Java pour le réseau

S. Reynal

Automne 2016

Cette série de quatre séances de TP Java a pour objectif de vous initier au développement rigoureux d'applications orientées objet, et en particulier d'IHM, dans le domaine du réseau. Seront abordés :

- la modularité du développement, autour du paradigme model-view-controller
- les sockets, les datagrams et la communication en java sur un réseau
- la parallélisation avec les Threads
- enfin, la réalisation d'IHM avec les frameworks Graphics2D et Swing

Le sujet consiste à réaliser un client-serveur simple pour simuler un système de tchat, en UDP puis en TCP, d'abord en ligne de commande, puis avec une interface graphique sommaire.

1 Evaluation

L'évaluation portera sur les quatre critères suivants (5 points par item lorsque tous les critères sont réunis) :

- **Fonctionnalité du code produit** : l'application est disponible sous forme de fichier jar, elle est fonctionnelle et remplit le cahier des charges, il n'y pas de bugs apparents, l'interface graphique est intuitive et agréable à utiliser ;
- **Qualité de la documentation du code** : le code est systématiquement commenté en anglais, le code est clair et auto-explicatif et les noms de classes, de champs et de méthodes ont été judicieusement choisis, une documentation au format HTML claire et lisible a été produite via l'outil javadoc ;
- **Qualité de l'organisation et de la rédaction du code** : choix judicieux des noms de classes et de méthodes, rangement efficace des classes dans les différents packages, utilisation efficace des classes, classes abstraites, interfaces et enums lorsque le besoin s'en fait sentir, choix judicieux des qualificateurs d'accès (private, protected, public, package) ;

— **Progression au cours des 4 séances** : vous avez progressé de façon nette et évidente en tenant compte des conseils et indications des enseignants, vous avez gagné en autonomie et en confiance et êtes capable de debugger seuls votre code en analysant les messages d'erreur du compilateur ou les exceptions produites par le code à l'execution, vous savez utiliser la documentation de l'API de façon autonome et systématique.

2 Préalables

Pour coder vous avez deux solutions :

— utiliser un éditeur de texte et compiler/exécuter en ligne de commande

— utiliser un environnement de développement intégré (IDE) : eclipse, ou netbeans, pour l'essentiel (il y en a d'autres).

Je suggère aux débutants de commencer par l'approche 1. Lorsque vous maîtrisez bien, passez à eclipse, mais pas avant. Eclipse a l'avantage d'être très puissant pour de gros projets, mais il cache des myriades de menus de configuration dans lesquels on peut vite se perdre si l'on n'a pas d'expérience du développement java.

Approche "ligne de commande" :

— choisissez un éditeur de texte simple (geany, kate, ou mieux : ATOM, le top ; vous pouvez l'installer en local si besoin) ;

— ouvrez une console (on dit aussi un "shell" ou un "terminal")

— compilez avec la commande "javac MaClasse.java" ; vérifier avec "ls" que cette commande produit bien (au moins) un fichier MaClasse.class

— exécutez avec la commande java MaClasse, étant entendu que la méthode main se trouve bien dans cette classe.

Méthode simple et efficace pour comprendre ce qu'on fait avec de petits projets.

Attention : si vous créez un package, disons "ui", alors pour exécuter une classe contenant une méthode main, il faut faire "java ui.MaClasse" depuis le répertoire se trouvant juste au-dessus de "ui", et non "java MaClasse", qui donnera une exception de type "ClassNotFoundException".

Approche "Eclipse" : pour lancer eclipse, ouvrez une console et entrez "eclipse" suivi d'un TAB énergique pour faire apparaître toutes les versions qui se trouvent dans le path.. car... il existe plusieurs versions sur le serveur d'application de l'ENSEA, et il vous faut vérifiez que vous lancez la plus récente (qui se trouve être "neon", la suivante étant "mars", et enfin "kepler", qui date de 2014 ; tout ça se vérifie, hélas, une fois eclipse lancé, dans "à propos").

Documentation : Par ailleurs, inutile de vouloir développer en java sans avoir la documentation de l'API (Application Programming Interface) sous les yeux en permanence. Pour cela, le plus simple est d'ouvrir un browser, de taper "Java API Doc" dans google, ou d'aller directement à `https://docs.oracle.com/javase/7/docs/api/`.

Conventions de codage en Java :

- les noms de classe sont en minuscule mais commencent par une majuscule : ainsi, `UdpServer` ou `MultiThreadTcpServer` ;
- les noms de méthodes aussi, exceptés qu'ils commencent par une minuscule ; ainsi, `computeFourierTranform` ou `createDataModel` ;
- les noms de variables suivent les mêmes conventions que les noms de méthodes.
- les noms de package sont exclusivement en minuscules : ainsi, `network` ou `ui.event.keyboard` ;
- les constantes (enum ou bien static final) sont en majuscules, les "mots" étant séparés par le symbole "_" : ainsi, DEFAULT_MAX_VALUE ou bien COLOR_RED.
- lorsqu'on manque d'imagination pour inventer un nom de variable qui soit parlant, le mieux est de choisir un nom directement inspiré du *type* de la variable en question : par exemple, une variable de type `UdpServer` pourra tout simplement s'appeler `udpServer` ou `udpServerPrincipal` s'il peut y avoir plusieurs serveurs par exemple. En tout les cas, pas `server` (pas assez précis), ni `paquetserv` (trop sybillin), encore moins `systemReseau` (n'a carrément rien à voir avec le sujet).

D'une manière générale, les noms doivent être self-explanatory. Une variable `screenSizeX` est plus parlante que `ssx`. Pensez que votre code doit pouvoir être compris sans trop d'effort intellectuel par quelqu'un d'extérieur au projet.

Quelques commandes unix utiles :

- `cd dossier` : aller dans un nouveau dossier
- `cd ..` : remonter d'un dossier
- `ls` : lister le contenu du dossier
- `ls -al` : idem, avec tous les détails
- `mv source destination` : déplacer ou bien renommer
- `cp source destination` : copier un fichier
- `cp -R source destination` : copier tout un dossier vers un autre
- `rm nom` : supprimer un fichier ou un dossier
- `man nomDeCOmmandeInconnue` : afficher la doc d'une commande unix

— et un pense-bête ici :
`http://www.dummies.com/computers/operating-systems/linux/common-linux-commands/`

3 Cahier des charges

Il vous est proposé de développer quelques exemples de client-serveur (un simple echo, un tchat, un serveur musical) où le serveur communique d'abord avec un client, puis avec plusieurs clients à la fois en utilisant des Threads. Dans un second temps vous devrez intégrer une interface graphique, par exemple afficher un fil de conversation dans un composant java de type JTextArea. Ce cahier des charges vous donne quelques contraintes de développement de votre application. Celles-ci étant volontairement succinctes, il s'avère que ce qui est imposé dans la suite doit impérativement être respecté, et qu'a contrario tout ce qui n'est pas imposé constitue votre marge de liberté et de créativité.

Vous organiserez toutes vos classes en plusieurs packages :

— **network** : classes dédiées à la communication client-serveur ;
— **model** : classes dédiées à la modélisation, aux algorithmes, au son, etc ; bref, toute la machinerie interne ;
— **view** : classes dédiées à la visualisation ; ici on trouvera, à base de composants Swing, les éléments habituels d'une interface graphique : menus, boutons, listes déroulantes, ... Pour le tchat, on pourra rajouter l'affichage des conversations, un textfield pour entrer du texte, ... mais aussi les classes qui gèrent la configuration graphique du serveur et du client (port, adresse ip du serveur, etc)
— **controler** : classes dédiées aux évènements de contrôle de l'interface graphique, c'est-à-dire gestion des évènements souris et clavier (toutes les classes de type XXXListener).

C'est à vous seuls de décider quelle classe va dans quel package ! Au début, tout (ou presque) ira dans le package model, mais au fur et à mesure, ça s'enrichira de nouveaux packages.

Pour ce qui relève du réseau, s'aider du memo en ligne sur les classes du package java.net : `http://www-reynal.ensea.fr/docs/java-rt/MemoTPJavaRT.pdf`

3.1 Création d'un client-serveur UDP

3.1.1 Serveur UDP

Dans cette partie, on veut créer une classe UDPServer représentant un serveur UDP qui attend de recevoir de ses clients des datagrammes contenant des chaînes de caractères encodées en "UTF-8". Ce serveur se contente d'afficher sur la sortie standard la chaîne reçue préfixée par l'adresse du client. On considèrera que le serveur doit accepter des chaînes de caractères dont la taille une fois encodée peut aller jusqu'à 1024 octets ; le serveur tronque les données reçues au delà de cette taille.

Le serveur doit avoir au minimum :

— un constructeur prenant en argument le numéro de port d'écoute du serveur,

— un constructeur par défaut (numéro de port par défaut),

— une méthode launch() qui démarre le serveur (pour l'instant on utilisera pas les Threads)

— et une méthode main(String[] args) qui permet de démarrer un serveur par exemple avec la commande : `java UDPServer 8080`, ce qui signifie qu'il vous faut récupérer l'argument du numéro de port dans arg[0] et le traiter...

— une méthode toString() qui renvoie une String décrivant l'état du serveur.

Pour tester ce serveur, vous pouvez dans un premier temps utiliser la commande `netcat`, depuis un autre terminal, par exemple avec : `$ nc -u localhost 8080 abcdef`

3.1.2 Client UDP

On souhaite dans un second temps créer une classe UDPClient qui lit sur l'entrée standard les lignes de texte saisies par l'utilisateur et les envoie, encodées en "utf-8", dans un datagramme UDP à destination du serveur spécifié en argument sur la ligne de commande.

Par exemple, avec le serveur UDPServer précédemment démarré, on lancerait un client par : `$ java UDPClient localhost 8080`.

Indications : pour lire le clavier, utiliser la classe `java.io.Console` que l'on peut obtenir via la classe `System`.

3.2 Création d'un client-serveur TCP

Dans cet exemple, une communication entre un processus client et un processus serveur est établie. Les deux processus échangent des messages sous forme de lignes de texte. Le protocole est très simple : le processus client commence par

émettre un message et le serveur lui répond par un écho de cette ligne où chaque caractère a été converti en hexa. Par exemple, si le serveur reçoit "Hello !", il répond "48 65 6C 6C 6F 20 21".

3.2.1 Serveur TCP

Créer une classe TCPServer sur le même modèle que UDPServer, à l'exception que la méthode launch() suit la séquence suivante :

— création d'une instance de ServerSocket ;
— attente d'une connection via la méthode `accept()` ;
— acceptation de cette connection et obtention d'un Socket de connection ;
— obtention de l'InputStream associé au Socket
— lecture des données provenant du Socket, et affichage à la console ;
— réponse (echo) au client.

Indications : on lira à dessein la documentation de la classe InputStream ...

3.2.2 Client TCP

Créer une classe TCPClient sur le modèle de UDPClient, qui établit une connection TCP avec le serveur spécifié sur la ligne de commande (adresse, port). Une fois la connection établie, ce client lit sur l'entrée standard une ligne de texte saisie par l'utilisateur, l'envoie au serveur après l'avoir encodée en UTF8, et lit en guise de réponse la ligne de texte hexa renvoyée par le serveur. Le client répète l'opération jusqu'à ce que l'entrée standard soit fermée par l'utilisateur (CTRL+D).

Comme pour l'UDP, le client se lancera avec une commande du type `$ java TCPClient localhost 8080`.

3.2.3 Serveur acceptant de multiples connections TCP

On souhaite maintenant créer un serveur TCPMultiServer capable d'accepter plusieurs requêtes simultanément. Pour chaque client qui se connecte, le serveur continue à renvoyer la ligne encodée en hexa.

La possibilité de gérer plusieurs clients simultanément repose sur l'utilisation des Threads (on consultera le tutorial `http://docs.oracle.com/javase/tutorial/essential/concurrency/` si besoin).

Pour commencer et se faire une idée du fonctionnement des Threads, on va provisoirement mette de côté les aspects réseaux : pour cela, créez une classe ThreadTest qui hérite de la classe java.lang.Thread. Surchargez la méthode run() pour qu'elle affiche périodiquement un message à la console :

```
public void run(){
    while(true){
```

```
            System.out.println(getName() + ": " + counter++);
        }
        sleep(100); // attend 100ms
}
```

où counter est un champ de votre classe (qui permet simplement de suivre les appels successifs). Rq : la méthode sleep() lève une exception, qu'il faudra gérer...

Dans la méthode "main()", créer plusieurs instances de `ThreadTest` et appeler à chaque fois `start()` sur chacun de ces instances pour démarrer le Thread, et observer ce qui se passe...

A partir de là, si vous avez compris ce qui se passe, vous devriez pouvoir créer un TCPServer "multithread".

Tout d'abord, il vous faut créer une classe supplémentaire, ConnectionThread, qui sera chargée de gérer une connection client à la fois. Cette classe hérite de Thread, et sa méthode run() contient désormais à peu près la même boucle infinie qui dans TCPServer lisait les données provenant du client et les affichait à la console. Comme vous avez besoin de connaître l'InputStream du Socket de connection pour lire les données, il faut que cet InputStream soit passé en argument au constructeur de ConnectionThread, et que vous le stockiez dans un champ de votre classe pour pouvoir le réutiliser dans la méthode run() (technique TRES classique de communication inter-objets).

Une fois que ceci est fait, passez au codage de la classe TCPMultiServer proprement dite. En l'occurence, l'idée à suivre est la suivante :

- dans la méthode launch(), on attend une connection entrante avec accept() ;
- dés que celle-ci arrive, on récupère le Socket correspondant, puis l'InputStream du Socket ;
- on instancie alors un nouveau ConnectionThread en lui passant cet InputStream comme argument, on le démarre avec start(), et le tour est joué... On a plus qu'à revenir à l'étape 1, consistant à attendre une nouvelle connection (car pendant ce temps un ConnectionThread est en train de s'occuper du dernier client qui s'est connecté).

Indications : utilisez moultes System.out.println() pour débugguer votre code, car avec les Threads, comprendre les problèmes de synchronization peut vite s'avérer tricky. Pensez notamment à utiliser la possiblité, dans la classe Thread, de passer un "nom" au Thread via son constructeur, ce qui permettra ensuite, avec la méthode `getName()`, de savoir dans quel Thread on se trouve à tel ou tel moment de l'execution du code...

3.3 Ajout d'une interface graphique côté client

Pour finir on souhaite ajouter une interface graphique au système multithread précédent, afin de réaliser un système de tchat. Pour cela, chaque client ouvrira une fenêtre (JFrame) contenant un champ (JTextField) où l'utilisateur peut rentrer du texte, et un champ (JTextArea) affichant les conversations des autres participants. Le serveur est laissé inchangé pour l'instant (il est toujours lancé en ligne de commande depuis le terminal).

Une contrainte forte dans la suite sera de "ranger" les classes qui relèvent de l'interface graphique, exclusivement dans le package "ui". Cela vous obligera à séparer la "vue" (composants graphiques) du "model" (l'aspect réseau).