2000** et **java Example1Sender localhost 2000 "texte"** (localhost référence ici la machine locale càd la machine sur laquelle le programme est lancé). Que se passe—il ?
- relancez ensuite le code mais avec un ordre différent : i.e. ; **java Example1Sender localhost 2000 "texte"** puis **java Example1Receiver 2000** (toujours à partir de 2 fenètres différentes). Que se passe t il ? expliquez.
- Connectez vous ensuite une autre machine en ouvrant une fenêtre sur une machine distante. Relancez **Example1Receiver** puis sur l'autre machine et **java Example1Sender machine-distante 2000 "texte" (**avec machine distante qui est le nom de la machine sur laquelle fonctionne **Example1Receiver).**
- Relancez les mêmes séquences d 'opérations mais avec une chaîne de caractères plus longue en paramètre). Que se passe t il ? expliquez.
- Rajoutez du code afin que le serveur attende un maximum de 5 secondes (méthode setSoTimeout). Que se passe t il lorsque le délai est écoulé ?
- Modifiez le code de **Example1Receiver** afin qu'il boucle sur la réception de message. Que se passe t il s'il y a 2 émetteurs désormais au lieu d'un seul ? vérifiez que les messages émis sont bien reçus.
-

2ème partie : les sockets avec TCP

- récupérez et installez dans un répertoire les 2 fichiers Example4ConnectionAcceptor.java et Example4ConnectionRequestor.java. Ces 2 programmes mettent en œuvre une communication avec TCP
- consultez dans un premier temps le code et regardez comment les sockets sont gérées. Regardez en particulier les différents appels à l'API Java.
- compilez ce code puis dans 2 fenêtres, lancez ces programmes en respectant l'ordre avec les commandes **java Example4ConnectionAcceptor 12345 « titi »** et **java Example4ConnectionRequestor localhost 12345**. Que se passe—il ?
- relancez ensuite le code mais avec un ordre différent. Que se passe t il ? expliquez.

- Modifiez le code afin que l'émetteur envoie une chaîne de caractère et un entier. Modifiez le récepteur également.

- récupérez et installez dans un répertoire les 2 fichiers Example5* ainsi que MyStreamSocket.java. Cette classe java permet la séparation de la logique de l'application et de la logique de service, en masquant les détails de l'utilisation des sockets. Regardez le code puis compilez le et exécutez le. En quoi ces derniers programmes sont-ils différents de ceux de l'exemple 4 ? expliquez l'impact pour le programmeur.

3ème partie : réalisation d'un outil de 'chat' avec les socket

- on s'intéresse à l'écriture d'une application de type 'chat' (messagerie instantanée) entre dans 2 utilisateurs. On souhaite dans un premier temps que ces 2 programmes se désignent explicitement (en précisant le numéro de port et le nom de la machine de l'interlocuteur). Pour simplifier l'écriture du programme, nous allons également considérer que les 2 programmes fonctionnement alternativement : un en émission puis en réception et l'autre, en réception puis en émission.
- Dans un premier temps, vous pouvez récupérer les fichiers dans le répertoire « chat/base ». Ce programme met en oeuvre une communication entre deux processus distants ou non et exécutent presque le même code. Compilez le code puis sur deux terminaux, lancez :

```
java Example2ReceiverSender  localhost  10000    20000    msg1
java Example2SenderReceiver  localhost   20000    10000    msg2
```

- Décrivez ce qui se passe . Pourquoi l'ordre du lancement est il important ?
- Modifiez le code de façon à introduire une boucle : chaque lecture se fait dans une boucle avec une temporisation de 3 secondes par exemple.
- On s'intéresse maintenant à une version plus élaborée avec une sorte de serveur de noms (qui mémorise les différents interlocuteurs enregistrés et les moyens de les contacter). Pour cela, chaque processus communicant est repéré par un nom logique (son identité). Le problème qui se pose ici est de savoir comment les deux processus peuvent se connaître mutuellement et connaître les ports qu'ils doivent utiliser pour communiquer. Notre approche est assez simple : elle consiste à ce que chacun des 2 écrivent ses coordonnées réseau dans un fichier qui porte le nom logique (vous trouvrez un exemple de manipulation de fichiers avec Java avec lecture.java).
- Ecrivez le code …. Ou …. Trouvez la solution directement dans « chat/solution ».
- Pour lancez le code de l'application, utilisez ici 2 terminaux et lancez les commandes suivantes : - on suppose ici que le premier s'appelle *lulu* et qu'il attend des messages sur le port 20001 et désire communiquer avec *titi*

```
java Example2ReceiverSender lulu 20001 titi "msg2"
java Example2SenderReceiver titi 20004 lulu msg1
```

- Expliquez pourquoi ces deux programmes doivent être lancés dans le même répertoire ? pourquoi faut il NFS ou un autre système de partage de disque ? Que faudrait il pour passer à un 'serveur de noms' qui éviteraient ces limitations ?

4ème partie : multicast

Dans le répertoire ‘multicast’ vous trouverez un exemple d’utilisation du multicast. Compilez le code et sur ou plusieurs machines lancez un ou plusieurs receveurs. Lancez ensuite l’expéditeur