

Réseau en Java

par **Humbert Florent**

Date de publication : 16/10/2007

Dernière mise à jour : 16/10/2007

Ce cours introduira les notions pour permettre la gestion du réseau en Java à l'aide de la bibliothèque standard Java. Notamment pour la gestion des communications du côté serveur et du côté client suivant les protocoles TCP et UDP.

- I - Introduction
- II - Adressage
 - II-A - Exemple : Obtention de son adresse IP
 - II-B - Exercices
 - II-B-1 - Obtenir une adresse IP à partir d'un nom d'hôte
 - II-B-2 - Validité d'un nom d'hôte
 - II-C - Récapitulatif
- III - Premières communications
 - III-A - Socket côté client
 - III-A-1 - La classe Socket
 - III-A-2 - Meilleure gestion des flux
 - III-A-2-a - BufferedOutputStream et BufferedInputStream
 - III-A-2-b - BufferedWriter et BufferedReader
 - III-A-3 - Exercices
 - III-A-3-a - Obtention de notre IP internet
 - III-A-3-b - Réel différence entre flux bufferisé et non bufferisé ?
 - III-A-3-c - Application équivalente à netcat ou à telnet
 - III-A-4 - Récapitulatif
 - III-B - Socket côté serveur
 - III-B-1 - La classe ServerSocket
 - III-B-2 - Exercices
 - III-B-2-a - Serveur écoutant les paquets envoyés par un client
 - III-B-2-b - Suite de l'exercice
 - III-B-3 - Récapitulatif
- IV - Réseau avancé
 - IV-A - Utilité et gestion des timeout
 - IV-B - Communication à travers un proxy
 - IV-C - Protocole UDP et utilisation des datagrammes
 - IV-C-1 - Classes DatagramSocket et DatagramPacket
 - IV-C-2 - Premiers pas
 - IV-C-2-a - Serveur
 - IV-C-2-b - Client
 - IV-C-3 - Ping/Pong complet
 - IV-C-3-a - Serveur
 - IV-C-3-b - Client
 - IV-D - Comment réaliser une application réseau monothread
 - IV-D-1 - Sockets non bloquants
 - IV-D-2 - Selector
- V - Ressources
- VI - Remerciements

I - Introduction

Même si la plupart des applications développés en java repose sur des bibliothèques de haut-niveau, il est parfois utile d'utiliser les *socket* à bas niveau.

Depuis la sortie de *java*, une *API* permettant de gérer le réseau est disponible dans le *JDK*, elle permet de gérer les adresses, les sockets clients, les sockets serveurs, les URL... Dans ce cours, nous verrons comment utiliser cette *API* en faisant parfois un lien avec la gestion du réseau bas-niveau.

Il est utile d'avoir des connaissances en réseau même si je tenterai au maximum de réexpliquer les notions utiles.

II - Adressage

Avant d'entamer la partie communication proprement dite, nous allons apprendre à utiliser les adresses *IP* en *java*.

Afin que deux applications communiquent (en utilisant le protocole IP), il est nécessaire de connaître l'adresse *IP* de l'autre machine. L'*API* standard permet de les manipuler en utilisant la classe **InetAddress**. Cette classe permet de manipuler aussi bien les adresses *IPv4* que les adresses *IPv6*. Ceci correspond à deux normes différentes, la plupart des *IP* rencontrés sont en *IPv4* et ont une notation sous la forme **127.0.0.1**.

II-A - Exemple : Obtention de son adresse IP

La classe **InetAddress** dispose d'une méthode statique **getLocalHost()** qui permet de récupérer son adresse IP et des méthodes **getHostName()** et **getHostAddress()** qui permettent respectivement de récupérer le nom d'hôte et l'adresse sous forme pointée.

```
package developpez;

import java.net.InetAddress;
import java.net.UnknownHostException;

/**
 * @author millie
 */
public class TestInetAddress {

    /**
     * @param args
     * @throws UnknownHostException
     */
    public static void main(String[] args) throws UnknownHostException {
        InetAddress address = InetAddress.getLocalHost();

        System.out.println("Nom d'hôte : " + address.getHostName());

        System.out.println("IP : " + address.getHostAddress());
    }
}
```

En exécutant ce code, vous allez obtenir un résultat qui dépend de votre machine et de votre réseau. La plupart du temps, vous n'obtiendrez pas votre vraie adresse internet, mais une adresse local en **127.0.0.1** ou un adresse de réseau local en **192.168.1.x**

En effet, la plupart du temps, une machine dispose d'au moins trois adresses :

- adresse IP local (127.0.0.1) dénommé **localhost**
- adresse IP réseau (fréquemment 192.168.x.x sur les réseaux maisons)
- adresse IP internet

Afin d'obtenir l'ensemble des adresses IP de la machine, il est nécessaire de lister l'ensemble des interfaces réseaux (classe **NetworkInterface**) et de lister ses adresses IP. Nous utiliserons la méthode statique **getNetworkInterfaces** de la classe **NetworkInterface** pour obtenir l'ensemble des interfaces réseaux.

```
package developpez;

import java.net.InetAddress;
import java.net.NetworkInterface;
import java.net.SocketException;
import java.net.UnknownHostException;
import java.util.Enumeration;

/**
 * @author millie
 */
public class TestInetAddress {

    /**
     * @param args
     * @throws UnknownHostException
     * @throws SocketException
     */
    public static void main(String[] args) throws Exception {
        //enumère l'ensemble des interfaces réseaux, typiquement une carte réseau
        Enumeration<NetworkInterface> interfaces = NetworkInterface.getNetworkInterfaces();

        while (interfaces.hasMoreElements()) {
            NetworkInterface currentInterface = interfaces.nextElement();

            //chaque carte réseau peut disposer de plusieurs adresses IP
            Enumeration<InetAddress> addresses = currentInterface.getInetAddresses();
            while (addresses.hasMoreElements()) {
                InetAddress currentAddress = addresses.nextElement();
                System.out.println(currentAddress.getHostAddress());
            }
        }
    }
}
```

Vous noterez que vous obtiendrez des adresses contenant le caractère :. Cela correspond à des adresses **IPv6**.

II-B - Exercices

II-B-1 - Obtenir une adresse IP à partir d'un nom d'hôte

En utilisant la méthode statique **getByName** de la classe **InetAddress**, cherchez à obtenir l'adresse IP de **www.developpez.com**.

II-B-2 - Validité d'un nom d'hôte

Dans l'exercice précédent, nous avons vu une manière d'obtenir l'IP à partir d'un nom d'hôte (ceci se faisant automatiquement à l'aide d'un serveur DNS). Ceci ne signifie pas que l'IP est accessible, utilisez la méthode **isReachable(int timeout)** pour savoir si un nom d'hôte est accessible.

Solutions

II-C - Récapitulatif

Pour plus d'informations sur la classe **InetAddress**, je vous invite à regarder la *javadoc* de cette classe. Voici une description des méthodes les plus utiles :

Méthodes static

- **static InetAddress getByAddress(byte[] addr)** permet d'obtenir un **InetAddress** à partir d'un champ de bytes (4 pour l'IPv4)
- **static InetAddress getByName(String host)** permet d'obtenir l'adresse à partir du nom d'hôte
- **static InetAddress getLocalHost()** permet d'obtenir l'adresse de l'hôte local

Méthodes de classe

- **byte[] getAddress()** retourne un champ de byte correspond à l'adresse
- **String.getHostAddress()** retourne l'adresse sous forme pointée
- **String.getHostName()** retourne le nom d'hôte
- **boolean isReachable(int timeout)** permet de savoir si une adresse est atteignable

III - Premières communications

Afin de communiquer entre plusieurs machines de manière *simple*, les informaticiens ont définis deux protocoles réseaux couramment utilisés de nos jours, le protocole **TCP** et le protocole **UDP**.

De manière assez sommaire, le protocole **TCP** a été créé pour permettre d'envoyer des données de manière fiable par le réseau, notamment en :

- s'assurant que les données arrivent au destinataire, en cas d'échec de transmission, l'ordinateur émetteur doit être mis au courant
- s'assurant que les données arrivent dans le bon ordre
- définissant une connexion entre les machines

Le protocole **UDP** ajoute très peu au protocole **IP**. Par rapport au protocole **TCP**, **UDP** est peu fiable, il n'est pas certain que les données arrivent et il n'est pas certain que les données arrivent dans le bon ordre. **TCP** effectue un nombre important d'allée et retour, ce qui a l'inconvénient de faire diminuer la vitesse de connexion. De nos jours, **TCP** est quasiment tout le temps utilisé, sauf éventuellement dans un réseau local où les pertes de paquets réseaux ne doivent pas arriver.

Java propose plusieurs classes pour manipuler ces protocoles de manière absolument transparente. La classe **Socket** repose sur le protocole **TCP** et permet donc de transmettre des données de manière fiable. La classe **DatagramSocket** quant à elle repose sur **UDP**.

III-A - Socket côté client

La plupart des systèmes d'exploitations propose une abstraction pour manipuler des données réseaux que l'on nomme *socket*. Du point de vue bas-niveau, un *socket* se comporte de la même manière qu'un fichier (que l'on manipule généralement via ce que l'on appelle un *file descriptor*). Une fois une connexion établie, la gestion des sockets est similaire à la gestion des fichiers.

En Java a été introduit une classe **Socket** qui permet de manipuler à plus haut-niveau cette abstraction du système d'exploitation. Il existe deux méthodes : **getOutputStream()** et **getInputStream()** qui va vous permettre de manipuler les données comme pour un fichier.

Java fait la différence entre *socket* côté client (objet qui permet de se connecter à des ordinateurs distants) et *socket* côté serveur (objet qui permet d'attendre des demandes de connexions de l'extérieur). La classe **Socket** correspond à un *socket* client et la classe **ServerSocket** correspond à un *socket* serveur.

III-A-1 - La classe Socket

Dans un premier temps, nous allons voir comment nous connecter à un serveur distant, comment envoyer des données et comment en recevoir de manière assez simple.

```
Socket s = new Socket("www.developpez.com", 80);

//ou
InetAddress address = InetAddress.getByName("www.developpez.com");
Socket s2 = new Socket(address, 80);
```

Le code précédent permet de se connecter à un serveur nommé par son IP ou par son nom d'hôte sur un port particulier (ici le port 80). La notion de port, qui est purement virtuelle, a été définie au niveau de la couche **TCP** et **UDP** pour permettre de ne pas mélanger les données envoyés à une personne. Par exemple sur un chat, si vous souhaitez parler à quelqu'un et envoyer des fichiers à cette même personne en même temps, il est nécessaire de séparer les données. Ceci étant fait en utilisant deux ports différents pour les deux traitements.

Il est également possible de réutiliser la classe **InetAddress**.

Nous allons à présent envoyer des données en utilisant un *OutputStream*.

```
Socket s = new Socket("www.developpez.com", 80);

String g = "GET / HTTP/1.1\n" +
    "Host: www.developpez.com\n\n";

OutputStream oStream = s.getOutputStream();
oStream.write(g.getBytes());
```

La chaîne *g* utilise en réalité un protocole de plus haut niveau (protocole HTTP) qui permet de récupérer la page d'index du serveur. Ainsi, on s'attendrait à obtenir la page d'accueil suite à la demande au serveur. Evidemment, à ce moment, nous n'avons pas encore traité les données renvoyés par le serveur, nous utiliserons donc un *InputStream*.

```
InputStream iStream = s.getInputStream();

byte[] b = new byte[1000]; //définition d'un tableau pour lire les données arrivées
int bitsRecus = iStream.read(b); //il n'est pas sûr que l'on receive 1000 bits
if(bitsRecus>0) {
    System.out.println("On a reçu : " + bitsRecus + " bits");
    System.out.println("Reçu : " + new String(b,0, bitsRecus));
}
```

 Nous avons utilisé le constructeur **public String(byte bytes[], int offset, int length)** de *String* pour indiquer la bonne taille de la chaîne.

Normalement, vous devriez obtenir à ce stade quelque chose du genre :

```
On a reçu : 1000 bits
HTTP/1.1 200 OK
Date: Wed, 17 Oct 2007 11:57:24 GMT
Server: Apache/2.2.4 (Unix) PHP/4.4.6
X-Powered-By: PHP/4.4.6
Transfer-Encoding: chunked
Content-Type: text/html

3c82

<html>
<head>
...
```

Nous voyons donc une partie de la page **html** qui commence. C'est un bon début. La méthode **read** retourne le nombre d'octet lu. Si -1 est renvoyé, c'est que la communication est terminée et que vous avez fini de lire. Hors, nous n'avons pas lu toutes les données, nous pouvons à la place utiliser :

```
int bitsRecus = 0;
```



```
while((bitsRecus = iStream.read(b)) >= 0) {  
    System.out.println("On a reçu : " + bitsRecus + " bits");  
    System.out.println("Reçu : " + new String(b, 0, bitsRecus));  
}
```

Lorsque l'on crée et que l'on connecte un *socket*, on a acquis des ressources donné par le système d'exploitation, il est nécessaire de les rendre au système en utilisant la méthode **close**, aussi bien sur les flux que sur les sockets. Cela est également utile pour la communication car vous indiquez que vous avez terminé de parler. Un code plus complet pourrait donc être le suivant :

```
public static void main(String[] args) throws IOException {  
    Socket s = new Socket("www.developpez.com", 80);  
    //recupération des flux  
    OutputStream oStream = s.getOutputStream();  
    InputStream iStream = s.getInputStream();  
  
    byte[] b = new byte[1000];  
    String g = "GET / HTTP/1.1\n" + "Host: www.developpez.com\n\n";  
  
    try {  
        oStream.write(g.getBytes());  
  
        int bitsRecus = 0;  
        while((bitsRecus = iStream.read(b)) >= 0) {  
            System.out.println("On a reçu : " + bitsRecus + " bits");  
            System.out.println("Reçu : " + new String(b, 0, bitsRecus));  
        }  
    } catch (Exception e) {  
        e.printStackTrace();  
    } finally {  
        //fermeture des flux et des sockets  
        oStream.close();  
        iStream.close();  
        s.close();  
    }  
}
```

Si vous remplacez la chaîne "Host: www.developpez.com\n\n" par la chaîne "Host: www.developpez.com\n" (simple retour à la ligne), vous vous rendrez compte que le programme s'arrête au niveau de l'appel à la méthode **read**. En effet, par défaut, la **lecture est bloquante**, tant que l'on a pas reçu de données, le thread courant s'endort et ne se réveille que lorsque le serveur enverra des données.

III-A-2 - Meilleure gestion des flux

Nous avons vu que pour communiquer avec une autre machine, il était nécessaire d'utiliser des flux d'entrée et de sortie. Les appels à **read** et à **write** sont particulièrement gourmand en temps (si l'on regarde les sources d'openJDK, on voit que **read** et **write** font des appels systèmes ce qui prend en général du temps). Java introduit des classes intéressantes qui permettent de s'abstraire de cela.

III-A-2-a - BufferedOutputStream et BufferedInputStream

Les classe **BufferedOutputStream** et **BufferedInputStream** permettent de manipuler plus efficacement les flux standards réseaux.

Les classes bufferisées fonctionnent pour l'utilisateur, presque de la même manière que les classes non bufferisées, ces classes sont héritées de **OutputStream** et de **InputStream**. Il est possible d'en créer en utilisant directement les flux normaux :

```
oStream = s.getOutputStream();
iStream = s.getInputStream();

BufferedOutputStream bOStream = new BufferedOutputStream(oStream);
BufferedInputStream bIStream = new BufferedInputStream(iStream);

bOStream.write(g.getBytes());
```

A ce niveau, vous vous rendrez compte que le code ne fonctionne plus. En effet, comme le flux de sortie est bufferisé, il n'est pas immédiatement envoyé au serveur, pour forcer l'envoi, il est nécessaire d'utiliser la méthode : **flush** de la classe **BufferedOutputStream**. En tout, nous avons :

```
// récupération des flux
oStream = s.getOutputStream();
iStream = s.getInputStream();

BufferedOutputStream bOStream = new BufferedOutputStream(oStream);
BufferedInputStream bIStream = new BufferedInputStream(iStream);

bOStream.write(g.getBytes());
bOStream.flush();

int bitsRecus = 0;
while((bitsRecus = bIStream.read(b)) >= 0) {

    System.out.println("On a recu : " + bitsRecus + " bits");
    System.out.println("Recu : " + new String(b, 0, bitsRecus));
}

bOStream.close();
bIStream.close();
```

A ce moment, vous vous rendrez peut être compte que l'exécution de votre programme est plus rapide.

III-A-2-b - BufferedWriter et BufferedReader

Pour la lecture et l'écriture de donnée textuelle, java fournit les classes **BufferedWriter** et **BufferedReader** qui se construisent à partir de **OutputStreamReader** et de **InputStreamReader**. Cela offre les avantages des flux bufferisés et des méthodes supplémentaires pour gérer directement les chaînes de caractères.

La classe **BufferedReader** dispose notamment d'une méthode **readLine()** qui permet de lire une ligne (donc finissant par \n). La classe **BufferedWriter** dispose d'une méthode **write** qui permet d'écrire directement une chaîne de caractère (au format **String**). Le code précédent peut donc devenir :

```
// récupération des flux

BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(
    s.getOutputStream()));
BufferedReader reader = new BufferedReader(new InputStreamReader(
    s.getInputStream()));

writer.write(g);
writer.flush();
```

```
String ligne;
while((ligne = reader.readLine()) != null) {

    System.out.println("Recu : " + ligne);
}

writer.close();
reader.close();
```

III-A-3 - Exercices

A cet étape, vous avez les briques de base pour écrire des applications clientes. Je vous propose un ensemble d'exercice pour vous familiariser avec les **Socket**.

III-A-3-a - Obtention de notre IP internet

Nous avons vu dans le chapitre **InetAddress** une manière de récupérer l'ensemble de nos adresses IP. Il existe une autre méthode utilisant les sockets qui permet de savoir quel est l'IP que l'on utilise sur *internet*. Pour cela, utiliser la méthode **getLocalAddress()** de la classe **Socket**. A noter que cette méthode peut faire défaut lorsque l'on est derrière un routeur (box).

Solution

III-A-3-b - Réel différence entre flux bufferisé et non bufferisé ?

Dans le chapitre précédent, nous avons signalé des performances importantes entre les classes **BufferedInputStream** et **InputStream** dans le cas général. Pour vérifier cela, écrivez une méthode prenant en entrée un **InputStream** et lisant le flux caractère par caractère (en utilisant un tableau de taille 1). Tester avec un **BufferedInputStream** et l'**InputStream** directement obtenu avec la méthode **getInputStream()**

Solution

III-A-3-c - Application équivalente à netcat ou à telnet

Lorsque l'on développe une application serveur, on utilise parfois une application qui permet d'envoyer à la main des données au serveur. Typiquement *telnet* sous windows ou *socket* sous Unix.

Le but de l'exercice est de réaliser une application se connectant à un serveur et composé de 2 threads :

- un thread écoute les entrées au clavier, dès que la touche entrée est pressée, la ligne est envoyée au serveur
- un thread écoute le serveur et écrit sur le flux standard de sortie dès qu'il reçoit une ou plusieurs lignes

Solution

III-A-4 - Récapitulatif

Voici la liste des méthodes les plus couramment utilisées :

- **Socket()** : création d'un socket non connecté
- **Socket(String host, int port)** : Création d'un socket connecté
- **void close()** : fermeture du socket
- **void connect(SocketAddress endpoint)** : connexion à un serveur
- **InetAddress getInetAddress()** : Permet d'obtenir l'adresse auquel est connecté le socket
- **InetAddress getLocalAddress()** : Permet d'obtenir l'adresse local qu'utilise votre machine pour se connecter au serveur
- **int getLocalPort()** : Permet d'obtenir le port local qu'utilise la machine
- **int getPort()** : Permet d'obtenir le port (serveur) avec laquelle la machine s'est connectée
- **InputStream getInputStream()** : Obtention d'un flux d'entrée
- **OutputStream getOutputStream()** : Obtention d'un flux de sortie
- **void setSoTimeout(int timeout)** : Régle le timeout (en millisecondes)

III-B - Socket côté serveur

Jusqu'à maintenant, nous n'avons vu que les communications vers un serveur. Il manque donc un outil qui permettrait de créer soi-même un serveur en attendant des demandes de connexion de l'extérieur.

III-B-1 - La classe ServerSocket

java propose une classe nommée **ServerSocket** qui est dédiée à l'attente de demande de connexion de la part d'une autre machine.

Dans la littérature, vous rencontrerez souvent le terme *bind*. Cela correspond à une assignation d'une adresse et d'un port à un socket. En effet, pour pouvoir écouter les demandes de connexions, il est nécessaire d'assigner une adresse et un port d'écoute.

```
ServerSocket server = new ServerSocket(300);  
server.close();
```

Ce code correspond à la création d'un **ServerSocket** assigné à l'adresse locale et au port 300. La méthode **accept()** attend jusqu'à obtenir une connexion extérieur.

```
ServerSocket server = new ServerSocket(300);  
Socket client = server.accept();  
  
client.close();  
server.close();
```

Vous pouvez exécuter ce code, vous verrez que le programme se met en attente. Dans une console DOS (ou Unix), tapez : **telnet localhost 300**, vous verrez que le programme se termine. En effet, la connexion a été établie et vous disposez d'une classe **Socket** pour communiquer.

De nombreuses applications serveurs créait un thread dédié à l'écoute des demandes de connexion et crée un nouveau thread par client. Il est toutefois possible de n'utiliser qu'un thread en tout.

Maintenant, vous disposez des briques de base pour créer un serveur relativement simple, en effet, le reste des communication se faisait à partir d'une classe **Socket**.

III-B-2 - Exercices

III-B-2-a - Serveur écoutant les paquets envoyés par un client

Ecrivez une application qui attend un unique client et qui écrit sur le flux standard tout ce que le client a envoyé.

Vous pouvez tester avec **telnet/socket** ou en tapant dans un navigateur : **http://localhost:300** si vous attendez sur le port 300.

Solution

III-B-2-b - Suite de l'exercice

Vous pouvez remarquer que si vous faites plusieurs demandes de connexion sur votre ancien serveur, cela ne fonctionne pas. Proposez une solution qui permet de gérer plusieurs clients en même temps.

Solution

III-B-3 - Récapitulatif

Voici les méthodes les plus couramment utilisés :

- **ServerSocket(int port)** : crée un socket serveur écoutant sur un certain port
- **Socket accept()** : attente de connexion d'un client
- **void close()** : fermeture du socket
- **void setSoTimeout(int timeout)** : spécifie un timeout (en millisecondes) lors des demandes de connexion

IV - Réseau avancé

IV-A - Utilité et gestion des timeout

Lorsque l'on crée un serveur, il est nécessaire qu'il soit robuste, ce qui est en général une grande difficulté. Imaginons que l'on s'en tienne à ce que l'on a vu précédemment, c'est à dire que l'on cherche à lire sur un socket jusqu'à ce que la méthode **read** retourne -1. Il est possible que le client cherche à envoyer de nombreuses données mais sans fermer le socket. Il est alors possible que le serveur soit saturé. L'API java introduit des méthodes permettant de gérer des *timeout*. Si le temps indiqué est dépassé, alors la méthode *read* lance une exception.

Les classes **Socket** et **ServerSocket** dispose d'une méthode : **setSoTimeout(int)** qui spécifie le timeout en millisecondes. Si le timeout arrive à son échéance, les méthodes lancent une exception **SocketTimeoutException**. Par exemple :

```
Socket s = new Socket("www.developpez.com", 80);
s.setSoTimeout(1000);

InputStream iStream = s.getInputStream();
byte[] b = new byte[1000];
try {
    iStream.read(b);
}
catch (SocketTimeoutException e) {
    System.err.println("Timeout");
}
catch (IOException e) {
    e.printStackTrace();
}
finally {
    iStream.close();
    s.close();
}
```

 A noter qu'une fois le timeout positionné, il n'est pas utile de le redéfinir à chaque appel à **read**, à **write** ou à **accept**.

IV-B - Communication à travers un proxy

Cette partie a été fortement inspirée de la **FAQ**.

La méthode la plus simple consiste à modifier les paramètres lors du lancement de la machine virtuelle (il n'est donc pas nécessaire de modifier le code source de votre programme)

```
java -DproxySet=true -DproxyHost=nomproxy -DproxyPort=numport test
```

Une deuxième méthode consiste à modifier les propriétés du système via la méthode **getProperties()** de la classe **System**

```
Properties prop = System.getProperties();
String ipProxy = "127.0.0.1";
String portProxy = "80";
```

```
prop.put("http.proxyHost", ipProxy);  
prop.put("http.proxyPort", portProxy);
```

IV-C - Protocole UDP et utilisation des datagrammes

La plupart des applications réseau se base sur le protocole TCP (donc sur des classes **Sockets**) pour communiquer. Il peut être parfois nécessaire d'utiliser un protocole un peu plus simple et plus rapide dans certains cas : le protocole *UDP*.

La communication entre deux machines avec le protocole UDP se fait en envoyant des paquets (cela ne fonctionne donc pas comme des flux). Ce protocole est considéré comme non fiable et fonctionne en mode non connecté. Cela signifie que :

- il n'y a pas qu'un unique channel (socket) pour communiquer entre un unique client et un unique serveur
- il n'est pas sûr que les données arrivent à destination
- il n'est pas sûr que les paquets arrivent dans le bon ordre

Dans le protocole *TCP*, il y a de nombreux allées et retours entre le client et le serveur (basé sur un système d'accusé réception) pour être sûr que les données arrivent. Il arrive parfois que l'on souhaite communiquer des informations très rapidement et que la perte d'informations ne soit pas très importante. Ce qui est typiquement le cas des serveurs de streaming vidéo, de voix sur IP...

IV-C-1 - Classes DatagramSocket et DatagramPacket

Java offre deux classes qui permettent de manipuler des données en utilisant le protocole *UDP*. La classe **DatagramPacket** permet de forger des paquets de données et est utile en envoi et en réception. La classe **DatagramSocket** permet quand à elle d'envoyer un paquet.

Pour montrer l'utilisation de ces classes, nous allons prendre un exemple simple d'un serveur et d'un client qui réalisent un simple ping/pong. Le serveur attend des données des clients, lorsque le serveur en reçoit, il retourne une chaîne PONG au client.

IV-C-2 - Premiers pas

IV-C-2-a - Serveur

Tous d'abord, il est nécessaire de créer un **DatagramSocket** qui écoutera les données en provenance de l'extérieur sur un port particulier. Contrairement aux **sockets**, il n'y a pas de notion de **DatagramServerSocket**. Tout transite au même endroit.

```
DatagramSocket socket = new DatagramSocket(300);  
  
byte[] buffer = new byte[1500];  
DatagramPacket packet = new DatagramPacket(buffer, buffer.length);  
  
socket.receive(packet);  
System.out.println(new String(packet.getData(), 0, packet.getLength()));  
  
socket.close();
```

Dans le code précédent, nous créons un **DatagramSocket** qui s'occupe de récupérer toutes les données provenant du port 300. Nous fabriquons un paquet qui servira à recevoir les données extérieurs. Première chose que l'on peut remarquer, c'est qu'il n'y a pas de notion de client, le **DatagramSocket** peut recevoir des données de n'importe où (et non pas que d'un unique client comme avec la classe **Socket**). De plus, la méthode **receive** est bloquante et ne termine que lorsque des données sont arrivées.

 Pour connaître l'origine des données, il est nécessaire d'utiliser la méthode **getAddress()** sur un paquet pour obtenir l'adresse de la personne émettrice.

IV-C-2-b - Client

De la même manière que précédemment, nous allons écrire la partie cliente qui ne fera qu'envoyer des données en utilisant **UDP**.

```
DatagramSocket socket = new DatagramSocket();

//on réalise les tests en locaux, on considère l'adresse du serveur comme étant la notre
InetAddress serverAddress = InetAddress.getLocalHost();

//on aurait pu tout de suite utiliser buffer = "Coucou".getBytes()
byte[] buffer = new byte[1500];
DatagramPacket packet = new DatagramPacket(buffer, buffer.length, serverAddress, 300);
packet.setData("Coucou".getBytes());

socket.send(packet);

socket.close();
```

La définition du serveur se fait lors de la création du **DatagramPacket**. Il est obligatoire d'utiliser la méthode **setData** pour forcer les données avec ce que l'on souhaite. Vous remarquerez que la méthode **send** ne prévoit rien pour savoir si l'envoi s'est bien passé. Comme il se doit, nous fermons le socket à la fin de la communication.

IV-C-3 - Ping/Pong complet

En utilisant les méthodes que nous avons vu précédemment, nous pouvons à présent écrire un serveur qui reçoit des données, et renvoie PONG au client.

IV-C-3-a - Serveur

```
DatagramSocket socket = new DatagramSocket(300);

byte[] buffer = new byte[1500];
DatagramPacket packet = new DatagramPacket(buffer, buffer.length);

while(true) {
    socket.receive(packet);

    InetAddress clientAddress = packet.getAddress();
    int port = packet.getPort();

    System.out.println("Reception de : " + clientAddress.getHostName() + " : " + new
String(packet.getData(), 0, packet.getLength()));

    String g = "PONG";
    byte[] b = g.getBytes();
```



```
DatagramPacket clientPacket = new DatagramPacket(b, b.length, clientAddress, port);
socket.send(clientPacket);
}
```

IV-C-3-b - Client

```
DatagramSocket socket = new DatagramSocket();
InetAddress serverAddress = InetAddress.getLocalHost();
int port = 300;

String g = "Coucou";
byte[] bG = g.getBytes();
DatagramPacket packet = new DatagramPacket(bG, bG.length, serverAddress, port);
packet.setData("Coucou".getBytes());

socket.send(packet);

byte[] buffer = new byte[1500];
DatagramPacket pong = new DatagramPacket(buffer, buffer.length, serverAddress, port);

//on met un timeout
socket.setSoTimeout(2000);
socket.receive(pong);
System.out.println("Client a reçu : " + new String(pong.getData(), 0, pong.getLength()));

socket.close();
```

IV-D - Comment réaliser une application réseau monothread

Historiquement, les applications réseaux n'étaient pas multithreadés, et il était donc nécessaire de définir une méthode pour utiliser plusieurs sockets en même temps. Nous avons déjà vu que les méthodes **read** et **accept** étaient bloquantes, ce qui pose un réel problème dans une application monothread, il a donc été utile d'introduire des outils permettant de gérer le réseau de manière non bloquante.

Les API réseau bas-niveau (Unix, Windows) ont définis un ensemble de primitives et d'appels systèmes permettant de manipuler des ensembles de sockets.

Intuitivement, on s'attend à avoir des primitives du genre :

```
socket1 : Socket
socket2 : Socket

Créer un ensemble de Socket socketSet (fd_set en C)
ajouter à l'ensemble socket1 et socket2 (FD_SET)
attendre qu'un événement de produise sur socketSet (appel système select)
Parcourir tous les sockets de socketSet
    si le socket a causé l'événement, des données sont arrivés, on peut lire (FD_ISSET)
    sinon la lecture serait bloquante
Recommencer
```

Java introduit des objets assez similaire aux sockets et aux ServerSocket. Les **SocketChannel** et les **ServerSocketChannel**.

IV-D-1 - Sockets non bloquants

Les **Channel** sont reliés à une socket lors de sa création, ces objets supportent les connections non bloquantes. Dans le code suivant, nous créons un **SocketChannel** connecté à www.developpez.com sur le port 80. l'appel à la méthode **configureBlocking** permet d'indiquer que l'on travaille en non bloquant.

```
SocketAddress address = new InetSocketAddress("www.developpez.com", 80);

SocketChannel socketChannel1 = SocketChannel.open(address);
socketChannel1.configureBlocking(false);

//code

socketChannel1.close();
```

 La classe **InetSocketAddress** permet de représenter une adresse de socket qui correspond au couple : IP + port.

De la même manière, un **ServerSocketChannel** peut être défini comme suit :

```
InetSocketAddress address = new InetSocketAddress(InetAddress.getLocalHost(), 3000);
ServerSocketChannel serverChannel = ServerSocketChannel.open();

serverChannel.configureBlocking(false);

//acquisition de l'adresse socket
serverChannel.socket().bind(address);

//code

serverChannel.close();
```

IV-D-2 - Selector

La classe **SocketChannel** n'est pas utilisable seule, il est nécessaire de la coupler à la classe **Selector**.

La classe **Selector** permet de manipuler des ensembles de **Channel**. Cette classe contient un ensemble de clef qui permet d'identifier tous les **Channel** et dispose d'une méthode permettant d'attendre qu'un événement arrive sur un **Channel** selon le principe vu précédemment.

```
//création d'un selector
Selector selector = Selector.open();
```

Il est ensuite nécessaire d'ajouter le **ServerSocketChannel** au selector. On indique que l'on veut que l'ensemble **selector** se débloque lorsque le serverChannel recoit une demande de connexion (comme avec la méthode **accept()**).

```
serverChannel.register(selector, SelectionKey.OP_ACCEPT);
//ou serverChannel.register(selector, serverChannel.validOps());
```

Ensuite, il est nécessaire de mettre en attente l'ensemble des *channels* et de n'être réveillé que lorsqu'un événement ce produit sur l'un des sockets. On utilise pour cela la méthode **select** qui retourne le nombre de channels où un événement est arrivé.

```
while(selector.select() > 0) {  
    //un événement s'est produit, des données sont arrivés sur un socket  
    // ou sur un server socket  
}
```

La méthode **selectedKeys** permet d'obtenir l'ensemble des clefs **SelectionKey** où un événement s'est produit.

```
while(selector.select() > 0) {  
    Iterator<SelectionKey> keys = selector.selectedKeys().iterator();  
    while(keys.hasNext()) {  
        SelectionKey key = keys.next();  
        keys.remove(); //on supprime la clef  
  
        //l'événement correspond à une acceptation de connexion  
        // on est dans le cas du ServerChannel  
        if(key.isValid() && key.isAcceptable()) {  
  
        }  
    }  
}
```

A présent, nous pouvons accepter la connexion et ajouter le nouveau socket dans le **selector**.

```
while(selector.select() > 0) {  
    Iterator<SelectionKey> keys = selector.selectedKeys().iterator();  
  
    while(keys.hasNext()) {  
        SelectionKey key = keys.next();  
        keys.remove();  
  
        //l'événement correspond à une acceptation de connexion  
        // on est dans le cas du ServerChannel  
        if(key.isValid() && key.isAcceptable()) {  
            ServerSocketChannel server = (ServerSocketChannel) key.channel();  
            SocketChannel client = server.accept();  
            client.configureBlocking(false);  
  
            //on ajoute le client à l'ensemble  
            client.register(selector, SelectionKey.OP_READ | SelectionKey.OP_WRITE);  
  
        }  
  
        //l'événement correspond à une lecture possible  
        //on est donc dans un SocketChannel  
        if(key.isValid() && key.isReadable()) {  
            //byte[] b = new byte[1000];  
            System.out.println("la");  
            SocketChannel client = (SocketChannel) key.channel();  
            try {  
                ByteBuffer buffer = ByteBuffer.allocate(1000);  
  
                int bitsLus = client.read(buffer);  
                if(bitsLus>0)  
                    System.out.println("Recu : " + new String(buffer.array(), 0, bitsLus));  
                else {  
                    //c'est la fin de la connexion, on retire la clef  
                    // du selector  
                    client.close();  
                    System.out.println("Fin de connexion");  
                    key.cancel();  
                }  
            }  
        }  
    }  
}
```

```
    }  
    catch(Exception e) {  
        //une erreur s'est produit, on retire  
        //du selector  
        e.printStackTrace();  
        key.cancel();  
        client.close();  
    }  
}  
  
if(key.isValid() && key.isWritable()) {  
    //écrire éventuellement  
}  
}  
}
```

V - Ressources

Je ne peux que conseiller d'aller voir la javadoc sur le site de *Sun*, notamment en ce qui concerne le package *net* : **cliquez ici**. Depuis que le *JDK* est devenu open source, vous pouvez également aller voir les sources des *API* réseaux qui permettent parfois de mieux comprendre le fonctionnement d'une classe.

VI - Remerciements

Je tiens à remercier ... pour les corrections apportées.

