
TP1 : LES SOCKETS

L'objectif de ce TP est de vous familiariser avec les sockets en JAVA. Pour ce faire nous allons construire progressivement une application de chat. Cette application sera codée de deux façons :

- une version centralisée en TCP
- une version multicast avec UDP

On vous demande de **coller strictement aux consignes** en particulier sur le nommage des classes et des méthodes.

Comme pour tous les TP's à suivre, vous êtes invités à consulter la javadoc. Voici quelques liens utiles :

Pour les interfaces :

- <https://docs.oracle.com/javase/7/docs/api/java/net/InetAddress.html>
- <https://docs.oracle.com/javase/7/docs/api/java/net/NetworkInterface.html>

En TCP :

- <https://docs.oracle.com/javase/7/docs/api/java/net/ServerSocket.html>
- <https://docs.oracle.com/javase/7/docs/api/java/net/Socket.html>

En UDP :

- <https://docs.oracle.com/javase/7/docs/api/java/net/DatagramSocket.html>
- <https://docs.oracle.com/javase/7/docs/api/java/net/DatagramPacket.html>

Vous avez deux séances pour réaliser ce TP.

PARTIE 1 : ADRESSAGE ET RAPPELS (15 MINUTES)

TCP/IP s'appuie sur la notion de ports et d'adresses.

En vous appuyant sur les classes **java.net.InetAddress** et **java.net.NetworkInterface** décrites en cours, écrire une classe **AfficheInterfaces** dont la méthode *main* affiche l'ensemble des interfaces de votre réseau et la/les adresses IP associées.

Votre classe doit appartenir au package `istic.pr.socket.address`.

Le code fait une dizaine de lignes.

Vous devrez parcourir deux énumérations :

- une pour avoir chaque interface et le nom + nom «Human-readable» associés (sur certains systèmes les deux peuvent être identiques),
- et pour chaque interface pour avoir leurs adresses (parfois aucune, souvent une IPv4 et une IPv6)

Votre affichage doit ressembler à cela :

net9:Carte Microsoft ISATAP \#2

eth5:VMware Virtual Ethernet Adapter for VMnet1

->/192.168.149.1

->/fe80:0:0:0:6563:9fc2:b4d2:6b0b%19

PARTIE 2 : IMPLÉMENTATION D'UN CHAT EN TCP

EXERCICE 1: SERVEUR ET CLIENT ECHO

Question 1

En vous basant sur les exemples de TCP fournis en cours, on vous demande d'écrire une classe **ServeurTCP** dans le package `istic.pr.socket.tcp.echo`

Ce serveur TCP se contente de retourner au client ce qu'il lit.

On vous donne le squelette suivant à compléter :

```
//...

package istic.pr.socket.tcp.echo;

public class ServeurTCP {

    public static void main(String[] args) {

        //Attente des connexions sur le port 9999

        //Traitement des exceptions

        //Dans une boucle, pour chaque socket clientes, appeler traiterSocketClient

    }

    public static void traiterSocketClient(Socket socketVersUnClient) {
        //Créer printer et reader

        //Tant qu'il y'a un message à lire via recevoirMessage

        //Envoyer message au client via envoyerMessage

        //Si plus de ligne à lire fermer socket cliente
    }

    public static BufferedReader creerReader(Socket socketVersUnClient)
    throws IOException {
        //créé un BufferedReader associé à la Socket
    }
}
```

```

    public static PrintWriter creerPrinter(Socket socketVersUnClient) throws
    IOException {
        //créé un PrintWriter associé à la Socket
    }

    public static String recevoirMessage(BufferedReader reader) throws
    IOException {
        //Récupérer une ligne
        //Retourner la ligne lue ou null si aucune ligne à lire.
    }

    public static void envoyerMessage(PrintWriter printer, String message)
    throws IOException {
        //Envoyer le message vers le client
    }
}

```

Question 2

Testez votre programme avec netcat (nc) ou putty (sur Windows) ou telnet.

Question 3

On vous demande d'écrire le code client dont le squelette est le suivant :

```

//...

package istic.pr.socket.tcp.echo;

public class ClientTCP {

    public static void main(String[] args) {
        //créer une socket client

        //créer reader et writer associés

        //Tant que le mot «fin» n'est pas lu sur le clavier,

        //Lire un message au clavier

        //envoyer le message au serveur

        //recevoir et afficher la réponse du serveur
    }

    public static String lireMessageAuClavier() throws IOException {
        //lit un message au clavier en utilisant par exemple un BufferedReader
        //sur System.in
    }

    public static BufferedReader creerReader(Socket socketVersUnClient)
    throws IOException {
        //identique serveur
    }
}

```

```

    }

    public static PrintWriter creerPrinter(Socket socketVersUnClient) throws
    IOException {
        //identique serveur
    }

    public static String recevoirMessage(BufferedReader reader) throws
    IOException {
        //identique serveur
    }

    public static void envoyerMessage(PrintWriter p, String message) throws
    IOException {
        //identique serveur
    }
}

```

Question 4

Faites en sortes que votre serveur supporte l'arrêt brutal du client sans planter en gérant correctement les exceptions dans traiterSocketCliente.

EXERCICE 2 : AJOUT DU NOM

Vous vous appuyerez sur les méthodes précédentes pour répondre à ces questions.

Copiez votre package et renommez votre nouveau package pour `istic.pr.socket.tcp.nom`

Question 1

Modifiez votre client pour qu'il prenne en paramètre un nom (`args[0]` dans `main`). Ecrire la méthode suivante :

```

public static void envoyerNom(PrintWriter printer, String nom) throws
IOException {
    //envoi « NAME: nom » au serveur
}

```

Modifiez la méthode `main` pour envoyer le nom à la première connexion.

Question 2

Ajoutez la méthode suivante au serveur :

```

public static String avoirNom(BufferedReader reader) throws IOException
{
    //retourne le nom du client (en utilisant split de la classe String par exemple)
}

```

Modifiez le code de la méthode `traiterSocketCliente` pour faire afficher le nom de l'utilisateur à chaque nouveau message. Le client doit obtenir des messages formatés sur le modèle suivant (écritures en vert, réponse en

noir):

```
test1
bob>test1
test2
bob>test2
```

Question 3

Modifiez le code de votre serveur pour envoyer un message au client en cas d'erreur sur le nom (il est vide ou la commande "NAME:" n'a pas été envoyé)

EXERCICE 3 : ENCODAGE DES CARACTÈRES

Copiez votre package et créez un nouveau package `istic.pr.socket.tcp.charset`

On souhaite gérer l'encodage des caractères coté serveur et client.

Question 1

Dans le client et le serveur, modifiez les méthodes `creerReader` et `creerPrinter` pour prendre en compte un charset.

```
public static BufferedReader creerReader(Socket socketVersUnClient,
String charset) throws IOException {
    //créer un reader qui utilise le bon charset
}

public static PrintWriter creerPrinter(Socket socketVersUnClient, String
charset) throws IOException {
    //créer un printer qui utilise le bon charset
}
```

Modifiez le code pour que ce charset soit utilisé dans tous les appels.

Question 2

Modifiez votre client et votre serveur pour qu'ils prennent en paramètre un charset (`args[1]` dans `main`).

Question 3

Essayez votre programme avec différents charset et vérifiez qu'il fonctionne.

Question 4

Vous testerez avec un charset UTF-16 du coté client et un charset UTF-8 coté serveur et vérifierez que le programme ne fonctionne plus.

EXERCICE 4 : GESTIONS DE PLUSIEURS CLIENTS

Copiez votre package et créez un nouveau package `istic.pr.socket.tcp.thread`

Question 1

On vous demande de modifier votre serveur pour prendre en compte plusieurs clients à la fois. Pour ce faire, vous déclarez une classe `TraiteUnClient` qui implémente `Runnable`.

Vous vous calquerez sur le modèle donné en cours pour le `ServeurTemps`. Comme la méthode `traiteSocketClient` de `Serveur` est statique vous pouvez l'appeler dans la méthode `Run` de votre classe `TraiteUnClient`.

La méthode `run` fait une ligne (elle appelle la méthode statique précédemment définie)

⚠ Vous ferez attention dans le cas où vous aviez utilisé un `try-with-resources` dans le main du serveur à ce qu'il ne ferme pas immédiatement la socket avant le traitement par le Thread. La socket cliente ne doit être fermée qu'à la fin du Thread.

On vous recommande d'utiliser un `ExecutorService` pour lancer vos Thread. Vous pourrez par exemple utiliser un `FixedThreadPool` afin de limiter le nombre de Threads actifs.

Question 3

Testez avec telnet que votre serveur fonctionne

Question 4

Testez avec plusieurs clients que votre serveur fonctionne.

Question 5

Adaptez votre programme pour utiliser un pool de threads fixe comme indiqué en cours (voir la javadoc de `Executors` notamment)

EXERCICE 5 : UN VRAI CHAT

Copiez votre package et créez un nouveau package `istic.pr.socket.tcp.chat`

Il faut maintenant envoyer les messages reçus vers tout le monde.

On modifie le serveur pour ajouter une liste statique des printer vers les sockets actives. Elle nous permettra d'envoyer des informations à tous les clients à la fois.

```
private static List<PrintWriter> printerSocketActives = new ArrayList<PrintWriter>();
```

Question 1

On vous demande d'ajouter deux méthodes :

```
public static synchronized void ajouterPrinterSocketActives(PrintWriter  
printer) {  
    //ajouter le printer à la liste
```

```
}

public static synchronized void enleverPrinterSocketActives(PrintWriter
printer) {
    //enlever le printer à la liste
}
```

Question 2

Modifiez le code de `traiterSocketClient` pour :

- ajouter le printer de la socket à la liste
- enlever le printer de la socket juste avant que la socket soit fermée.

Question 3

Ecrire la méthode suivante :

```
public static synchronized void envoyerAToutesLesSocketsActive(String
message) throws IOException {
    //envoie le message à toutes les sockets actives
}
```

Question 4


Modifiez le code de `traiterSocketClient` pour qu'elle envoie ses messages à tous les clients (et plus seulement à l'émetteur) en utilisant la méthode précédente.

Question 5

Testez votre programme avec `telnet` et différents clients utilisant des charsets différents. Le code des clients n'est pas adapté, pourquoi ?

Question 6

Proposez une solution pour modifier le client de façon à pouvoir faire les lectures sur la socket en parallèle de la saisie clavier.

 **Attention** : la lecture sur le clavier doit toujours être effectuée par le Thread main.

Question 7

Testez avec l'ordinateur d'un voisin que vos programmes peuvent communiquer.

PARTIE 3 : CHAT UDP MULTICAST

On vous demande d'écrire le même chat en utilisant l'API Multicast de Java sur les sockets. Vous vous référerez à l'exemple du serveur temps du cours.

Les caractéristiques sont les suivantes :

- il n'y a pas de serveur. Une seule classe de nom ChatMulticast.java
- Les messages s'échangent sur le canal **225.0.4.Y** Mettez-vous d'accord avec les autres groupes pour ne pas utiliser la même adresse pendant vos tests. Vous pourrez tester ensuite avec les autres groupes sur la même adresse.
- pas de gestion de charset
- le nom doit être envoyé à chaque envoi de message « bob> message »
- votre classe possédera les méthodes statiques suivantes :
 - envoyerMessage(MulticastSocket s, String message) ;
 - String recevoirMessage(MulticastSocket s) qui bloque tant qu'un message n'est pas reçu.
 - lireMessageAuClavier sur le même modèle que pour le chat TCP.
- vous devrez sans doute utiliser un Thread pour être capable d'envoyer et de recevoir simultanément.
- la taille maximale des messages est de 1024.