

<p style="text-align: center;"><b>Programmation Réseau en Java</b> <i>Support Numéro 1</i> <i>Sockets</i></p>
---

## 1. Introduction

Le langage Java a été conçu par SUN, une société dont l'activité est fortement tournée vers le réseau et qui a mis au point plusieurs protocoles réseaux utilisés sur Internet (RPC, NFS, NIS). Il peut donc paraître logique que ce langage soit tourné vers le réseau et notamment vers Internet.

Courte bibliographie :

Ces TPs sont basés sur les sources suivantes :

- Le livre "Programmation réseau avec Java" d'Elliotte Rusty Harold paru chez O'Reilly.
- Le cours de Java accessible à l'URL suivante : <http://www.eteks.com/coursjava/>
- Des articles parus dans les revues : "Linux France Magazine" et "Login:"

## 2. Historique

UNIX est un système multi-tâches et multi-utilisateurs. Les processus peuvent coopérer entre eux. Différents outils de communication inter-processus ont été développés: les IPC (Inter Process Communication).

La communication interprocess peut se faire à travers les signaux et les pipes. Mais tandis que la version d'ATT était orientée gestion de ressources (sémaphores, messages, mémoire partagée), l'université de Berkeley a développé une version orientée réseaux en implémentant les protocoles Internet ARPA/DOD et en offrant l'interface des "Sockets".

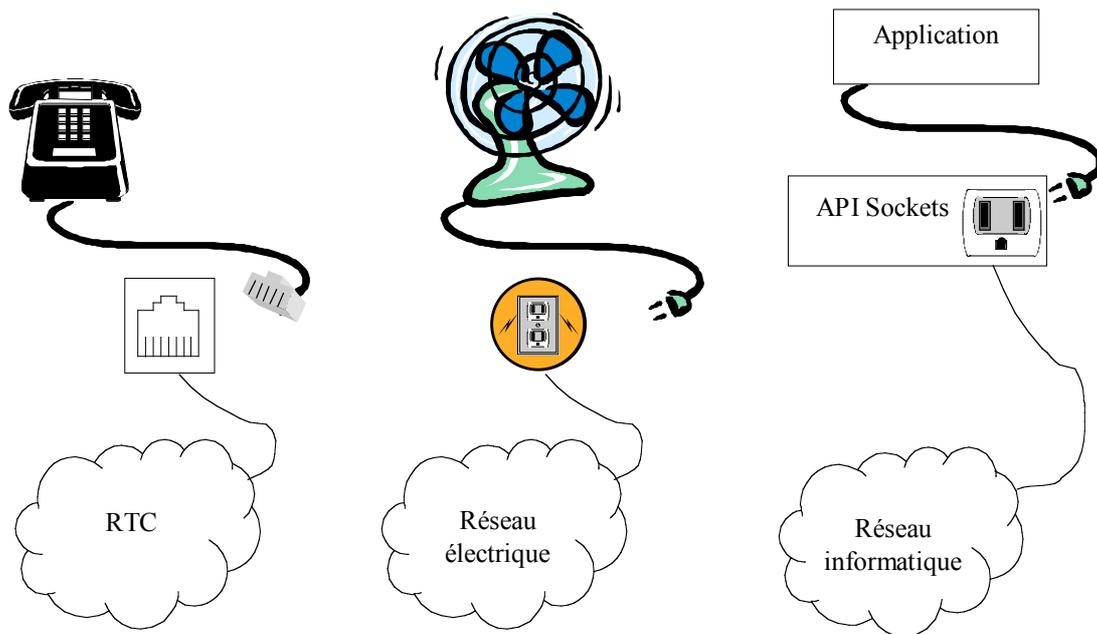
## 3. Concept de socket

Les Sockets forment une API (Application Program Interface): ils offrent aux programmeurs une interface entre le programme d'application et les protocoles de communication. En aucun cas, les sockets ne forment une norme de communication ou une couche de protocole à l'instar de TCP/IP.

Les sockets (prises de raccordement) forment un mécanisme de communication bidirectionnel interprocessus dans un environnement distribué ce qui n'est pas le cas des autres outils tels que les pipes. Ils permettent évidemment la communication interprocess à l'intérieur d'un même système.

L'interface des "sockets" n'est pas liée à une pile de protocoles spécifique. Dans ce cours, nous nous intéresserons, à l'utilisation des sockets dans le monde TCP/IP.

La notion de socket en tant que prise de raccordement vient d'une analogie avec le réseau électrique et le réseau téléphonique :



**Figure 1 : La métaphore des prises**

Les sockets représentent donc d'une part une API c'est à dire un ensemble de primitives de programmation et d'autre part les extrémités de la communication (notion de prise). Les extrémités de communication sont identifiées dans le monde TCP/IP par trois informations : une adresse IP, le protocole utilisé (TCP ou UDP) et un numéro de port (entier sur 16 bits donc de 0 à 65535). Etant donné que ces informations sont uniques dans l'Internet (les adresses IP sont uniques et les numéros de ports sont uniques pour un protocole donné) ces trois informations permettent d'identifier de façon unique une extrémité de communication (à l'instar d'un numéro de téléphone dans un réseau téléphonique).

Pour maintenir l'unicité des numéros de port (par protocole) on les a répartis en trois catégories :

- Les ports systèmes (appelés aussi ports bien connus – well known ports) de 0 à 1023 sont réservés sous UNIX au processus démarrés automatiquement par le système d'exploitation et peuvent être déposés auprès de l'organisme IANA (Internet Assigned Numbers Authority).
- Les ports utilisateurs (appelés aussi ports déposés – registered ports) de 1024 à 49151 sont disponibles pour les utilisateurs et peuvent eux aussi être déposés auprès de l'organisme IANA.
- Les ports dynamiques (appelés aussi ports privés) de 49152 à 65535.

Le site web de IANA permet de connaître la liste des ports systèmes et utilisateurs assignés à un protocole applicatif ou une application donnée (<http://www.iana.org/assignments/port-numbers>). Il propose de plus deux formulaires permettant de déposer des numéros de ports systèmes ou utilisateurs (<http://www.iana.org/cgi-bin/sys-port-number.pl> et <http://www.iana.org/cgi-bin/usr-port-number.pl>).

Pour les TPs il est recommandé d'utiliser les numéros de ports dynamiques.

En fonction du protocole transport utilisé les sockets vont fonctionner différemment. Nous présenterons dans un premier temps les sockets utilisant le protocole TCP qui fonctionnent en utilisant le modèle client/serveur et offrent une communication par flux. Les sockets utilisant le protocole UDP seront ensuite présentés. Ils ont un fonctionnement beaucoup plus proche de ce qui se passe au niveau de la couche réseau (échange de paquets – appelés datagrammes dans le monde TCP/IP – acheminés indépendamment). Les sockets UDP peuvent de plus être utilisés pour une communication entre une application et un groupe de machine (diffusion plus ou moins restreinte – broadcast ou multicast).

## 4. Adresses IP

Une machine (appelée aussi hôte ou host) est identifiée dans l'Internet par son adresse. L'adresse IP d'une machine correspond à un numéro qui est unique dans le monde.

Pour des raisons mnémoniques, il est possible de donner un nom à une machine (ex : Toto, Garonne, Mimosa...). Certains hôtes ont plusieurs noms. Ce nom est géré de façon hiérarchique par le DNS (Système de Noms de Domaines).

L'adresse utilisée par la version actuelle du protocole IP (adresse IP), comporte deux champs: le champ adresse réseau (Network) dans l'Internet et le champ adresse hôte (Host) dans le réseau. Sa taille est de 4 octets (32 bits). Elle est souvent donnée en notation décimale pointée (ex: 127.95.35.54).

Comme l'adresse IP contient l'adresse réseau, une station changeant de réseau change d'adresse. D'autre part, une station multidomiciliée (qui dispose de plusieurs interfaces réseau) ou un routeur ont plusieurs adresses.

Il existe actuellement cinq classes d'adresses IP. Les trois premières permettent de gérer des réseaux de tailles diverses. La classe D permet de gérer une communication multipoint (un message est envoyé à plusieurs machines à la fois). La classe E est réservée et ne sera probablement jamais utilisée puisqu'on devrait bientôt migrer vers la nouvelle version d'IP IPv6 qui stockera les adresses IP dans 16 octets.

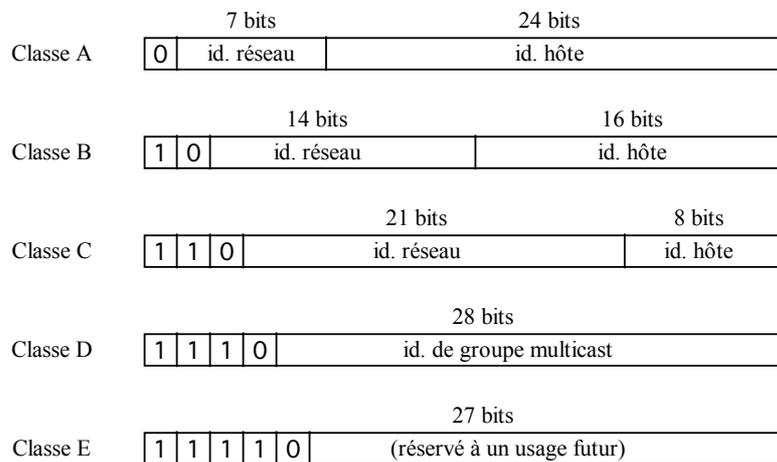


Figure 2 : Les classes d'adresses IP

### 4.1. La classe `InetAddress`

La classe `java.net.InetAddress` permet de représenter les adresses IP. Chaque objet de cette classe possède deux champs `hostName` et `address` contenant respectivement une chaîne de caractère et un tableau d'octets.

Le champ `hostName` stocke le plus souvent le nom de l'hôte ([www.irit.fr](http://www.irit.fr) par exemple) et le champ `address` l'adresse IP.

Cette classe ne possède pas de constructeur publics. Pour créer un objet de type `InetAddress` il faut donc utiliser l'une des méthodes suivantes :

```
public static InetAddress getByName (String nom_hote)
public static InetAddress [ ] getAllByName (String nom_hote)
public static InetAddress getLocalHost ()
```

#### 4.1.1. `public static InetAddress getByName (String nom_hote)`

Cette méthode utilise le DNS pour renvoyer une instance de la classe `InetAddress` représentant l'adresse Internet de la machine de nom `nom_hote`. Voici un exemple d'utilisation (en supposant que la classe `java.net.InetAddress` a été précédemment importée) :

```
InetAddress adr = InetAddress.getByName("www.irit.fr");
```

Lorsque la recherche DNS n'aboutit pas, une exception `UnknownHostException` est levée.

**Exercice** : Sachant que `InetAddress` (comme la plupart des classes Java) surcharge la méthode `toString()`, écrivez un programme créant un objet `InetAddress` et l'affichant.

Testez le programme avec divers noms de machines.

Donnez une adresse IP en lieu et place du nom de machine. Que se passe-t-il ?

### 4.1.2. `public static InetAddress [ ] getAllByName (String nom_hote)`

Cette méthode renvoie toutes les adresses Internet de la machine de nom `nom_hote`.

**Exercice** : modifiez le programme précédent pour récupérer toutes les adresses d'une machine donnée. Essayez de trouver une machine disposant de plusieurs adresses IP.

### 4.1.3. `public static InetAddress getLocalHost ()`

Renvoie une instance de la classe `InetAddress` représentant l'adresse Internet de la machine locale. Très pratique pour tester sur une même machine les programmes client et serveur. Equivalent à `getByName (null)` ou `getByName ("localhost")`.

**Exercice** : modifiez le programme précédent pour afficher en plus l'adresse IP de la machine courante.

### 4.1.4. Autres méthodes

#### 4.1.4.1. `public String getHostName ()`

Renvoie le nom de la machine hôte, ou bien l'adresse IP si la machine n'a pas de nom.

#### 4.1.4.2. `public byte [] getAddress ()`

Renvoie l'adresse IP stockée par une instance de la classe `InetAddress` sous la forme d'un tableau d'octets rangés dans l'ordre standard du réseau (network byte order). Ainsi l'octet d'indice 0 contient l'octet de poids fort de l'adresse.

La longueur du tableau retourné est actuellement 4 dans la plupart des cas (IPv4) mais devrait passer à 16 lorsque les adresses IP sur 128 bits auront cours (IPv6).

**Exercice** : tentez d'afficher à l'aide de votre programme les adresses IP renvoyées par cette méthode. Que remarquez vous ?

**Remarque** : la manipulation d'octets non signés pose des problèmes en Java car il n'y a pas d'équivalent au type C `unsigned char`. Ainsi, les octets supérieurs à 127 sont traités comme des nombres négatifs.

Pour récupérer les bonnes valeurs il convient de procéder comme suit :  
`int octetNonSigne = octet < 0 ? octet + 256 : octet;`

**Exercice** : Corrigez votre programme pour afficher correctement les adresses IP renvoyées par `getAddress()`.

#### 4.1.4.3. Méthodes surchargées

**public int hashCode ()**

**public boolean equals (Object obj)**

**public String toString ()**

Ces méthodes surchargent celles de la classe `Object`, pour renvoyer un code de hashing, comparer un objet de classe `InetAddress` à un objet ou renvoyer une chaîne de caractères décrivant une adresse Internet.

**Remarque importante** : la première méthode est directement utilisée par la classe `InetAddress` qui contient une `HashTable` en membre statique. Ainsi, une adresse requise plusieurs fois sera immédiatement extraite du cache et donc obtenue beaucoup plus rapidement que par l'envoi d'une requête au serveur DNS.

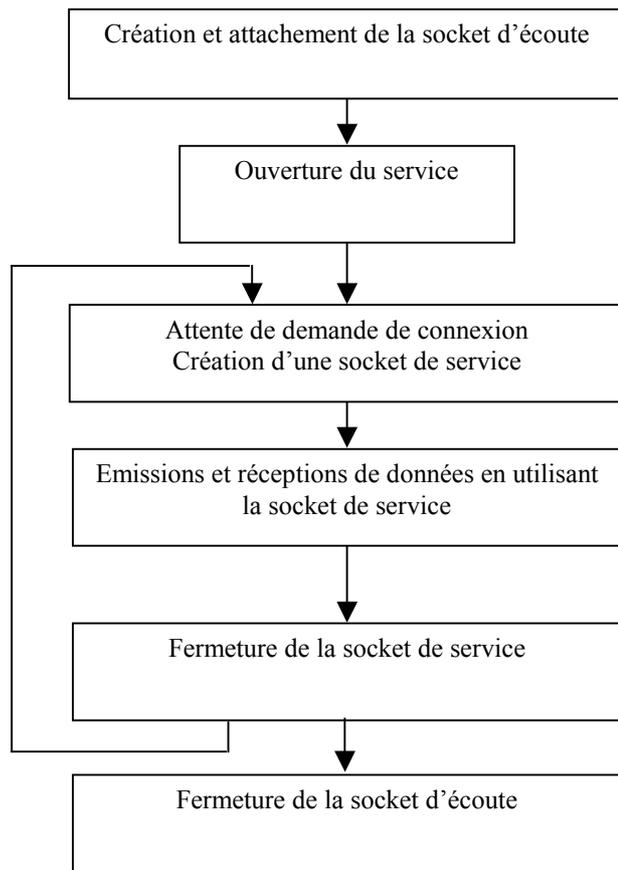
## 5. Sockets TCP

### 5.1. Le modèle client/serveur

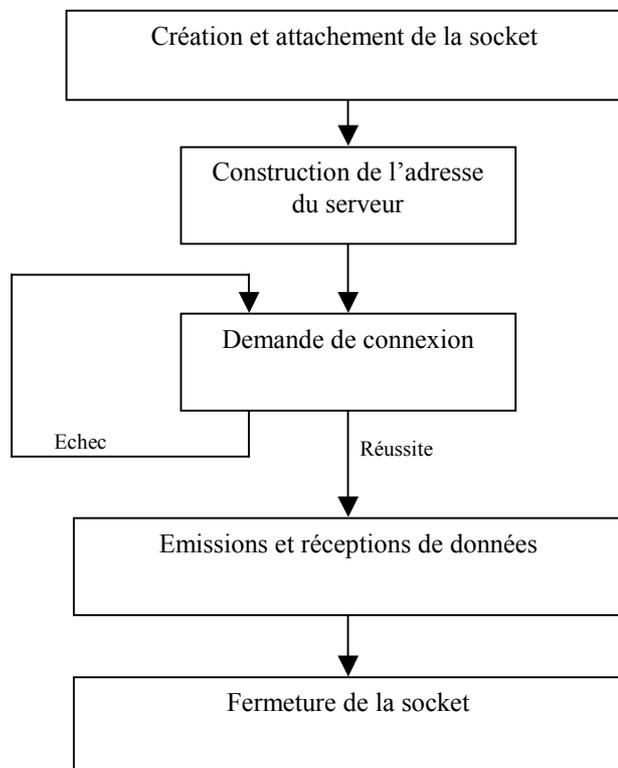
Le protocole TCP offre un service en mode connecté et fiable. Les données sont délivrées dans l'ordre de leur émission.

La procédure d'établissement de connexion est dissymétrique. Un processus, appelé serveur, attend des demandes de connexion qu'un processus, appelé client, lui envoie. Une fois l'étape d'établissement de connexion effectuée le fonctionnement redevient symétrique.

Les deux schémas suivants présentent les algorithmes de fonctionnement des clients et serveurs. Il est à noter que côté serveur on utilise deux sockets : l'un, appelé socket d'écoute, reçoit les demandes de connexion et l'autre, appelé socket de service, sert pour la communication. En effet, un serveur peut être connecté simultanément avec plusieurs clients et dans ce cas on utilisera autant de sockets de service que de clients.



**Fonctionnement du serveur**



**Fonctionnement du client**

## 5.2. La classe *Socket*

La classe *Socket* représente en Java les sockets utilisés côté client ou les sockets de service.

### 5.2.1. Constructeurs

#### **public Socket (String hote, int port) throws UnknownHostException, IOException**

Ce constructeur crée un socket TCP et tente de se connecter sur le port indiqué de l'hôte visé. Le premier paramètre de ce constructeur représente le nom de la machine serveur. Si l'hôte est inconnu ou que le serveur de noms de domaine est inopérant, le constructeur générera une *UnknownHostException*. Les autres causes d'échec, qui déclenchent l'envoi d'une *IOException* sont multiples : machine cible refusant la connexion sur le port précisé ou sur tous les ports, problème lié à la connexion Internet, erreur de routage des paquets...

Voici un exemple d'utilisation de ce constructeur :

```
Socket leSocket = new Socket("www.irit.fr", 80);
```

#### **public Socket (InetAddress adresse, int port) throws IOException**

Ce constructeur fonctionne comme le premier, mais prends comme premier paramètre une instance de la classe *InetAddress*. Ce constructeur provoque une *IOException* lorsque la tentative de connexion échoue mais il ne renvoie pas d'*UnknownHostException* puisque cette information est connue lors de la création de l'objet *InetAddress*.

#### **public Socket(String hote, int port, InetAddress adresseLocale, int portLocal) throws IOException**

Comme les deux précédents, ce constructeur crée un socket et tente de se connecter sur le port indiqué de l'hôte visé. Le numéro du port et le nom de la machine sont fournis dans les deux premiers paramètres et la connexion s'établit à partir de l'interface réseau physique (choix d'une carte réseau sur un système possédant plusieurs accès réseau) ou virtuelle (système multi-adresse) et du port local indiqués dans les deux derniers paramètres. Si **portLocal** vaut 0, la méthode (comme les deux premières d'ailleurs) choisit un port disponible (parfois appelé port anonyme) compris entre 1024 et 65 535.

Comme la première méthode cette méthode peut générer une *IOException* en cas de problème de connexion et lorsque le nom d'hôte donné n'est pas correct (il s'agit dans ce cas d'une *UnknownHostException*).

#### **public Socket(InetAddress address, int port, InetAddress adresseLocale, int portLocal) throws IOException**

Cette méthode est identique à la précédente à la seule différence que le premier paramètre est, comme pour la seconde méthode, un objet *InetAddress*.

#### **protected Socket()**

#### **protected Socket(SocketImpl impl) throws SocketException**

Ces deux derniers constructeurs sont protégés. Ils créent un socket sans le connecter. On utilise ces constructeurs pour implanter un socket original qui permettra par exemple de chiffrer les transactions ou d'interagir avec un Proxy.

Il existe de plus deux autres constructeurs qui permettent le choix du protocole via un booléen mais ces méthodes sont dépréciées et ne doivent pas être utilisées. Pour créer un socket utilisant UDP on utilisera *DatagramSocket*.

### 5.2.2. Méthodes informatives

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

Ces méthodes renvoient l'adresse Internet et le port distants auquel le socket est connecté.

```
public InetAddress getLocalAddress ()  
public int getLocalPort ()
```

Ces méthodes renvoient l'adresse Internet et le port locaux que le socket utilise.

### 5.2.3. Communication avec un socket

```
public InputStream getInputStream () throws IOException
```

Cette méthode renvoie un flux d'entrées brutes grâce auquel un programme peut lire des informations à partir d'un socket. Il est d'usage de lier cet *InputStream* à un autre flux offrant d'avantage de fonctionnalités (un *DataInputStream* par exemple) avant d'acquérir les entrées.

Voici un exemple d'utilisation de cette méthode :

```
DataInputStream fluxEnEntree = new DataInputStream(leSocket.getInputStream());
```

```
public OutputStream getOutputStream () throws IOException
```

Cette méthode renvoie un flux de sortie brutes grâce auquel un programme peut écrire des informations sur un socket. Il est d'usage de lier cet *OutputStream* à un autre flux offrant d'avantage de fonctionnalités (un *DataOutputStream* par exemple) avant d'émettre des données.

Voici un exemple d'utilisation de cette méthode :

```
DataOutputStream fluxEnSortie = new DataOutputStream(leSocket.getOutputStream());
```

### 5.2.4. Fermeture d'un socket

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

Bien que Java ferme tous les sockets ouverts lorsqu'un programme se termine ou bien lors d'un « garbage collect », il est fortement conseillé de fermer explicitement les sockets dont on n'a plus besoin à l'aide de la méthode **close**.

Une fois un socket fermé on peut toujours utiliser les méthodes informatives, par contre toute tentative de lecture ou écriture sur les « input/output streams » provoque une *IOException*.

### 5.2.5. Options des sockets

Java permet l'accès à un certain nombre d'options qui modifient le comportement par défaut des sockets. Ces options correspondent à celles que l'on manipule en C via `ioctl` (ou `ioctlsocket` avec Windows).

**public boolean getTcpNoDelay() throws SocketException**  
**public void setTcpNoDelay(boolean valide) throws SocketException**

Valide/invalide l'option `TCP_NODELAY`. Valider `TCP_NODELAY` garantit que les données seront expédiées aussi rapidement que possible quelle que soit leur taille. Ceci correspond à interdire la mise en œuvre de l'**algorithme de Nagle** qui, en simplifiant, concatène les données de taille réduite dans des segments de taille supérieure avant envoi.

Si l'on programme une application très interactive on pourra valider l'option `TCP_NODELAY`.

**public int getSoLinger() throws SocketException**  
**public void setSoLinger(boolean valide, int secondes) throws SocketException**

L'option `SO_LINGER` indique le comportement à adopter lorsqu'il reste des données à expédier au moment de la fermeture du socket. Par défaut, la méthode `close()` rend la main immédiatement et le système tente d'envoyer le reliquat de données. Si la durée des prolongations **secondes** vaut 0 et que **valide** est à vrai, les éventuelles données restant sont supprimés lorsque la fermeture à lieu. Si la durée est positive et que **valide** est à vrai, la méthode `close` se fige pendant le nombre de secondes spécifiées en attendant l'expédition des données et la réception de l'acquiescement. Une fois les prolongations écoulées, le socket est clos et les données restantes ne sont pas transmises.

Si cette option n'est pas validée (**valide** est à faux) on obtient le comportement par défaut et l'appel de la méthode `getSoLinger()` retourne -1, sinon la méthode renvoie la durée des prolongations.

**public int getSoTimeout() throws SocketException**  
**public void setSoTimeout(int ms) throws SocketException**

Par défaut, lors d'une tentative de lecture sur le flux d'entrée d'un socket, `read()` se bloque jusqu'à la réception d'un nombre d'octets suffisant. Lorsque `SO_TIMEOUT` est initialisée, cette attente ne dépasse pas le temps imparti, exprimé en millisecondes. Tout dépassement de temps se solde par une **java.net.SocketTimeoutException**. Le socket reste malgré tout connecté.

La valeur par défaut est 0 qui signifie, ici, un laps de temps infini. Il peut être préférable de choisir une durée raisonnable (>0) pour éviter que le programme se bloque en cas de problème.

L'option doit être validée avant l'appel à l'opération bloquante pour être prise en compte.

Le paramètre doit être  $\geq 0$  sans quoi une exception *SocketException* est générée.

**public int getSendBufferSize() throws SocketException**  
**public void setSendBufferSize(int size) throws SocketException**  
**public int getReceiveBufferSize() throws SocketException**  
**public void setReceiveBufferSize(int size) throws SocketException**

Ces méthodes permettent, grâce aux options `SO_SNDBUF` et `SO_RCVBUF`, de préciser une indication sur la taille à allouer aux tampons d'entrée/sortie bas niveaux. Il s'agit uniquement d'une indication donc il est conseillé après l'appel d'une méthode `set...` d'appeler la méthode `get...` correspondante pour vérifier la taille allouée.

### 5.2.6. Méthode surchargée

**public String toString()**

La seule méthode surchargée de la classe *Object* est **toString()**. Cette méthode provoque l’affichage d’une chaîne ressemblant à cela :

```
Socket[addr=amber/192.11.4.2,port=80,localport=1167]
```

Cette méthode est donc plutôt destinée au débogage.

### 5.3. La classe **ServerSocket**

Cette classe permet de créer des sockets qui attendent des connexions sur un port spécifié et lors d’une connexion retournent un *Socket* qui permet de communiquer avec l’appelant.

#### 5.3.1. Constructeurs

##### **public ServerSocket (int port) throws IOException**

Ce constructeur crée un socket serveur qui attendra les connexions sur le **port** spécifié. Lorsque l’entier **port** vaut 0, le port est sélectionné par le système. Ces ports anonymes sont peu utilisés car le client doit connaître à l’avance le numéro du port de destination. Il faut donc un mécanisme, comme le portmapper des RPC, qui permet d’obtenir ce numéro de port à partir d’une autre information.

L’échec de la création se solde par une *IOException* (ou à partir de Java 1.1 une *BindException* qui hérite de *IOException*) qui traduit soit l’indisponibilité du port choisi soit, sous UNIX, un problème de droits (sous UNIX les ports inférieurs à 1024 ne sont disponibles que pour le super utilisateur).

##### **public ServerSocket (int port, int tailleFile) throws IOException**

Ce constructeur permet en outre de préciser la longueur de la file d’attente des requêtes de connexion. Si une demande de connexion arrive et que la file est pleine la connexion sera refusée sinon elle sera stockée dans la file pour être traitée ultérieurement. La longueur de la file d’attente par défaut (pour le premier constructeur) est 50.

Lorsque la valeur donnée est supérieure à la limite fixée par le système, la taille maximale est affectée à la file.

##### **public ServerSocket (int port, int tailleFile, InetAddress adresseLocale) throws IOException**

Ce constructeur permet en outre de préciser l’adresse Internet locale sur laquelle attendre des connexions. Ainsi on pourra choisir l’une des interfaces réseau de la machine locale si elle en possède plusieurs. Si **adresseLocale** est à **null** le socket attendra des connexions sur toutes les adresses locales (ce qui est aussi le cas quand on utilise l’un des deux autres constructeurs).

#### 5.3.2. Accepter et clore une connexion

##### **public Socket accept () throws IOException**

Cette méthode bloque l’exécution du programme serveur dans l’attente d’une demande de connexion d’un client. Elle renvoie un objet *Socket* une fois la connexion établie.

Si vous ne voulez pas bloquer l’exécution du programme il suffit de placer l’appel à **accept** dans un thread spécifique.

##### **public void close () throws IOException**

Cette méthode ferme le socket serveur en libérant le port. Les sockets serveurs sont eux aussi fermés automatiquement par le système à la fermeture de l'application.

### 5.3.3. Méthodes informatives

```
public InetAddress getInetAddress ()  
public int getLocalPort ()
```

Ces méthodes renvoient l'adresse Internet et le port locaux sur lequel le socket attends les connexions.

### 5.3.4. Options des sockets serveurs

Seule l'option `SO_TIMEOUT` est disponible pour les sockets serveurs.

```
public int getSoTimeout() throws SocketException  
public void setSoTimeout(int ms) throws SocketException
```

Par défaut, l'appel à **accept()** se bloque jusqu'à la réception d'une demande de connexion. Lorsque `SO_TIMEOUT` est initialisée, cette attente ne dépasse pas le temps imparti, exprimé en millisecondes. Tout dépassement de temps se solde par une *InterruptedException*. Le socket reste malgré tout connecté.

La valeur par défaut est 0 qui signifie, ici, un laps de temps infini ce qui convient à la plupart des serveurs, conçus en général pour s'exécuter indéfiniment.

L'option doit être validée avant l'appel à **accept()** pour être prise en compte.

Le paramètre doit être  $\geq 0$  sans quoi une exception *SocketException* est générée.

### 5.3.5. Méthode surchargée

```
public String toString()
```

La seule méthode surchargée de la classe *Object* est **toString()**. Cette méthode provoque l'affichage d'une chaîne ressemblant à cela :

```
ServerSocket[addr=0.0.0.0/0.0.0.0,port=0,localport=80]
```

Sous Java 1.1 et les précédentes versions `addr` et `port` sont toujours nulles. La seule information utile est donc `localport` qui indique le numéro du port d'attente des connexions. Cette méthode est ici aussi plutôt destinée au débogage.

#### **5.4. Exemple de programme client**

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

public class ClientEcho extends Object {

    public static void main (String args[]) {
        String          reponse;
        Socket          leSocket;
        PrintStream     fluxSortieSocket;
        BufferedReader  fluxEntreeSocket;

        try {
            // creation d'une socket et connexion à la machine marine sur le port numéro 7
            leSocket = new Socket("marine.edu.ups-tlse.fr", 7);

            System.out.println("Connecté sur : "+leSocket);

            // création d'un flux de type PrintStream lié au flux de sortie de la socket
            fluxSortieSocket = new PrintStream(leSocket.getOutputStream());
            // creation d'un flux de type BufferedReader lié au flux d'entrée de la socket
            fluxEntreeSocket = new BufferedReader(new
                InputStreamReader(leSocket.getInputStream()));

            // envoi de données vers le serveur
            fluxSortieSocket.println("Bonjour le monde!");

            // attente puis réception de données envoyées par le serveur
            reponse = fluxEntreeSocket.readLine();

            System.out.println("Reponse du serveur : " + reponse);

            leSocket.close();
        } // try
        catch (UnknownHostException ex)
        {
            System.err.println("Machine inconnue : "+ex);
            ex.printStackTrace();
        }
        catch (IOException ex)
        {
            System.err.println("Erreur : "+ex);
            ex.printStackTrace();
        }
    } // main
} // class
```

#### **5.5. Exemple de programme serveur**

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

public class ServeurEcho extends Object {

    public static void main (String args[]) {
        ServerSocket    socketEcoule;
        Socket          socketService;
        InputStream     entreeSocket;
```

```

OutputStream    sortieSocket;

try {
    // création du socket d'écoute (port numéro 7)
    socketEcoute = new ServerSocket(7);

    while (true) {
        // attente d'une demande de connexion
        socketService = socketEcoute.accept();

        System.out.println("Nouvelle connexion : " + socketService);

        // récupération des flux d'entrée/sortie de la socket de service
        entreeSocket = socketService.getInputStream();
        sortieSocket = socketService.getOutputStream();

        try {
            int b = 0;
            while (b != -1) {
                b = entreeSocket.read();
                sortieSocket.write(b);
            } // while
            System.out.println("Fin de connexion");
        } // try
        catch (IOException ex)
        {
            // fin de connexion
            System.out.println("Fin de connexion : "+ex);
            ex.printStackTrace();
        }

        socketService.close();
    } // while (true)
} // try
catch (Exception ex)
{
    // erreur de connexion
    System.err.println("Une erreur est survenue : "+ex);
    ex.printStackTrace();
}
} // main
} // class

```

## 6. Sockets UDP

Le protocole UDP offre un service non connecté et non fiable. Les données peuvent être perdues, dupliquées ou délivrées dans un ordre différent de leur ordre d'émission.

Le fonctionnement est ici symétrique. Chaque processus crée un socket et l'utilise pour envoyer ou attendre et recevoir des données.

En Java on utilise la classe `DatagramPacket` pour représenter les datagrammes UDP qui sont échangés entre les machines.

### 6.1. La classe `DatagramPacket`

#### 6.1.1. Constructeurs

```
public DatagramPacket(byte[] tampon, int longueur)
```

Ce constructeur crée un objet utilisé pour recevoir les données contenues dans des datagrammes UDP reçus par la machine. Le paramètre tampon doit correspondre à un tableau d'octets (type byte en java) correctement créé (attention si le tableau n'est pas assez grand les données en excès seront détruites). Le paramètre longueur indique la longueur du tableau.

Exemple d'utilisation :

```
byte[] tampon = new byte[1024];
DatagramPacket datagramme = new DatagramPacket(tampon, tampon.length);
```

```
public DatagramPacket(byte[] tampon,
                      int longueur,
                      InetAddress adresse,
                      int port)
```

Ce constructeur crée un objet utilisé pour envoyer un datagramme UDP. Le paramètre tampon doit correspondre aux données à envoyer. Le paramètre longueur doit indiquer la longueur des données à envoyer. Le paramètre adresse et le paramètre port doivent indiquer respectivement l'adresse IP et le port de destination.

Il est à noter qu'il existe d'autres constructeurs qui permette d'indiquer un offset dans un tableau de grande dimension. Ils doivent être utilisés pour envoyer un tableau dont la taille dépasse la taille maximale des datagrammes IP.

Exemple d'utilisation :

```
String message = "Bonjour le monde!";
byte[] tampon = message.getBytes();
InetAddress adresse = InetAddress.getByName("marine.edu.ups-tlse.fr");
DatagramPacket datagramme = new DatagramPacket(tampon, tampon.length, adresse, 7);
```

## 6.1.2. Accesseurs

```
public byte[] getData()
public void setData(byte[] buf)
```

Ces méthodes permettent de récupérer et de changer le tableau d'octets utilisé soit pour recevoir soit pour envoyer un datagramme UDP.

```
public void setLength(int length)
public int getLength()
```

Ces méthodes permettent de récupérer et de changer la longueur du tableau d'octets utilisé soit pour recevoir soit pour envoyer un datagramme UDP.

```
public InetAddress getAddress()
public void setAddress(InetAddress iaddr)
public int getPort()
public void setPort(int iport)
```

Ces méthodes permettent de récupérer et de changer les adresses IP et numéros de ports. Si l'objet est utilisé pour envoyer des datagrammes UDP il s'agira de l'adresse et du port de destination sinon il s'agira de l'adresse et du port de l'émetteur du datagramme UDP.

## 6.2. La classe *DatagramSocket*

### 6.2.1. Constructeurs

```
public DatagramSocket() throws SocketException
```

Ce constructeur crée un socket UDP qui permet d'envoyer et recevoir des datagrammes UDP. Le port qu'utilise ce socket est choisi par le système d'exploitation. De même l'adresse IP qu'utilise ce socket est choisie automatiquement par le système.

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

Ce constructeur crée un socket UDP qui permet d'envoyer et recevoir des datagrammes UDP. Le port qu'utilise ce socket est indiqué par le paramètre port. L'adresse IP qu'utilise ce socket est choisie automatiquement par le système.

```
public DatagramSocket(int port, InetAddress adresse) throws SocketException
```

Ce constructeur crée un socket UDP qui permet d'envoyer et recevoir des datagrammes UDP. Le port qu'utilise ce socket est indiqué par le paramètre port. L'adresse IP qu'utilise ce socket est indiqué par le paramètre adresse.

Ces trois constructeurs peuvent générer des exceptions de type `SocketException` si il y a un problème de configuration réseau, si le port choisi est déjà utilisé ou si l'adresse choisie n'est pas l'une des adresses de la machine locale.

## 6.2.2. Emission et Réception de Datagrammes

```
public void send(DatagramPacket p) throws IOException
```

Envoie un datagramme UDP qui contiendra toutes les informations référencées par l'objet p.

```
public void receive(DatagramPacket p) throws IOException
```

Attends un datagramme UDP et lors de sa réception stocke ses informations dans l'objet p.

Des exceptions de type `IOException` peuvent être générées si un problème d'entrée/sortie survient.

## 6.2.3. Méthodes informatives

```
public InetAddress getLocalAddress ()  
public int getLocalPort ()
```

Ces méthodes renvoient l'adresse Internet et le port locaux que le socket utilise.

## 6.2.4. Fermeture du socket

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

Bien que Java ferme tous les sockets ouverts lorsqu'un programme se termine ou bien lors d'un « garbage collect », il est fortement conseillé de fermer explicitement les sockets dont on n'a plus besoin à l'aide de la méthode `close`.

Une fois un socket fermé on peut toujours utiliser les méthodes informatives, par contre toute tentative d'émission ou de réception de datagrammes UDP provoque une *IOException*.

## 6.2.5. Options

Comme pour les sockets UDP on peut positionner un certain nombre d'options qui modifient le comportement par défaut des sockets.

```
public int getSoTimeout() throws SocketException  
public void setSoTimeout(int ms) throws SocketException
```

Par défaut, lors d'une tentative de réception d'un datagramme UDP, **receive** se bloque jusqu'à la réception d'un datagramme. Lorsque **SO\_TIMEOUT** est initialisée, cette attente ne dépasse pas le temps imparti, exprimé en millisecondes. Tout dépassement de temps se solde par une **java.net.SocketTimeoutException**. Le socket reste malgré tout connecté.

La valeur par défaut est 0 qui signifie, ici, un laps de temps infini. Il peut être préférable de choisir une durée raisonnable (>0) pour éviter que le programme se bloque en cas de problème.

L'option doit être validée avant l'appel à l'opération bloquante pour être prise en compte.

Le paramètre doit être  $\geq 0$  sans quoi une exception *SocketException* est générée.

```
public int getSendBufferSize() throws SocketException  
public void setSendBufferSize(int size) throws SocketException  
public int getReceiveBufferSize() throws SocketException  
public void setReceiveBufferSize(int size) throws SocketException
```

Ces méthodes permettent, grâce aux options **SO\_SNDBUF** et **SO\_RCVBUF**, de préciser une indication sur la taille à allouer aux tampons d'entrée/sortie bas niveaux. Il s'agit uniquement d'une indication donc il est conseillé après l'appel d'une méthode **set...** d'appeler la méthode **get...** correspondante pour vérifier la taille allouée.

```
public void setReuseAddress(boolean on) throws SocketException  
public boolean getReuseAddress() throws SocketException
```

Ces méthodes permettent, grâce à l'option **SO\_REUSEADDR**, de permettre à plusieurs sockets d'utiliser le même numéro de port. L'option est désactivée par défaut. Cette option peut être utilisée lors de diffusions (broadcast ou multicast – voir aussi la classe suivante).

## 6.3. La classe *MulticastSocket*

Cette classe permet d'utiliser le multicasting IP pour envoyer des datagrammes UDP à un ensemble de machines repéré grâce à une adresse multicast (classe D dans IP version 4 : de 224.0.0.1 à 239.255.255.255).

### 6.3.1. Constructeurs

Les constructeurs de cette classe sont identiques à ceux de la classe *DatagramSocket*.

### 6.3.2. Abonnement/résiliation

Pour pouvoir recevoir des datagrammes UDP envoyés grâce au multicasting IP il faut s'abonner à une adresse multicast (de classe D pour IP version 4). De même lorsqu'on ne souhaite plus recevoir des datagrammes UDP envoyés à une adresse multicast on doit indiquer la résiliation de l'abonnement.

```
public void joinGroup(InetAddress adresseMulticast) throws IOException
```

Cette méthode permet de s'abonner à l'adresse multicast donnée.

Note : un même socket peut s'abonner à plusieurs adresses multicast simultanément. D'autre part il n'est pas nécessaire de s'abonner à une adresse multicast si on veut juste envoyer des datagrammes à cette adresse.

Une `IOException` est générée si l'adresse n'est pas une adresse multicast ou si il y a un problème de configuration réseau.

```
public void leaveGroup(InetAddress adresseMulticast) throws IOException
```

Cette méthode permet de résilier son abonnement à l'adresse multicast donnée.

Une `IOException` est générée si l'adresse n'est pas une adresse multicast, si la socket n'était pas abonnée à cette adresse ou si il y a un problème de configuration réseau.

### 6.3.3. Choix du TTL

Dans les datagrammes IP se trouve un champ spécifique appelé TTL (Time To Live – durée de vie) qui est normalement initialisé à 255 et décrémente par chaque routeur que le datagramme traverse. Lorsque ce champ atteint la valeur 0 le datagramme est détruit et une erreur ICMP est envoyé à l'émetteur du datagramme. Cela permet d'éviter que des datagrammes tournent infiniment à l'intérieur d'un réseau IP.

Le multicasting IP utilise ce champ pour limiter la portée de la diffusion. Par exemple avec un TTL de 1 la diffusion d'un datagramme est limitée au réseau local.

```
public int getTimeToLive() throws IOException
public void setTimeToLive(int ttl) throws IOException
```

Ces méthodes permettent la consultation et la modification du champ TTL qui sera écrit dans les datagrammes envoyés par ce socket.

## 6.4. Exemple de programme qui envoie un datagramme

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

class EnvoiDatagramme
{
    public static void main(String argv[])
        throws SocketException, IOException,
            UnknownHostException
    {
        String message = "Bonjour le monde!";
        byte[] tampon = message.getBytes();
        InetAddress adresse = null;
        DatagramSocket socket;

        // recupère l'adresse IP de la machine marine
        adresse = InetAddress.getByName("marine.edu.ups-tlse.fr");

        // crée l'objet qui stockera les données du datagramme à envoyer
        DatagramPacket envoi=
            new DatagramPacket(tampon, tampon.length, adresse, 50000);

        // crée un socket UDP (le port est choisi par le système)
        socket=new DatagramSocket();

        // envoie le datagramme UDP
        socket.send(envoi);
    }
}
```

```
}
```

## 6.5. Exemple de programme qui reçoit un datagramme

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

class ReceptionDatagramme
{
    public static void main(String argv[])
        throws SocketException, IOException
    {
        byte[] tampon = new byte[1000];
        String texte;
        // crée un socket UDP qui attends des datagrammes sur le port 50000
        DatagramSocket
            socket =new DatagramSocket(50000);
        // crée un objet pour stocker les données du datagramme attendu
        DatagramPacket
            reception = new DatagramPacket(tampon, tampon.length);

        // attends puis récupère les données du datagramme
        socket.receive(reception);

        // récupère la chaîne de caractère reçue
        // Note: reception.getLength() contient le nombre d'octets reçus
        texte=new String(tampon, 0, reception.getLength());
        System.out.println("Reception de la machine "+
            reception.getAddress().getHostName()+
            " sur le port "
            +reception.getPort()+" :\n"+
            texte );
    }
}
```

## 6.6. Exemple de programme qui envoie et reçoit des datagrammes avec le multicasting IP

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

class MulticastingIP
{
    public static void main(String argv[])
        throws SocketException, IOException
    {
        String msg = "Bonjour le monde!";
        // on travaillera avec l'adresse multicast 228.5.6.7
        InetAddress groupe = InetAddress.getByName("228.5.6.7");
        // crée le socket utilisé pour émettre et recevoir les datagrammes
        // il utilisera le port 50000
        MulticastSocket s = new MulticastSocket(50000);
        // s'abonne à l'adresse IP multicast
        s.joinGroup(groupe);
        // crée l'objet qui stocke les données du datagramme à envoyer
        DatagramPacket envoi = new DatagramPacket(msg.getBytes(), msg.length(),
            groupe, 50000);
        // envoie le datagramme a tout le monde
    }
}
```

```

s.send(envoi);
while (true) {
    byte[] tampon = new byte[1024];
    DatagramPacket reception = new DatagramPacket(tampon, tampon.length);
    // attends les réponses
    s.receive(reception);
    String texte=new String(tampon, 0, reception.getLength());
    System.out.println("Reception de la machine "+
        reception.getAddress().getHostName()+
        " sur le port "
        +reception.getPort()+" :\n"+
        texte );
}
// si la boucle n'était pas infinie on pourrait écrire:
// s.leaveGroup(groupe);
}
}

```