

TP9 - Sockets UDP et TCP en Java

Dans ce TP, vous allez mettre en œuvre un client et un serveur qui vont communiquer via des sockets, avec deux programmes écrits en Java. Comme il s'agit ici de faire communiquer deux machines, il faudra que vous ayez deux consoles ouvertes dans le dossier courant, une pour lancer le serveur, l'autre pour lancer le client.

Dans la dernière partie, vous écrirez un client HTTP, qui se connectera à un site Web distant.

1 Travail préliminaire

Le travail se fera dans une VM Debian. Le plus simple est d'utiliser Vagrant, les lignes suivante permettent en quelques minutes d'avoir une machine fonctionnelle (mais sans bureau, imposant l'usage de nano comme éditeur...) :

```
$ mkdir ~/tpjava; cd ~/tp_java
$ vagrant init debian/bullseye64
$ vagrant up
$ vagrant ssh
```

A défaut, vous pouvez utiliser une VM Debian sous VirtualBox (ce qui permet d'avoir un bureau graphique d'un confort appréciable).

Dans tous les cas, il faudra installer dans la machine les paquets permettant la compilation et l'exécution de programmes Java : `default-jdk` (pour *Java Development Kit*) pour pouvoir compiler, et `default-jre` (pour *Java Runtime Environment*) pour pouvoir exécuter les programmes.

Une fois l'environnement fonctionnel, tester en saisissant dans un fichier `Monprog.java` le code dans l'autre document. Compiler avec `$ javac *.java` et une fois les erreurs corrigées, exécuter le programme en lui passant des arguments.

Par exemple : `$ java Monprog un "2 3 4" 55 six` et observez le résultat.

2 Communication en mode UDP

Vous allez vérifier le mode UDP en écrivant un **serveur**, qui attend et affiche tout ce qu'il reçoit sur un port donné, et un **client**, qui ne fait qu'envoyer une chaîne de caractères au serveur.

Dans un dossier TP9, créer deux fichiers `ClientUDP.java` et `ServeurUDP.java` et y créer les deux programmes (fonction `main()`). Vous devrez ajouter en tête du fichier les imports suivants :

```
import java.io.*;
import java.net.*;
```

Pour le client, le contenu du "main" sera :

```
InetAddress addr = InetAddress.getLocalHost();
System.out.println( "adresse=" +addr.getHostName() );
...
DatagramPacket packet = new DatagramPacket( data, data.length, addr
, 1234 );
DatagramSocket sock = new DatagramSocket();
sock.send(packet);
sock.close();
```

La variable `data` sera de type "tableau de byte" et on copiera dedans une chaîne de caractère déclarée par `String s="Hello World";` avec la méthode `getBytes()` :

```
byte[] data = s.getBytes();
```

Pour le serveur, le contenu du "main" sera :

```
DatagramSocket sock = new DatagramSocket(1234);
while(true)
{
    System.out.println( "-Waiting data" );
    DatagramPacket packet = new DatagramPacket(new byte[1024], 1024);
    sock.receive(packet);
    String str = new String(packet.getData() );
    System.out.println( "str=" + str );
}
```

Compiler et vérifier le fonctionnement en lançant le serveur en premier. A chaque lancement du client, vous devrez avoir l'affichage de la chaîne dans la console du serveur.

Questions

Q2.1 - Que se passe-t-il si on lance le client sans que le serveur ne soit actif :

Q2.2 - Quel est le numéro de port utilisé : _____

Chercher si cela est un choix pertinent. Pourquoi ? : _____

Q2.3 - Modifier les deux programmes pour que le serveur renvoie la chaîne reçue au client, et que celui-ci attende cette chaîne et l'affiche avant de quitter.

3 Communication en mode TCP

En sus des classes dédiées aux sockets (cf. cours), on utilisera les flots de données `DataInputStream` et `DataOutputStream` pour y placer les données.

Se rendre sur <https://docs.oracle.com/javase/8/docs/api/overview-summary.html> et chercher ces deux classes, via l'index.

3.1 Communication unidirectionnelle

Créer deux fichiers `ClientTCP1.java` et `ServeurTCP1.java`

Dans le serveur, insérer les lignes suivantes dans un bloc "try-catch" dans le `main()` :

```
// ServeurTCP1.java
ServerSocket socketserver = new ServerSocket( 2016 );
System.out.println( "serveur en attente" );
Socket socket = socketserver.accept();
System.out.println( "Connection d'un client" );
DataInputStream dIn = new DataInputStream( socket.getInputStream() );
System.out.println( "Message: " + dIn.readUTF());

socket.close();
socketserver.close();
```

Pour que le client puisse envoyer des données au serveur, on va instancier un objet du type `DataOutputStream`, qu'on va associer au socket. On pourra ensuite transmettre des chaînes de caractères via sa méthode `writeUTF()`.

```
// ClientTCP1.java
Socket socket = new Socket( "localhost", 2016 );
DataOutputStream dOut = new DataOutputStream( socket.
    getOutputStream() );
dOut.writeUTF( "message test" );
socket.close();
```

Compiler et vérifier le fonctionnement : à chaque lancement du client, le serveur doit afficher le message et s'arrêter.

3.2 Serveur statique

Copiez les deux programmes en ClientTCP2.java et ServeurTCP2.java.

De façon à ce que le serveur reste actif, insérez le code du serveur (sauf la création et l'instanciation de la variable socketserver) dans une boucle infinie : `while(true) { ... }`.

Pour le client, la chaîne envoyée sera transmise via le premier argument de la ligne de commande :

```
dOut.writeUTF( args[0] );
```

Vérifiez le fonctionnement : lancer le serveur, puis lancer plusieurs fois le client en donnant des chaînes différentes (`java ClientTCP2 coucou`, etc.).

3.3 Communication bidirectionnelle

Renommer les deux fichiers précédents en ClientTCP3.java et ServeurTCP3.java et ajouter le code nécessaire pour que le serveur renvoie au client la chaîne reçue mais inversée. Ce dernier devra l'afficher.

Pour inverser une chaîne `msg` en Java, on utilisera :

```
String rev = new StringBuilder(msg).reverse().toString();
```

4 Client HTTP via socket TCP

Vous allez écrire un client HTTP permettant d'accéder à une page Web via son URL. Le contenu (code HTML) de la page téléchargée sera simplement affiché à l'écran.

Saisir le code ci-dessous dans le `main()` d'une classe `Clienthttp`. Ajouter le code pour ouvrir le socket sur le port 80, avec le nom d'hôte dans le 1^{er} argument de la ligne de commande (`args[0]`). Il faudra également insérer tout le code dans un bloc `try...catch` .

```

OutputStreamWriter osw = new OutputStreamWriter( socket.
    getOutputStream() );
InputStreamReader isw = new InputStreamReader( socket.
    getInputStream() );

BufferedWriter bufOut = new BufferedWriter( osw );
BufferedReader bufIn = new BufferedReader( isw );

String request = "GET / HTTP/1.0\r\n\r\n"; // requete HTTP
bufOut.write( request, 0, request.length() );
bufOut.flush();

String line = bufIn.readLine(); // lecture ligne par ligne
while( line != null ) {          // tant qu'il y a des donnees
    recues,
    System.out.println( line );   // ... les afficher
    line = bufIn.readLine();
}
bufIn.close();
bufOut.close();
socket.close();
}

```

Q4.1 - Vérifiez le fonctionnement en tapant : `java Clienthttp www.univ-rouen.fr`

Redirigez la sortie vers un fichier `ur.txt` via le shell (avec `>`), puis l'ouvrir avec un éditeur texte. Le fonctionnement est-il correct ? : _____

Q4.2 - Le contenu du fichier est-il conforme à ce qui est attendu ? (cf. Cours sur protocole HTTP)

Q4.3 - La sortie est-elle constituée uniquement d'un fichier html conforme ? Pourquoi ?

Q4.4 - Essayer ensuite avec le site `www.javaworld.com` Cela fonctionne-t-il correctement ? (comparez avec ce que vous obtenez via un navigateur) _____

Essayez ensuite avec d'autres URLs.

Q4.5 - A votre avis, pourquoi obtient-on des résultats différents d'un navigateur ? (relire le cours sur le protocole HTTP)

Q4.6 - Analyser la réponse de `www.univ-rouen.fr` (voir le fichier `ur.html`) et observez les en-têtes. En comparant avec la page `en.wikipedia.org/wiki/List_of_HTTP_header_fields`, indiquez ci-dessous les champs renvoyés par le serveur et indiquez leur sens :