# Langage Java Chapitre 9 : le réseau, l'essentiel du paquetage java.net

## 1. LES SOCKETS

### 1.1 Notion de Client-Serveur

La communication à travers les réseaux et notamment par Internet est basée sur le modèle Client-Serveur.

Un serveur est un programme qui s'exécute sur une machine connue dont la fonction principale est de fournir un service, par exemple :

- un serveur FTP fournit des fichiers,
- un serveur Web délivre des documents HTML …

La machine qui héberge et fait tourner le logiciel serveur est appelée par extension un serveur. Plusieurs logiciels serveur peuvent s'exécuter sur la même machine serveur, ce serveur peut ainsi fournir les services correspondants.

Un client est un programme qui s'exécute sur une machine dont la fonction est de demander un service à un serveur. Pour ce faire il émet une requête particulière en direction d'un serveur parfaitement identifié. Par extension la machine qui héberge et fait tourner le logiciel client est appelée un client.

### 1.2 Adresse Internet et nom Internet

Dans le réseau Internet un serveur est identifié par son adresse Internet qui est unique ou par son nom Internet également unique. La correspondance entre le nom Internet et l'adresse Internet d'une machine est mémorisée dans la base de données du serveur DNS (Domain Name System) du domaine auquel appartient la machine.

Une adresse Internet est un nombre sur 4 octets (4*8=32 bits), qu'on désigne ici par N1, N2, N3 et N4, ces 4 octets servant à exprimer la valeur numérique de l'adresse.
Une adresse Internet est exprimée en utilisant la notation décimale pointée qui est de la forme :

**N1.N2.N3.N4 avec 0<= Nx <=255**

Par exemple 172.16.32.236.

Le mécanisme qui permet d'obtenir l'adresse Internet d'une machine à partir de son nom est appelé la résolution de noms.
Par exemple, ce mécanisme de résolution de noms est mis en œuvre lorsqu'une personne valide *http://www.altavista.com* dans la fenêtre *Adresse* de son navigateur : le navigateur interroge automatiquement le serveur DNS du domaine auquel il appartient pour lui demander l'adresse Internet du serveur indiqué, ici *www.altavista.com*, le serveur DNS fait ce qu'il faut pour obtenir cette adresse IP puis la retourne au navigateur. Les noms Internet sont réservés

aux humains, les trames qui circulent sur le réseau contiennent les adresses Internet de la machine serveur et de la machine cliente, qu'on appelle d'une manière plus générale adresses destination et source.

### 1.3 Les ports

La machine sur laquelle s'exécute le serveur est maintenant parfaitement repérée à l'aide de son adresse Internet.
Il faut maintenant identifier le service sollicité sur cette machine serveur.
La normalisation Internet attribue de manière unique un numéro sur 16 bits appelé le numéro de port pour chaque service standard.
Ce numéro de port (ou port) est utilisé dans la trame émise par le client à destination de la machine serveur pour identifier le service sollicité.

Les numéros de port sont normalisés jusqu'à la valeur 1024.
En voici quelque uns :
Exemple : le port 80 est affecté au protocole http sur lequel les services du Web ont été développés.

| | |
|---|---|
| echo | 7 |
| discard | 9 |
| daytime | 13 |
| ftp-data | 20 |
| ftp | 21 |
| telnet | 23 |

| | |
|---|---|
| smtp | 25 |
| http | 80 |
| pop3 | 110 |
| snmp | 161 |
| talk | 517 |

Le numéro de port du client n'est jamais fixé à l'avance : lors de la demande de communication avec le serveur, le système octroie au navigateur un numéro de port dont la valeur est supérieure à 1024. On dit que le client dispose d'un numéro de port obtenu de manière dynamique, ce port reste le même tant que le client reste en communication avec un même serveur.

### 1.4 Les sockets

Une machine est identifiée de manière unique sur le réseau à l'aide de son adresse IP, comme il a été vu en 1.2. Chaque service bien connu est identifié par son nom (FTP, Telnet …) et le standard Internet lui associe un numéro unique dit de numéro de port, comme il a été vu en 1.3.

On voit donc qu'un service particulier sur une machine donnée est repéré de manière unique par le couple de données « adresse ip serveur – numéro de port du service».

Le logiciel client s'exécute sur une machine client qui est, elle aussi, identifiée par une adresse IP (fixe ou temporaire). Le logiciel client se voit attribuer de manière dynamique un numéro de port avant de se connecter au serveur. On voit que là aussi un client est identifié de manière unique par le couple « adresse ip client – numéro de port affecté au logiciel client».

Le couple de donnée **{ adresse ip , numéro de port }** est appelé un **socket**.

Par exemple, le socket { 195.168.15.52 , 80 } identifie un serveur Web d'adresse 195.168.15.52.
Une connexion client serveur peut se voir comme une voie de communication entre 2 sockets.

Remarque : certains auteurs ajoutent le protocole dans la définition d'un socket :

**{ adresse ip , numéro de port , protocole }**

## 1.5 Notions sur les protocoles TCP/IP

Le réseau Internet repose sur la famille de protocoles TCP/IP.

Le protocole IP (Internet Protocol) est chargé du routage des données dans le réseau, c'est à dire qu'il doit trouver le chemin qu'un paquet de données émis doit suivre pour arriver à destination.

Le protocole IP fournit donc un service analogue à celui de La Poste :
1) la Poste prend une enveloppe dans une boîte aux lettres,
2) amène cette enveloppe au centre de tri,
3) le centre de tri prend une décision pour la destination suivante : quel est le centre de tri suivant où l'enveloppe doit être amenée ou peut-on l'acheminer directement vers sa destination finale (boîte aux lettres du destinataire) ?
4) l'étape 3 est répétée par le centre de tri suivant jusqu'à ce qu'un centre de tri prenne la décision de délivrer directement l'enveloppe dans la boîte aux lettres du destinataire.

Les fonctionnalités de base du protocole IP sont les suivantes :
1) la machine émettrice envoie le paquet au routeur qu'elle connaît,
2) le routeur prend la décision d'envoyer le paquet à un routeur suivant ou de l'envoyer directement à la machine destinataire,
3) l'étape 2 est répétée par le routeur suivant jusqu'à ce qu'un routeur prenne la décision de délivrer directement le paquet vers la machine destinataire.

Mais la Poste n'informe pas l'expéditeur si :
- la lettre est perdue,
- le destinataire est absent pour une longue durée et que, malheureusement, l'expéditeur attend une réponse.

IP présente les mêmes défauts que la Poste, il ne s'occupe pas :
- de s'assurer que la machine destinataire est bien présente, c'est à dire de créer une connexion entre les machines qui doivent dialoguer,
- d'effectuer un acquittement à chaque paquet reçu : on se sait pas si un paquet est arrivé à destination, si des paquets peuvent être perdus….
- de l'ordre d'arrivée des paquets, les divers paquets peuvent emprunter des chemins différents et un paquet émis plus tard peut arriver avant le paquet précédemment envoyé.

C'est pour cela qu'on dit que le protocole IP assure un service de type **DATAGRAMME**.
Les défauts de IP sont corrigés par le protocole TCP.

Le protocole TCP (Transport Control Protocol) assure les services suivants :
- l'identification des processus qui communiquent : le client et le serveur grâce aux numéros de port,
- l'établissement d'une connexion entre les 2 processus (client et serveur) qui veulent dialoguer, et cela avant tout transfert de données,
- l'acquittement de chaque paquet transmis : le récepteur envoie vers l'émetteur un acquittement quand il a reçu un paquet,
- le découpage de l'information à transmettre en paquets et, lors de la réception, le réassemblage dans l'ordre des paquets reçus,

- la suppression de la connexion quand les 2 processus ne veulent plus dialoguer.

TCP est donc capable d'assurer un transport fiable de l'information, **on dit que TCP assure un transport en MODE CONNECTE**.

TCP ne s'occupe pas de savoir où sont physiquement situés les processus qui dialoguent, c'est le rôle du protocole IP.

Les protocoles TCP sont en réalité au nombre de 2 : TCP lui-même et UDP (User Datagram Protocol).

Le protocole UDP est très simple, son rôle est exclusivement limité à l'identification des processus qui dialoguent grâce aux numéros de port. **On dit que UDP assure un transport en mode datagramme** ou non connecté.

## 2. NOM INTERNET ET ADRESSE IP D'UNE MACHINE : LA CLASSE InetAddress

Les adresses IP sont manipulées en Java à l'aide d'objet InetAddress ; cette classe ne comporte pas de constructeur.

Les applications doivent utiliser les méthodes **getLocalHost()**, **getByName()**, ou **getAllByName()** pour créer une nouvelle instance de la **classe InetAddress**.

- La méthode **getLocalHost()** retourne une instance de la classe **InetAddress** correspondant à la machine locale.
- La méthode **getByName(String host)** retourne une instance de la classe **InetAddress** correspondant au nom de la machine donné en argument. L'argument est donné sous la forme d'un nom Internet comme www.wanadoo.fr ou la forme d'une adresse IP comme 192.168.11.20.
- La méthode **getAllByName(String host)** retourne un tableau d'objets **InetAddress** correspondant à toutes les adresses de la machine. L'argument est donné sous la forme d'un nom Internet comme www.wanadoo.fr ou la forme d'une adresse IP comme 192.168.11.20.

Exemple : affichage de sa propre adresse IP et de son nom

```
import java.net.* ;
import java.io.* ;
public class Inet {
  public static void main( String []  args) {
     InetAddress   myinet ;
     try {
          myinet = InetAddress.getLocalHost() ;
          System.out.println(myinet.getHostName() + " : " + myinet.getHostAddress() );
          } catch (UnknownHostException e)
                { System.out.println(" Erreur : "+e) ;
                }
       }
}
```

## 3. LA CLASSE Socket POUR CRÉER DES CLIENTS TCP

**Exemple** : connexion d'un client au port **echo** (N° 7) de la machine serveur **192.168.0.1**

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

public class ClientEcho {
  public static void main(String [] args ) {
    try {
       InetAddress      distante ;
       Socket           sock ;
       sock = new Socket("192.168.0.1",7) ;
       System.out.println("Connexion réussie") ;
       distante = sock.getInetAddress() ;
       System.out.println("Connecté a "+ distante.getHostName() );   //affiche l'ad IP distante
       System.out.println("Sur le port "+sock.getPort());                // affiche le Port distant
       sock.close() ;
    } catch(UnknownHostException e)
          { System.err.println("Erreur HOST " + e );
          }
    catch (IOException e)
          { System.err.println("Erreur IO "+ e) ;
          }
    }
}
```

La création de l'objet sock instance de la classe Socket

**sock = new Socket("192.168.0.1",7) ;**

entraîne automatiquement la tentative de création d'une connexion TCP/IP avec le socket serveur **{ "192.168.0.1",7 }** donné, ce socket précise de manière unique un service correspondant au N° de port donné situé sur la machine dont l'adresse IP est indiquée.

**Constructeurs**

```
public Socket (String host, int port)  throws UnknownHostException, IOException
public Socket (String host, int port, boolean stream) throws IOException
```

Ces constructeurs permettent de créer une instance de la classe Socket et d'obtenir pour un programme client une connexion avec le programme serveur de la machine host en attente sur le port.
Si **stream** (égal à true par défaut) est égal à false, un socket de type datagramme (utilisation du protocole UDP) est créé.

**Recevoir ou envoyer des données**

Une fois obtenue une instance de la classe Socket, il est possible d'obtenir côté client comme côté serveur un flux de données d'entrée et un flux de données de sortie, grâce aux méthodes **getInputStream ()** et **getOutputStream ()**. Ces méthodes renvoient des instances de classes dérivant des classes **InputStream** (flux de données en lecture) et **OutputStream** (flux de

données en écriture), dont les méthodes permettent de lire et d'envoyer des données entre le client et le serveur.

```
InputStream lireSock ;
lireSock =  sock.getInputStream() ;
```

```
OutputStream        ecrireSock ;
ecrireSock = sock.getOutputStream() ;
```

La méthode **getInputStream()** renvoie un flux d'entrée de données brutes grâce auquel un programme peut récupérer les données reçues sur le socket.
Il est d'usage de lier cet **InputStream** à un autre flux offrant davantage de fonctionnalités, par exemple DataInputStream ou InputStreamReader.

La méthode **getOutputStream()** renvoie un flux de sortie de données brutes grâce auquel un programme peut émettre des données sur le socket.
Il est d'usage de lier cet **OutputStream** à un autre flux offrant davantage de fonctionnalités, par exemple DataOutputStream ou OutputStreamWriter.

```
DataInputStream  lireSock ;
lireSock =  new DataInputStream(sock.getInputStream()) ;
```

```
DataOutputStream ecrireSock ;
ecrireSock = new DataOutputStream(sock.getOutputStream() );
```

Voir l'exercice 5.3 et la remarque faite.

**Fermer la connexion**

```
public synchronized void close () throws IOException
```

**Obtenir des informations sur le socket distant**

```
public InetAddress getInetAddress ()
public int getPort ()
```

Ces méthodes renvoient un objet InetAddres (pour l'adresse Internet distante) et le port du socket distant auquel le socket local est connecté.

**Obtenir son N° de port**

```
public int getLocalPort ()
```

Renvoie le port local sur lequel le socket est connecté.

**Exercices : 5.1, 5.2, 5.3, 5.4 et 5.5.**

## 4. LA CLASSE ServerSocket POUR CRÉER DES SERVEURS TCP

Exemple : serveur simple de type echo travaillant sur le port N° 5000, il attend la connexion d'un client, il récupère les données reçues, il les retourne au client puis il se termine

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

public class ServeurEcho1 {
 public static void main(String[] args) {
        ServerSocket serveur ;
        Socket          client ;
        try {
                serveur = new ServerSocket(5000) ;
                client = serveur.accept() ;
                InetAddress  inet = client.getInetAddress() ;
                System.out.println("Ip du client = "+inet.getHostAddress()) ;
                System.out.println("Port du client : "+client.getPort());
                InputStream lireSock = client.getInputStream() ;
                OutputStream ecrireSock = client.getOutputStream() ;
                byte [] Tampon = new byte[100] ;
                int n = 0 ;

                n=lireSock.read(Tampon) ;
                System.out.println("Nombre de caracteres lus = "+n) ;

                String recu  = new String(Tampon, 0, n) ;
                System.out.println("Recu = "+recu) ;
                ecrireSock.write(Tampon);

                serveur.close();
           } catch (IOException e)
                { System.err.println("IO : "+e);
                }
        }
}
```

Le constructeur **ServerSocket(int port)** reçoit comme argument un N° de port. Une instance de la classe ServerSocket est créée, ce socket a comme paramètre le N° de port donné et comme adresse IP l'adresse IP locale.

La méthode **accept()** place le serveur en attente de la demande de connexion d'un client. Elle retourne une instance de la classe Socket pour identifier la connexion réalisée.

**Constructeurs**

```
public ServerSocket (int port) throws IOException
public ServerSocket (int port, int count) throws IOException
```

Ces constructeurs permettent de créer un socket de serveur en attente sur le **`port`** donné.

Le 2ème argument **`count`** (égal à `50` par défaut) permet de spécifier le nombre maximum de demande de connexions en attente que le serveur acceptera.

**Méthodes**

Attendre un client

```
public Socket accept () throws IOException
```

Attend la demande de connexion d'un programme client, et renvoie un **socket** une fois la connexion établie.

Fermer la connexion

```
public void close () throws IOException
```

Ferme le socket du serveur.

Les méthodes de la classe Socket vues au chapitre 3 sont toutes valables, on peut néanmoins détailler les lignes suivantes issues du programme du serveur écho :

```
Socket client = serveur.acept() ;
// Pour récupérer l'adresse du client.
InetAddress  inet = client.getInetAddress () ;
// Pour récupérer la valeur du port.
int port = client.getLocalPort ()
```

## 5. EXERCICES SUR LES SOCKETS TCP

**`5.1`** Tester l'exemple donné au chapitre 2.

**`5.2`** Tester l'exemple donné au chapitre 3.

**`5.3`** Ajouter à l'exemple précédent l'affichage des paramètres du socket local, l'envoi de données au serveur echo (chaînes de caractères), la réception des données renvoyées par le serveur puis leur affichage. Les données envoyées au serveur seront terminées par un **\n**, afin d'en faciliter la lecture au retour par la méthode **DataInputStream.readLine()** (voir remarque). Ces données seront émises en passant par la méthode **DataOutputStream.writeBytes(String)**.

Remarque : La méthode **DataInputStream.readLine()** est actuellement dépréciée, la documentation Oracle conseille de remplacer l'utilisation d'un objet **DataInputStream** par l'objet **BufferedReader** .
On peut écrire :

```
BufferedReader    lireSock ;
lireSock = new BufferedReader(new InputStreamReader(sock.getInputStream())) ;
```

et utiliser la méthode **readLine()** de BufferedReader.

**`5.4`** Proposer une solution qui utilise **finally** afin de s'assurer de la fermeture du socket en cas de déclenchement d'exception à la suite d'un dysfonctionnement.

**5.5** Proposer un programmer qui permet de « scanner » les ports d'une machine et qui affiche le résultat. On peut par exemple tester les ports du N° 1 au numéro 9999.

**5.6** Création d'un client Web

Le serveur **d'adresse 192.168.12.51** héberge un serveur HTTP (serveur Web) .

Le but est de recevoir la page d'accueil du serveur. Dans le cas d'un serveur Apache sous Linux, le fichier **index.shtml** situé dans le répertoire /var/www/html du serveur représente la page d'accueil. La requête suivante **http://iris.agora.fr** ou **http://192.168.12.51** qui ne demande pas de document particulier entraîne automatiquement par défaut l'envoi du document **index.shtml** au client.

On demande de créer un client du service **HTTP** du serveur. Le serveur **HTTP** est en attente sur le N° de port standard **80** (80 en décimal). Le protocole HTTP repose sur TCP/IP.
La structure de base du programme est la même que celle du client *echo* mis à part les points suivants :

- La connexion établie, le client doit envoyer au serveur la chaîne :
  **"GET / HTTP/1.0\n\n"**
  (GET suivi d'un espace, d'un / ,d'un espace, puis du texte indiqué, les 2 \n sont impératifs).**GET** est une commande HTTP qui demande un document au serveur. Comme aucun nom de document n'est ici indiqué, le serveur envoie alors le document par défaut.
- En réponse le serveur envoie le fichier **index.shtml** situé dans son répertoire **/var/www/html**.

Attention : **Le serveur peut effectuer plusieurs envois successifs à destination du client si la taille du document demandé est importante (**chaque envoi d'une portion du fichier correspond à un « write » du serveur), le client se doit de gérer cette situation.

- Le serveur met fin à la connexion dès qu'il a terminé son envoi.

Attention : **Ce n'est pas le client qui initie la fin de la connexion comme pour un client echo**. Le client doit donc détecter la fermeture de la connexion par le serveur puis fermer à son tour.

**5.6.1** Ecrire le programme du client, on demande d'afficher à l'écran le texte du fichier html reçu.

**5.6.2** Le serveur Web dispose de quelques documents situés dans le répertoire **/var/www/html/manual**, par exemple **index.html**, **env.html** et **sections.html**.

Un document quelconque dans ce répertoire, par exemple **bind.html**, est demandé au serveur Web par l'envoi de la chaîne :
**"GET /manual/bind.html HTTP/1.0\n\n"**

Modifier le programme du 5.4.1 afin de saisir l'adresse IP de la machine serveur et le nom « relatif » du document demandé ( par exemple : **/manual/bind.html** ).
On demande d'afficher à l'écran le texte du fichier html reçu.

**5.7** Tester l'exemple donné au chapitre 4.

**5.8** Créer un serveur echo capable de traiter plusieurs clients. Un thread sera démarré pour chaque nouvelle connexion acceptée pour traiter un nouveau client. Le thread lira les données reçues du client puis les lui retournera.

**5.9** Ecrire les programmes d'un serveur de « chat » et de son client.

# 6. LES SOCKETS EN MODE DATAGRAMME (protocole UDP)

Les communications réseau qui reposent sur le protocole UDP n'offrent pas le même niveau de fiabilité que le protocole TCP, voir le paragraphe 1.5.
Il faut passer par l'étape de la construction d'un objet datagramme pour envoyer des données comme pour en recevoir.

## 6.1 LA CLASSE DatagramPacket

Cette classe dispose de 2 constructeurs, le premier constructeur est utilisé pour créer un datagramme pour recevoir des données, le second pour créer un datagramme à envoyer à une machine.

Constructeurs

public **DatagramPacket** (byte tampon [ ], int taille)
Construit une instance de la classe `DatagramPacket`, utilisé pour recevoir un paquet de données de longueur **`taille`** qui sera stocké dans le tableau **`tampon`**. Si le datagramme reçu est plus long que `taille`, les données restantes ne sont pas recopiées dans `le tampon`.

public **DatagramPacket** (byte tampon [ ], int taille, InetAddress addr, int port)
Construit une instance de la classe `DatagramPacket` utilisé pour envoyer un paquet de données `stocké dans` **`tampon`** de longueur **`taille`**, au **port** `indiqué` d'une machine d'adresse Internet **`addr`**.

Méthodes : Elles servent essentiellement à obtenir des informations dans le cas de la réception des datagrammes.

public InetAddress **getAddress** ()
Renvoie l'adresse Internet de la machine dont provient ou auquel est destiné ce datagramme selon qu'il s'agit d'un datagramme reçu ou envoyé.

public int **getPort** ()
Renvoie le port de la machine dont provient ou auquel est destiné ce datagramme selon qu'il s'agit d'un datagramme reçu ou envoyé.

public byte[] **getData** ()
public int **getLength** ()
Ces méthodes renvoient le tableau où sont stockées les données d'un datagramme et la longueur des données à recevoir (ou reçues) ou à envoyer.

Exemple : futur client du serveur echo (7) mode UDP

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

public class InfoDatagramme1 {
   public static void main(String[] args) {
        String  s = "Test sur les datagrammes" ;
        byte []   tampon = s.getBytes("ASCII") ;   //conversion en chaîne ascii
        try {
        InetAddress  addr =  InetAddress.getByName("serveurIris.agora.fr") ;
        int  port = 7  ;
        DatagramPacket dp ;
        dp = new DatagramPacket ( tampon , tampon.length , addr , port );

        System.out.println("Paquet qui sera adressé à "+dp.getAddress() +
                                                " sur le port "+dp.getPort() ) ;
                System.out.println("Il comporte les données "+ dp.getData() ) ;
                System.out.println("Nombre de caractères  = "+ dp.getLength() ) ;
           } catch (UnknownHostException e)
                { System.err.println("Host ? "+e);
                }
        }
}
```

## 6.2 LA CLASSE DatagramSocket

Cette classe permet de recevoir et envoyer des datagrammes, grâce aux méthodes `receive ()` et `send ()`.

Constructeurs

public **DatagramSocket** () throws SocketException

Doit être utilisé pour un client, construit une instance de la classe `DatagramSocket`. Le port utilisé pour recevoir ou envoyer des datagrammes est le premier disponible sur la machine locale (obtention dynamique du port).

public **DatagramSocket** (int port) throws SocketException

Doit être utilisé pour un serveur, construit une instance de la classe `DatagramSocket`. Le port `indiqué` est utilisé pour recevoir ou envoyer des datagrammes sur la machine locale.

Méthodes

int **getLocalPort** ()

Renvoie le port de la machine locale sur lequel l'envoi ou la réception de datagrammes s'effectue.

void **receive** (DatagramPacket packet) throws IOException

Permet de recevoir un paquet de données dans le datagramme `packet`. Cette méthode met en attente le thread courant jusqu'à réception d'un datagramme.

void **send** (DatagramPacket packet) throws IOException
Permet d'envoyer le datagramme `packet`.

void **close** () Ferme le socket.

## 7. EXERCICES SUR LES SOCKETS UDP

**7.1** Tester l'exemple donné en 6.1 en utilisant un nom de serveur existant.

**7.2** Ecrire un programme qui teste tous les ports UDP du n°1 au 65535 d'une machine et qui affiche tous les ports utilisés.

**7.3** Ecrire le programme client du serveur echo UDP.

**7.4** Ecrire un programme de serveur echo UDP. Tester ce programme avec le client précédent.