

## Chapitre 2

# Programmation réseau avec les sockets java

Mme L Zenaidi-Belkhodja  
 lzenaidi7@gmail.com  
 Département Informatique  
 Université de Sidi Bel Abbès

## Introduction

- Les échanges avec le réseau sont devenus omniprésents dans les applications et entre les applications. Ils permettent notamment :
  - un accès à un serveur ou à une base de données;
  - d'invoquer des services distants;
  - de développer des applications web;
  - d'échanger des données entre applications.
- La plupart de ces fonctionnalités sont mises en œuvre grâce à des API de haut niveau mais celles-ci utilisent généralement des API de bas niveau pour interagir avec le réseau.
- Java fournit plusieurs classes et interfaces destinées à faciliter l'utilisation du réseau par programmation en reposant sur les sockets. Celles-ci peuvent être mises en œuvre pour réaliser des échanges utilisant le protocole réseau IP avec les protocoles de transport TCP ou UDP. Les échanges se font entre deux parties : un client et un serveur.

L Zenaidi-Belkhodja  
 Chapitre2:Sockets Java

2

## Notion de Socket

### Origine

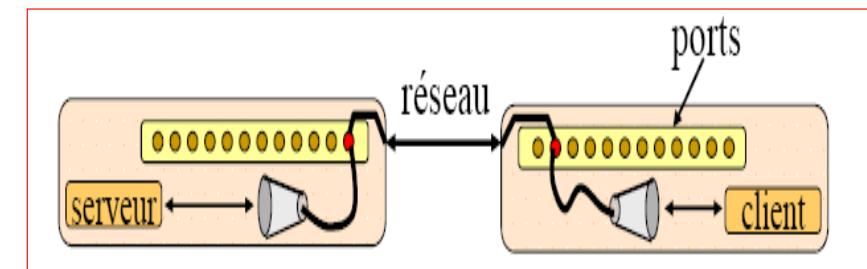
- Les sockets ont été lancés par l'université de Berkeley au début des années 1980 pour son système UNIX. Aujourd'hui, cette interface de programmation est proposée dans quasiment tous les langages de programmation populaires (Java, C#, C++, PHP, ...).
- Les sockets représentent des « interfaces de connexion », des « ports », des « points de connexion », des « connecteurs logiciels » par lesquelles une application peut se brancher à un réseau et communiquer avec une autre application branchée depuis un autre ordinateur.
- Avant les sockets, le seul mécanisme standard qui permettait à deux processus de communiquer se faisait par l'intermédiaire des pipes.

L Zenaidi-Belkhodja  
 Chapitre2:Sockets Java

3

## Notion de Socket

- Une socket (prise) représente un point de communication entre un processus et un réseau.
- Un processus client et un processus serveur, lorsqu'ils communiquent, ouvrent donc chacun une socket.
- A chaque socket est associé un port de connexion. Ces numéros de port sont uniques sur un système donné, une application pouvant en utiliser plusieurs.



L Zenaidi-Belkhodja  
 Chapitre2:Sockets Java

4

## Notion port

Un port est identifié par un entier sur 16 bits délimitant ainsi ses valeurs entre 0 et 65535. Trois classes de ports ont été définies:

-**Les ports connus (réservés):** compris entre 0 et 1023, réservés à des services et applications connus.

-**Les ports enregistrés:** compris entre 1024 et 49151, affectés à des processus ou applications d'utilisateurs.

-**Les ports dynamiques ou privés (éphémères):** compris entre 49152 et 65535, affectés de façon dynamique à des applications clientes lorsqu'une connexion est initiée.

## Quelques ports réservés

Nom du port	Numéro Port/Protocole	Service
ftp	21/tcp	File transfert protocole
smtp	25/tcp	mail sortant
pop-3	110/tcp	mail entrant
www	80/tcp	Web (http)
DNS	53/udp	système de noms de domaine
Telnet	23/tcp	terminal network

## Sockets en java

Les sockets en Java sont orientées transport (couche 4 du modèle OSI). Une socket est identifiée par le système comme une combinaison unique de :

- Protocole et un mode de communication
  - TCP (Transmission Control Protocol) : mode connecté
  - UDP (User Datagram Protocol) : mode non connecté
- Adresse socket locale
  - Une adresse local IP
  - Un numéro de port local
- Adresse socket du nœud distant
  - Une adresse éloignée IP
  - Un numéro de port éloigné

## Modes de communication :connecté /non connecté

### Mode connecté

La communication entre un client et un serveur est précédée d'une connexion et suivi d'une fermeture:

- La connexion est durable , l'adresse de destination n'est pas nécessaire à chaque envoi de données .
- Utilise le protocole TCP et les Flux (la transmission de données sous la forme d'un flux d'octets).
- Sans perte : un message arrive au moins un fois.
- Sans duplication : un message arrive au plus une fois.
- Ordre respecté : Communication de type téléphone.
- Permet un meilleur contrôle des arrivées/départs de clients.
- Utilisé uniquement en communication unicast (point à point ).
- Plus lent au démarrage.

## Modes de communication connecté /non connecté

### Mode non connecté

- Utilise le **protocole UDP** et les **datagrammes** (trames, paquets).
- Plus facile à mettre en œuvre.
- Plus rapide au démarrage.
- Permet une connexion de type point à point ou de type multipoint.
- Ne garantit pas que les données arriveront dans l'ordre d'émission.
- Nécessite l'adresse de destination à chaque envoi, et aucun accusé de réception n'est retourné.

Mode	Couche transport	Couche application
Connecté	TCP	FTP, Telnet, SMTP, JDBC, HTTP, NFS
Non connecté	UDP	TFTP, NFS, DNS

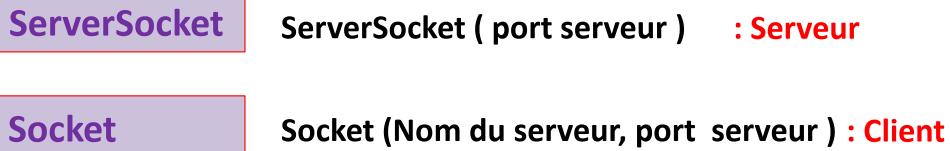
## Modes de communication

### Quand utiliser un protocole UDP?

- Lorsque les échanges sont brefs ( réponse en un message).
- Pour les applications qui doivent privilégier la rapidité à la sécurité:
- Dans une application qui fait du **streaming vidéo**, si une donnée se perd, cela n'est même pas remarqué. Si l'application utilise le protocole TCP, on devra attendre que cette donnée soit envoyée de nouveau avant de poursuivre la lecture du film!
- Dans les **jeux vidéos multi-joueurs en ligne**: les données des joueurs sont mises à jour plusieurs fois par secondes : si une donnée est manquante, ce n'est pas grave, mais si l'application se bloque, ça devient gênant.

## Classes de manipulation de Sockets en Java

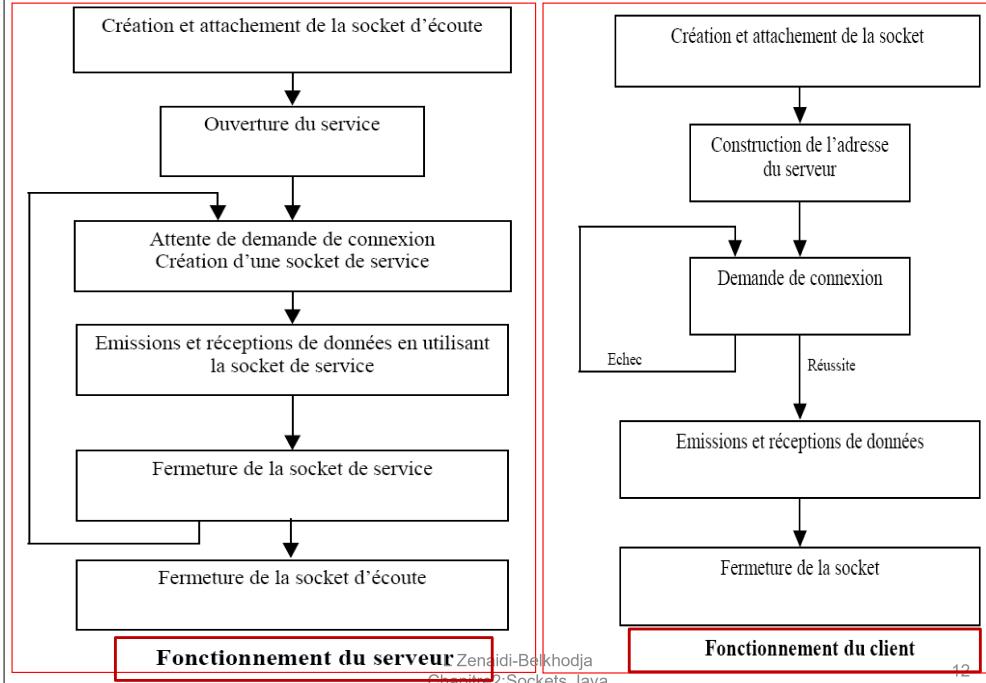
### Sockets TCP :



### Sockets UDP :



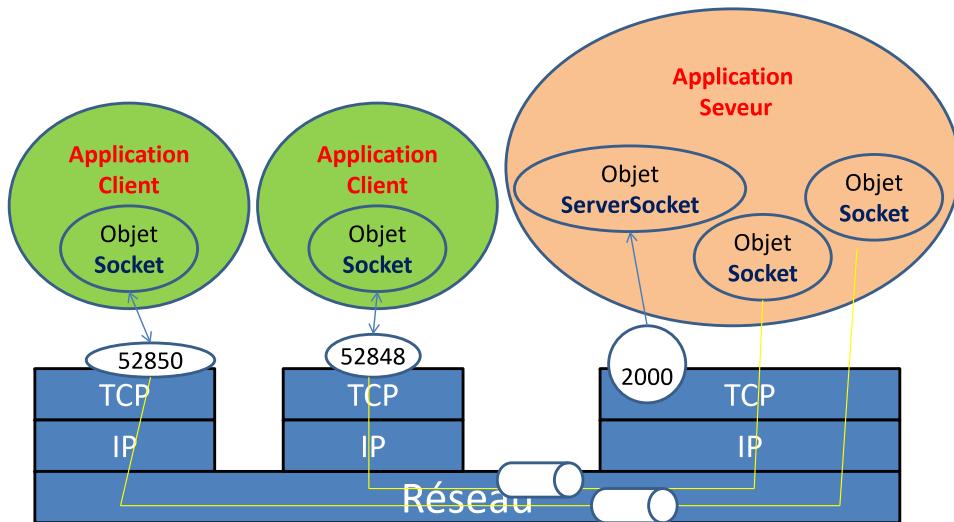
## Sockets TCP : Principe de fonctionnement



## Sockets TCP en Java

### Principe de fonctionnement

L'objet **ServerSocket** sert à établir des connexions (répond aux appels) pour ensuite laisser un objet **Socket** la gestion de chaque connexion.



L Zenaidi-Belkhodja  
Chapitre2:Sockets Java

13

## Sockets TCP en Java

### Principe de fonctionnement

- **Le serveur enregistre** son service sous un numéro de port, indiquant le nombre de clients qu'il accepte à un instant T: `new ServerSocket(...);`
- **Le serveur** se met en **attente d'une connexion** : méthode `accept()` de `ServerSocket`;
- **Le client** peut alors **établir une connexion** en demandant la création d'une socket (`new Socket()`) à destination du serveur pour le port sur lequel le service a été enregistré;
- **Le serveur sort de son accept()** et récupère une Socket de communication avec le client;
- **Le client et le serveur** peuvent alors utiliser des `InputStream` et `OutputStream` pour échanger les données.

L Zenaidi-Belkhodja  
Chapitre2:Sockets Java

14

## Sockets TCP en Java

### Principe de fonctionnement

#### Serveur

##### 1) Enregister le service:

```
ServerSocket server = new
ServerSocket ( port )
```

##### 2) Attendre une connexion client

```
Socket socket = server.accept()
```

##### 3) Utiliser la socket

InputStream



OutputStream

#### Client

##### 1) Etablir la connexion: Création d'un objet Socket

```
Socket socket = new Socket(adr, port);
```

##### 2) Utiliser la socket

OutputStream



InputStream

Close()

Close()

L Zenaidi-Belkhodja  
Chapitre2:Sockets Java

15

## Sockets TCP en Java

### 1- La classe `java.net.ServerSocket`

Utilisée par **le serveur** pour accepter les requêtes des clients.

#### Principaux constructeurs

```
ServerSocket server = new ServerSocket ( );
ServerSocket server = new ServerSocket ( int port );
ServerSocket server = new ServerSocket ( int port, int nbmax );
ServerSocket server = new ServerSocket ( int port, int nbmax, InetAddress
adresseLocale );
```

les appels entrants.

- **nbmax** : *nombre maximum de demandes stockables dans une file FIFO (50 par défaut).*
- *Tous les constructeurs lèvent IOException.*

L Zenaidi-Belkhodja  
Chapitre2:Sockets Java

16

## Sockets TCP en Java

### 1-La classe java.net. ServerSocket

#### Principales méthodes

```
Socket accept() throws IOException  
void close()throws IOException  
int getLocalPort()
```

La méthode **accept** : retourne une nouvelle **Socket** pour gérer la connexion. La méthode **accept()** est **bloquante**.

```
ServerSocket ecoute =new ServerSocket ( port )  
Socket service = ecoute . accept ();
```

L Zenaidi-Belkhodja  
Chapitre2:Sockets Java

17

## Sockets TCP en Java

### 1-La classe java.net. ServerSocket

#### Remarques

- Quand une ServerSocket est créée, elle essaie de s'attacher au port défini dans le constructeur. S'il existe une autre socket serveur qui écoute déjà sur le port spécifié, l'exception `java.net.BindException`, (sous classe de `IOException`) est levée.
- Aucun processus serveur ne peut être affecté à un port déjà utilisé par un autre processus serveur qu'il soit développé en java ou non.
- Une socket serveur ne peut plus être réutilisée une fois qu'elle est fermée.

L Zenaidi-Belkhodja  
Chapitre2:Sockets Java

18

## Sockets TCP en Java

### 2- La classe java.net.Socket

Utilisée par :

**un client** devant utiliser une connexion TCP pour communiquer avec un processus serveur (distant) (*utilisation constructeurs et méthodes*)

**un serveur** ayant acceptée une connexion TCP avec un client pour le transfert de données . (*utilisation des méthodes uniquement*)

#### Principaux constructeurs

```
Socket(String host,int port)  
Socket(InetAddress address, int port)  
Socket(String host,int port,InetAddress localAddr, int localPort)  
Socket(InetAddress address,int port,InetAddress localAddr, int localPort)  
Toutes ces methodes lèvent IOException.
```

L Zenaidi-Belkhodja  
Chapitre2:Sockets Java

19

## Sockets TCP en Java : 2-la classe java.net.Socket

#### Principales méthodes

```
InputStream getInputStream() throws IOException  
OutputStream getOutputStream() throws IOException  
void close()throws IOException
```

Les deux premières retournent respectivement un flux d'entrée et un flux de sortie associé à la socket courante. Ces flux de données brutes (octets) sont liés à d'autres flux offrant davantage de fonctionnalités.

La troisième ferme la socket interdisant toute entrée et toute sortie et provoque la libération des ressources.

L Zenaidi-Belkhodja  
Chapitre2:Sockets Java

20

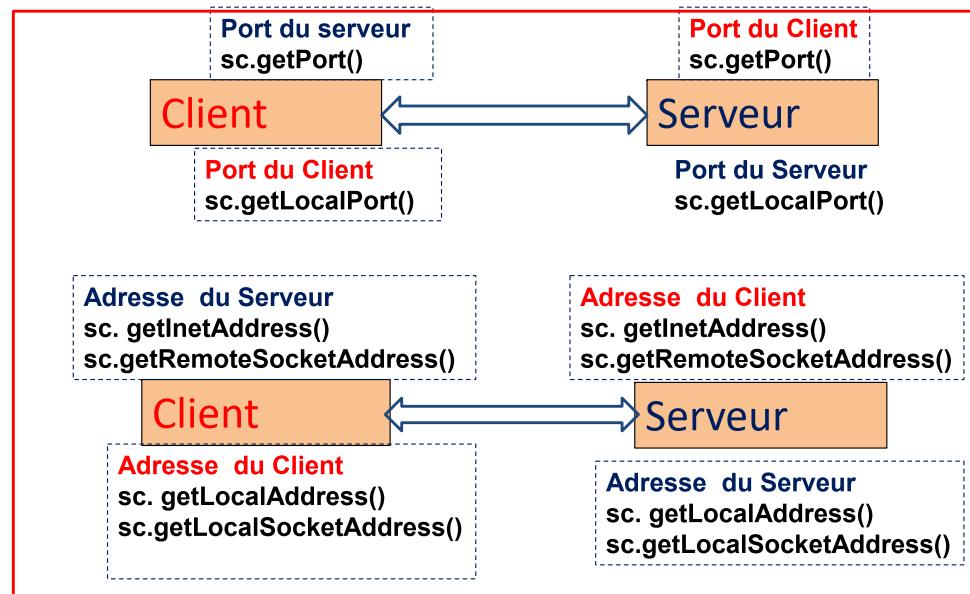
## Sockets TCP en Java : 2-la classe `java.net.Socket`

### Principales méthodes (suite)

```
InetAddress getInetAddress()  
InetAddress getLocalAddress()  
int getPort()  
int getLocalPort()  
SocketAddress getRemoteSocketAddress()  
SocketAddress getLocalSocketAddress()  
String toString()  
int getReceiveBufferSize()  
int getSendBufferSize()  
void setSoTimeout(int timeoutms)  
int getSoTimeout()
```

Accesseurs permettant de récupérer des informations sur les extrémités de la connexion TCP liée à la socket.

### Retrouver les adresses IP et les ports associés à sc de type Socket



## Sockets TCP en Java et le mode flux

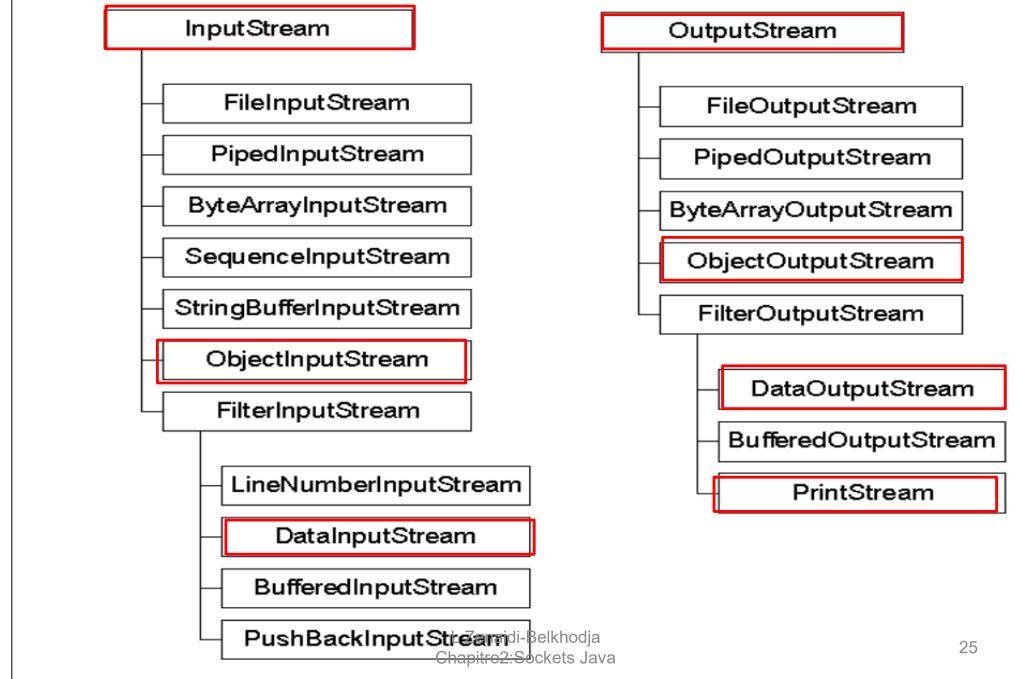
- Stream ou flux** : C'est un objet JAVA possédant la caractéristique de pouvoir envoyer ou recevoir des données.
- Toutes les entrées/sorties en JAVA sont gérées par des flux (streams) :
  - Lecture du clavier (`System.in`)
  - Affichage sur la console (`System.out`)
  - Lecture/Ecriture dans un fichier (`FileOutputStream`/  
`FileInputStream`)
  - Echange de données réseaux avec les Sockets.

## Sockets TCP en Java et le mode flux

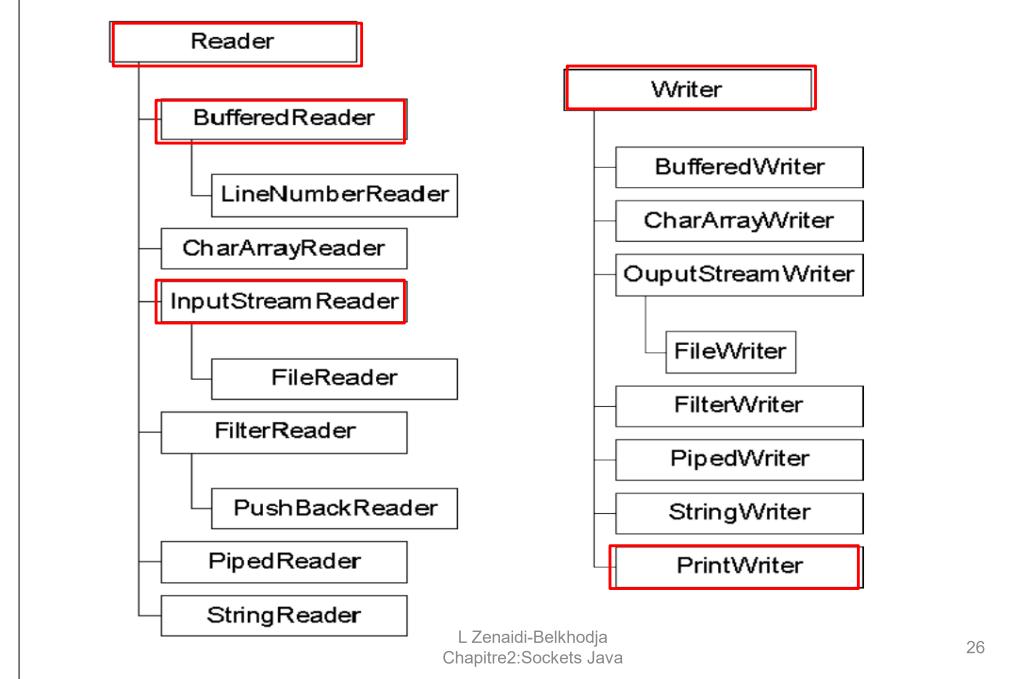
- Un flux** est un **canal de communication** dans lequel les données sont écrites ou lues de manière séquentielle.
- Un flux en lecture permet de lire les informations de manière séquentielle.
- Un flux en écriture permet d'écrire les informations de manière séquentielle.
- Les flux font partie des classes d'entrées/sorties définis dans le paquetage `java.io`. On distingue:

	Flux d'octets	Flux de caractères
Flux d'entrée	<code>InputStream</code>	<code>Reader</code>
Flux de sortie	<code>OutputStream</code>	<code>Writer</code>

## Hiérarchie des flux d'octets en Java



## Hiérarchie des flux de caractères en Java



### Les flux de caractères : Flux entrants

- obtention d'un flux simple (de communication) :  
`InputStream in = socket.getInputStream();`
- création d'un flux (de traitement) convertissant les bytes reçus en char : `InputStreamReader reader = new InputStreamReader(in);`
- création d'un flux de lecture avec tampon : pour lire ligne par ligne dans un flux de caractères:  
`BufferedReader istream = new BufferedReader(reader);`
- lecture d'une chaîne de caractères :  
`String line = istream.readLine();`

Reception de bytes



### Les flux de caractères: Flux sortants

- obtention du flux de données sortantes ( bytes):  
`OutputStream out = socket . getOutputStream () ;`
- création d'un stream qui permet la conversion d'une chaîne de caractères en bytes:  
`PrintWriter ostream = new PrintWriter(out);`
- envoi d'une ligne de caractères: `ostream.println(str);`
- envoi effectif sur le réseau des bytes : `ostream.flush();`



Remarque:

`PrintWriter(OutputStream out, boolean autoFlush) :`  
Si `autoFlush=true` l'utilisation de `flush` ne sera plus nécessaire.

## Exemple utilisant les flux de caractères

```
import java.net.*;
import java.io.*;
public class Serveurcar
{public static void main(String argv[])
 { int port = 1222;
 try { // obligatoire ?
// serveur positionne sa socket d'écoute
 ServerSocket ss = new ServerSocket(port);
// se met en attente de connexion de la part d'un client
System.out.println("Serveur en attente:");
 Socket soc = ss.accept();
// crée un flux de données en sortie
PrintWriter ostream = new PrintWriter ( soc . getOutputStream ());
String line="Hello";
// envoi de la chaîne de caractères
 ostream.println ( line );
ostream . flush (); // ici obligatoire
```

L Zenaidi-Belkhodja  
Chapitre2:Sockets Java

29

## Exemple utilisant les flux de caractères Suite du Serveur

```
// crée un flux de données en entrée
InputStreamReader reader = new InputStreamReader
(soc.getInputStream());
BufferedReader istream = new BufferedReader ( reader );

// Bloque jusqu'à l'arrivée de données
line = istream.readLine ();
System.out.println(" message reçu du client est:"+line);
// affiche les coordonnées du client qui vient de se connecter
System.out.println(" adresse client:
"+soc.getRemoteSocketAddress() );
System.out.println(" mon adresse :
"+soc.getLocalAddress()+":"+soc.getLocalPort());
}catch(Exception e) {
System.err.println("Erreur : "+e);}}
```

L Zenaidi-Belkhodja  
Chapitre2:Sockets Java

30

## Exemple utilisant les flux de caractères

```
import java.net.*;
import java.io.*;
public class Clientcar
{public static void main(String argv[])
 { int port = 1222;
 String host ="localhost";
try {
// adresse IP du serveur
InetAddress adr = InetAddress.getByName(host);
// ouverture de connexion avec le serveur sur son port d'écoute
Socket socket = new Socket(adr, port);
InputStreamReader reader = new InputStreamReader (socket.
getInputStream ());
BufferedReader istream = new BufferedReader ( reader );
String line = istream.readLine ();
System.out.println(" message reçu du serveur est:"+line);
```

L Zenaidi-Belkhodja  
Chapitre2:Sockets Java

31

## Exemple utilisant les flux de caractères Suite du Client

```
// Echo la ligne lue vers le serveur
PrintWriter ostream = new PrintWriter ( socket. getOutputStream ());
ostream . println ( line );
ostream . flush ();
System.out.println(" mon adresse client :
"+socket.getLocalAddress()+":"+socket.getLocalPort());
System.out.println(" mon serveur est :
"+socket.getInetAddress()+":"+socket.getPort());
}

catch (Exception e) {
System.err.println("Erreur : "+e);
}
}
```

L Zenaidi-Belkhodja  
Chapitre2:Sockets Java

32

## Exemple utilisant les flux de caractères

### Exécution

```
C:\ExemplesCours\char\Serveur>java Serveurcar  
Serveur en attente:  
message reçu du client est: Hello  
adresse client : /127.0.0.1:49907  
mon adresse : /127.0.0.1:1222
```

```
C:\ExemplesCours\char\Client>java Clientcar  
message reçu du serveur est:Hello  
mon adresse client : /127.0.0.1:49907  
mon serveur est : localhost/127.0.0.1:1222
```

Pour une bonne préparation de vos prochains TD et TP je vous propose d'implémenter et d'exécuter l' exemple précédent sur vos machines.

## Sockets TCP

### Méthodes manipulant les flux de communication

Peut-on utiliser uniquement les flux de communication ?

#### Quelques méthodes sur les flux en entrée (InputStream):

**int read()** : Lecture d'un octet. La valeur renvoyée est un entier compris entre 0 et 255 ou -1 si fin de flux.

**int read(byte[] tab)**: Lit une suite d'octets en les plaçant dans tab. Lit au plus la longueur de tab et retourne le nombre d'octets lu.

**void close()**: Ferme le flux.

#### Quelques méthodes sur les flux en sortie (OutputStream):

**void write(int b)**: Écrit un octet dans le flux.

**void write(byte[])**: Écrit le contenu d'un tableau d'octets dans le flux.

**void flush()**: Force l'écriture dans le flux de toutes les données à écrire, vide le tampon associé au flux en écrivant son contenu.

**void close()**: Ferme le flux.

#### Exemple d'utilisation de Méthodes manipulant les flux

```
try { // Un Serveur sans flux de traitement  
ServerSocket socketEcoute = new ServerSocket(7000);  
Socket socketService = socketEcoute.accept();  
InputStream entreeSocket = socketService.getInputStream();  
OutputStream sortieSocket = socketService.getOutputStream();  
int b = 0;  
// lecture octet par octet  
while (b != -1) {  
b = entreeSocket.read();  
sortieSocket.write(b);  
} // utilité while ?  
socketService.close();  
}  
catch (IOException ex)  
{ex.printStackTrace(); }
```

// chez le Client

```
.....  
PrintStream fluxSortieSocket;  
Socket SocketCl= new Socket("localhost", 7000);  
fluxSortieSocket = new  
PrintStream(SocketCl.getOutputStream());  
fluxSortieSocket.println("Bonjour le monde!");  
.....
```

// Que fait ce programme?

- Chez le serveur b contient des octets donc il ne sert à rien d'afficher b.
- Le client doit créer un flux de type **BufferedReader** pour pouvoir lire et afficher la réponse du serveur.

## Sockets TCP: Classes manipulant les flux Classes de flux spécialisées (de traitement)

Offrent des méthodes d'accès plus évoluées qu'au **niveau octet**.

Deux types de flux intéressants:

### 1- Data[Input/Output]Stream

**Rôle:** lecture/écriture de types primitifs Java (**int, char, boolean, double, long, byte, float, short**).

**Constructeurs:**

**DataOutputStream (OutputStream out)**

**DataInputStream (InputStream in)**

**Quelques méthodes:**

**void writeDouble(double b),double readDouble(), int readInt(),  
void writeInt(int v) ,String readLine()**

## Exemple de Méthodes manipulant les flux spécialisés

```
import java.io.*;  
import java.net.*;  
public class MyClient  
{ public static void main(String[] args)  
{ try{ Socket s=new Socket("localhost",6666);  
DataOutputStream dout=new  
DataOutputStream(s.getOutputStream());  
dout.writeDouble(1000000);  
dout.close();  
s.close(); }  
catch(Exception e)  
{System.out.println(e);} } }
```

## Exemple de Méthodes manipulant les flux spécialisés

```
import java.io.*;  
import java.net.*;  
public class MyServer  
{ public static void main(String[] args)  
{ try{ ServerSocket ss=new ServerSocket(6666);  
Socket s=ss.accept();  
DataInputStream dis=new DataInputStream(s.getInputStream());  
double str=dis.readDouble();  
System.out.println("message= "+str);  
ss.close(); }  
catch(Exception e){System.out.println(e);} } }
```

C:\Exemples\Serveur>java MyServer  
message= 1000000.0

C:\Exemples\Serveur>

## Sockets TCP Classes manipulant les flux

### 2- Object[Input/Output]Stream

**Rôle:** Lecture/écriture d'objets de toute nature, très puissante et confortable.

**Constructeurs:**

**ObjectOutputStream(OutputStream out)**

**ObjectInputStream(InputStream in)**

**Quelques méthodes:**

**void writeObject(Object o)**

**Object readObject()**

## Sockets TC

### Classes manipulant les flux

#### Exemple : envoi d'un objet de la classe Personne (classe n'appartenant pas à la hiérarchie Java)

1-Pour pouvoir envoyer ou recevoir un objet dans un flux :

- Sa classe doit implémenter l'interface **java.io.Serializable**
- Serializable est une interface vide qui sert juste à préciser que les objets peuvent être serialisés. C'est-à-dire peuvent être transformés en série de bytes et sont donc transmissibles via des flux.

2-Le client :Personne pers = new Personne ("Amina", 22);

3- **ObjectOutputStream** output = .... ;

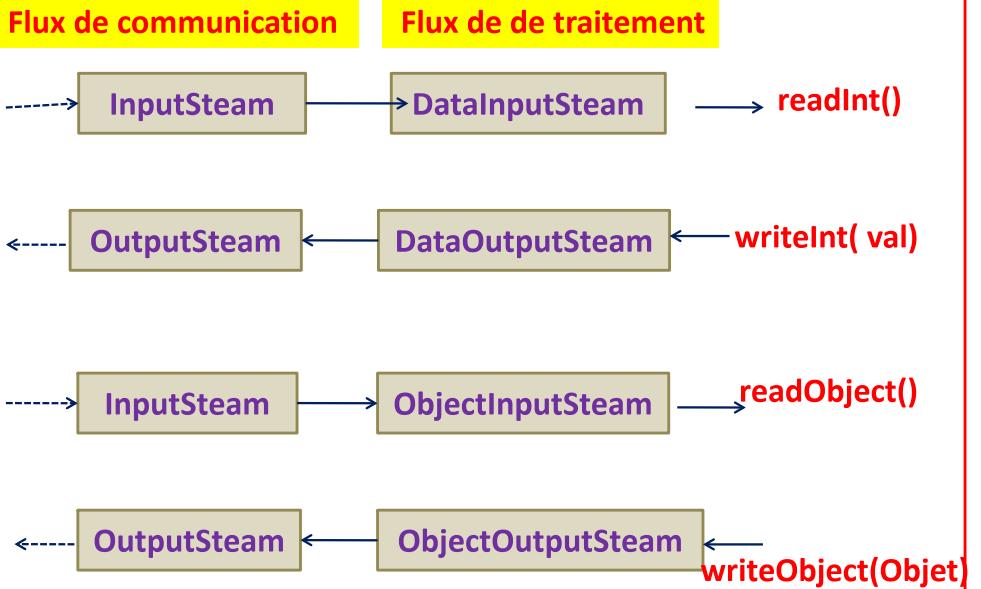
4- **output.writeObject(pers);**

### Exercice 1

En utilisant **Object[Input/Output]Stream** et les sockets java , écrire une application où le client envoie une chaîne de caractère suivie d'un entier au serveur, puis se met en attente d'une réponse du serveur.

Chacun du client et du serveur doit afficher ses coordonnées et celles de la machine distante.

## Principales classes manipulant les flux d'octets



### Exercice1 transmission de chaînes de caractères et d'entiers

```
import java.net.*;
import java.io.*;
public class ServeurTCP
{ public static void main(String argv[])
    { int port = 122;
    try {
        // serveur positionne sa socket d'écoute
        ServerSocket serverSocket = new ServerSocket(port);
        // se met en attente de connexion de la part d'un client distant
        System.out.println("Serveur en attente: ");
        Socket socket = serverSocket.accept();
        // récupère les flux objets pour communiquer avec le client
        ObjectOutputStream output =
            new ObjectOutputStream(socket.getOutputStream());
        ObjectInputStream input =
            new ObjectInputStream(socket.getInputStream());
        // attente d'une chaîne de caractère venant du client
    }
```

## Exercice1 Serveur TCP (suite)

```
String chaine = (String)input.readObject();
System.out.println(" reçu : "+chaine);
// attente d'un entier venant du client
int entier = (int)input.readObject();
System.out.println(" reçu : "+entier);
// affiche les coordonnées du client qui vient de se connecter
System.out.println(" ca vient de : "+socket.get InetAddress()":"+socket.getPort());
// affiche ses coordonnées
System.out.println(" mon adresse :
"+socket.getLocalAddress()":"+socket.getLocalPort());
// envoi d'une réponse au client
output.writeObject("message reçu");

}
catch(Exception e) {
    System.err.println("Erreur : "+e);
}
}
```

L Zenaidi-Belkhodja  
Chapitre2:Sockets Java

45

## Exercice1 Client TCP

```
import java.net.*;
import java.io.*;
public class ClientTCP
{ public static void main(String argv[])
  {
    int port = 122;int x=100;
    String host ="localhost";
    try {
        // adresse IP du serveur
        InetAddress adr = InetAddress.getByName(host);
        // ouverture de connexion avec le serveur sur son port d'écoute
        Socket socket = new Socket(adr, port);
        // construction de flux objets à partir des flux de la socket
        ObjectOutputStream output =
            new ObjectOutputStream(socket.getOutputStream());
        ObjectInputStream input =
            new ObjectInputStream(socket.getInputStream());
        // écriture d'une chaîne dans le flux de sortie : envoi de données au
        // serveur
    }
    catch(Exception e)
    {
        System.out.println("Erreur : "+e);
    }
}}
```

L Zenaidi-Belkhodja  
Chapitre2:Sockets Java

46

## Exercice1 Client TCP (suite)

```
output.writeObject("Bonjour ");
// envoi d'un entier au serveur
output.writeObject(x);
// attente de réception de données venant du serveur
String chaine = (String)input.readObject();
// conversion à la réception
System.out.println(" reçu du serveur : "+chaine);
// le client affiche ses coordonnées
System.out.println(" mon adresse client :
"+socket.getLocalAddress()":"+socket.getLocalPort());
// le client affiche les coordonnées du serveur
System.out.println(" mon serveur est :
"+socket.get InetAddress()":"+socket.getPort());
}
catch (Exception e) {
    System.err.println("Erreur : "+e);
}}
```

L Zenaidi-Belkhodja  
Chapitre2:Sockets Java

47

## Exécution de l'exercice1 Client/Serveur TCP

```
C:\ExemplesCours\Exercice1\Serveur>java ServeurTCP
Serveur en attente:
reçu : Bonjour
reçu : 100
ca vient de : /127.0.0.1:52252
mon adresse : /127.0.0.1:122
```

```
C:\ExemplesCours\Exemple1\Serveur>
```

```
C:\ExemplesCours\Exercice1\Client>java ClientTCP
reçu du serveur : message reçu
mon adresse client : /127.0.0.1:52252
mon serveur est : localhost/127.0.0.1:122
```

```
C:\ExemplesCours\Exemple1\Client>
```

L Zenaidi-Belkhodja  
Chapitre2:Sockets Java

48

## Avec les sockets TCP :

- peut-on tomber dans une situation d'interblocage?
- Un message peut-il être perdu ?

## Exercice 2

En utilisant Object[Input/Output]Stream] et les sockets java , écrire une application où le client envoi un objet de la classe Personne au serveur, puis se met en attente d'une réponse du serveur.

```
import java.net.*;
import java.io.*;
public class ServeurTCP
{ public static void main(String argv[])
    { int port = 10000;
    try {ServerSocket serverSocket = new ServerSocket(port);
    Socket socket = serverSocket.accept();
    ObjectOutputStream output = new
    ObjectOutputStream(socket.getOutputStream());
    ObjectInputStream input =
    new ObjectInputStream(socket.getInputStream());
    // le serveur attend un objet de type Personne
    Personne obj = (Personne)input.readObject();
    System.out.println(" reçu : "+obj); // et envoie une chaine
    output.writeObject("bien reçu");
    } catch(Exception e)
        System.err.println("Erreur : "+e);
    }}
```

## Exercice2 Serveur TCP

## Exercice2: Transmission d'un objet

```
import java.io.*;
public class Personne implements Serializable
{protected String nom;
protected int age;

public int getAge()
    { return age;}
public String getNom()
    { return nom;}
public Personne(String n, int a)
    { age = a;
    nom = n;
}
public String toString()
{
    return ("nom : "+nom+ " age : "+age);
}}
```

## Exercice2 Client TCP

```
import java.net.*;
import java.io.*;
public class ClientTCP
{ public static void main(String argv[])
    { int port = 10000;
    String adr ="localhost";
    try {
    Socket socket = new Socket(adr, port);
    ObjectOutputStream output =
    new ObjectOutputStream(socket.getOutputStream());
    ObjectInputStream input =
    new ObjectInputStream(socket.getInputStream());
    // le client envoie un objet de type Personne et reçoit une chaine
    output.writeObject(new Personne ("Amina", 20));
    String chaine = (String)input.readObject();
    System.out.println(" reçu du serveur : "+chaine);
    }
    catch (Exception e)
        System.err.println("Erreur : "+e);
    }}
```

## Exécution de l'exercice2

```
C:\ExemplesCours\Objet\Serveur>java ServeurTCP  
reçu : nom : Amina age : 20
```

```
C:\ExemplesCours\Objet\Serveur>
```

```
c:\ExemplesCours\Objet\Client>java ClientTCP  
reçu du serveur : bien reçu
```

```
C:\ExemplesCours\Objet\Client>
```

Dans quel répertoire (machine) doit se trouver la classe Personne ?

## Exercice 3 Serveur TCP multithreads

### Remarques sur les exemples précédents:

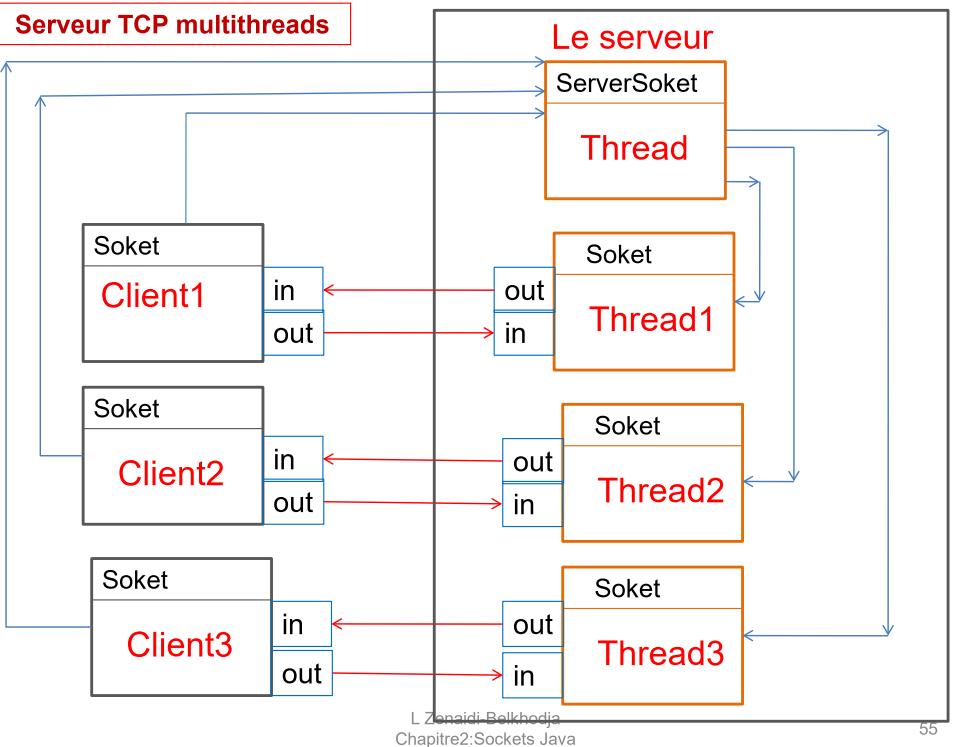
1- Le serveur accepte plusieurs connexions simultanées, mais ne traite qu'un client à la fois, les autres sont mis en attente.

2-Une fois qu'un client est traité, le serveur se termine.

Pour que le serveur puisse traiter les clients simultanément ,il faut utiliser les **threads java (java.lang.Thread)**.

### Comment développer un serveur multi threads ?

### Serveur TCP multithreads



## Exercice 3 Serveur TCP multithreads

```
import java.net.*;  
import java.io.*;  
public class ServeurTCP extends Thread  
{ private Socket socket;  
public static void main(String argv[]){  
    try {  
        ServerSocket serverSocket = new ServerSocket(122);  
        while (true) // est - il nécessaire ?  
        {Socket socketclient = serverSocket.accept();  
        ServeurTCP t=new ServeurTCP (socketclient);  
        t.start();}  
    catch (Exception e){ e.printStackTrace(); } }  
  
public ServeurTCP(Socket socket)  
{  
    this.socket=socket;  
}
```

### Exercice 3 Serveur TCP multithreads (suite)

```
public void run()
{
try
{ ObjectOutputStream output =
    new ObjectOutputStream(socket.getOutputStream());
ObjectInputStream input =
    new ObjectInputStream(socket.getInputStream());
// attente les données venant du client
String chaine = (String)input.readObject();
System.out.println("reçu : "+chaine);
output.writeObject("message reçu");
}
catch(Exception e)
{
    System.err.println("Erreur : "+e);
}
}
```

L Zenaidi-Belkhodja  
Chapitre2:Sockets Java

57

### Exécution de l'exercice 3 avec 2 clients lancés en même temps

```
C:\ExemplesCours\Thread\serveur>java ServeurTCP
reçu : Bonjour
reçu : Bonjour
```

```
C:\ExemplesCours\Thread\Client>java ClientTCP
reçu du serveur : message reçu
mon adresse client : /127.0.0.1:52252
```

```
C:\ExemplesCours\Thread\client>
```

```
C:\ExemplesCours\Thread\Client>java ClientTCP
reçu du serveur : message reçu
mon adresse client : /127.0.0.1:52253
```

```
C:\ExemplesCours\Thread\client>
```

58

Pour une bonne préparation de vos prochains TD et TP je vous propose d'implémenter et d'exécuter les exemples précédents sur vos machines.

L Zenaidi-Belkhodja  
Chapitre2:Sockets Java

59

### Sockets UDP: mode datagramme

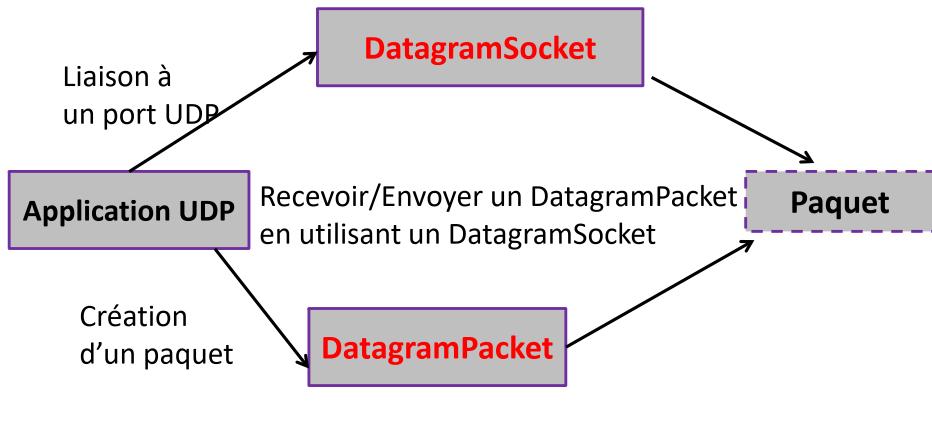
- Non connecté, de type courrier, communication par datagramme , plus rapide.
- Avec duplication : un message peut arriver plus d'une fois.
- Ordre non respecté .
- Avec perte : l'émetteur n'est pas assuré de la délivrance.
- Les données sont transmises sous forme de **tableaux de bytes**. Si le récepteur réserve un tableau trop petit par rapport à celui envoyé, une partie des données est perdue.
- Nécessite des **méthodes permettant la conversion** d'un objet quelconque en byte[] à l'envoi et d'un byte[] en un objet d'un certain type après réception.
- Les flux peuvent être utilisés pour une conversion automatique (par exemple **ByteArrayOutputStream ↔ ObjectOutputStream** ).

L Zenaidi-Belkhodja  
Chapitre2:Sockets Java

60

## Sockets UDP en mode datagramme

Deux classes sont utilisées : **DatagramPacket** et **DatagramSocket**.



## Sockets UDP en mode datagramme

### 1-La classe java.net.DatagramSocket

#### Méthodes:

```
void send(DatagramPacket p) throws IOException
void receive(DatagramPacket p) throws IOException
void close()
int getSoTimeout() throws SocketException
void setSoTimeout(int timeout) throws SocketException
int getReceiveBufferSize() throws SocketException
getLocalPort (), getLocalAddress () , getAddress(), getPort()
```

Chaque paquet envoyé doit comporter l'adresse et le numéro de port du récepteur (**particularité de UDP**).

Le receive est toujours bloquant.

Les 3 dernières méthodes ne sont pas utilisées car elles retournent, respectivement, 0.0.0.0/0.0.0.0, -1, -1.

## Sockets UDP en mode datagramme

### 1-La classe java.net.DatagramSocket

#### Constructeurs:

**DatagramSocket () throws SocketException**

**DatagramSocket ( int port ) throws SocketException**

**DatagramSocket(int port, InetAddress localAddr) throws SocketException**

Création – respectivement - d'une socket locale sur un port UDP anonyme (**un client** par exemple), ou sur un port local et écoutant sur toutes les interfaces réseau locales ou sur un port local n'écoutant que des datagrammes ayant pour destination l'adresse IP spécifiée (réduction à une seule interface réseau).

## Sockets UDP en mode datagramme

### 2-La classe java.net.DatagramPacket

#### Constructeurs:

**DatagramPacket ( byte [] buf , int length )**

**DatagramPacket(byte[] buf, int length, InetAddress address,int port)**

**DatagramPacket(byte[] buf, int offset, int length,InetAddress address, int port)**

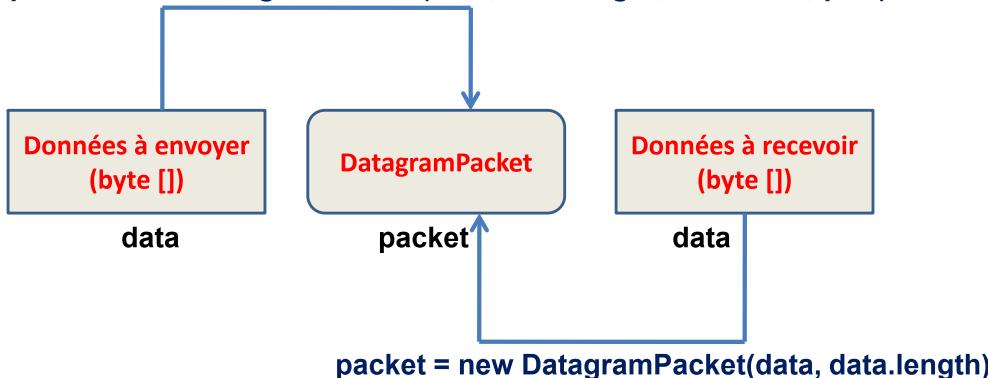
Construction d'un « objet paquet » pour l'envoi de *length* octets stockés dans *buf*, à partir du début ou depuis un déplacement d'*offset* octets vers le port UDP spécifié de **la machine distante** dont l'adresse IP est donnée.

Le premier est généralement utilisé pour la réception, les deux autres pour l'envoi.

## Sockets UDP en mode datagramme

### 2-La classe java.net.DatagramPacket

```
packet = new DatagramPacket(data, data.length, AddressIP, port)
```



## Sockets UDP en mode datagramme

### 2-La classe java.net.DatagramPacket

#### Méthodes:

`InetAddress getAddress() / setAddress(InetAddress)`  
`int getPort() / setPort(int)`  
`byte[] getData() / void setData([] byte)`  
`int getLength() / void setLength(int)`

Deux accesseurs permettant de récupérer ou de donner l'adresse IP/numéro de port de la machine distante .

Les deux suivants permettent respectivement de récupérer ou de préciser les données dont se compose le datagramme et le nombre d'octets correspondant.

## Sockets UDP mode de fonctionnement

### réception d'un datagramme:

- (Serveur: le récepteur)
1. créer un `DatagramSocket` qui écoute sur le port de la machine du destinataire.
  2. créer un `DatagramPacket` pour **recevoir** les paquets .
  3. utiliser la méthode bloquante `receive(DatagramPacket)`

### envoi d'un datagramme:

- (Client:l' émetteur)
1. créer un `DatagramSocket` sans le lier à un port.
  2. créer un `DatagramPacket` en spéciant :
    - les données à **envoyer**;
    - leur longueur;
    - la machine réceptrice;
    - le port.
  3. utiliser la méthode `send(DatagramPacket)`

## Exemple :Serveur UDP

```
import java.net.*;  
public class ServeurUDP // Recepteur  
{public static void main(String argv[]){  
    int port = 200;  
    try {  
        DatagramPacket packet;  
        // création d'une socket liée au port précisé en paramètre  
        DatagramSocket socket = new DatagramSocket(port);  
        // tableau de 15 octets qui contiendra les données reçues  
        byte[] data = new byte[15];  
        // création d'un paquet en utilisant le tableau d'octets  
        packet = new DatagramPacket(data, data.length);  
        // attente de la réception d'un paquet. Le paquet reçu est placé dans  
        // packet et ses données dans data.  
        System.out.println(" serveur en attente:");  
        socket.receive(packet);  
        // récupération et affichage des données (une chaîne de caractères)
```

## Exemple :Serveur UDP (suite)

```
//String chaine = new String(packet.getData(), 0, packet.getLength());  
String chaine = new String(packet.getData()); // new String(data);  
System.out.println(" recu : "+chaine);  
System.out.println(" ca vient de : "+packet.getAddress()+":"+  
packet.getPort());  
//le serveur envoie sa réponse  
byte[] reponse = (new String("bien recu")).getBytes();  
packet.setData(reponse);  
packet.setLength(reponse.length);  
// on envoie le paquet au client  
socket.send(packet);  
/* Que se passe t-il si les 3 dernières instructions deviennent ?  
DatagramPacket sendPacket = new DatagramPacket(reponse, reponse.length);  
DatagramPacket sendPacket = new DatagramPacket(reponse, reponse.length,  
packet.getAddress(), packet.getPort());  
socket.send(sendPacket); */  
}catch(Exception e) {  
    System.err.println("Erreur : "+e);}  
}
```

69

## Exemple : Client UDP

```
import java.net.*;  
public class ClientUDP // Emetteur  
{ public static void main(String argv[])  
{int port = 200;  
String host ="localhost";  
try {  
InetAddress adr;  
DatagramPacket packet;  
DatagramSocket socket;  
// adr contient l'@IP de la partie serveur  
adr = InetAddress.getByName(host);  
// données à envoyer : chaîne de caractères  
byte[] data = (new String("Bonjour")).getBytes();  
// création du paquet avec les données et en précisant l'adresse du  
// serveur(@IP et port sur lequel il écoute)  
packet = new DatagramPacket(data, data.length, adr, port);
```

L Zenaidi-Belkhodja  
Chapitre2:Sockets Java

70

## Exemple :Client UDP (suite)

```
// création d'une socket, sans la lier à un port particulier ?  
socket = new DatagramSocket();  
// envoi du paquet via la socket  
socket.send(packet);  
// création d'un tableau vide pour la réception  
byte[] reponse = new byte[15];  
packet.setData(reponse);  
packet.setLength(reponse.length);  
// attente paquet envoyé sur la socket du client  
socket.receive(packet);  
// récupération et affichage de la donnée contenue dans le paquet  
String chaine = new String(packet.getData());  
System.out.println(" reçu du serveur : "+chaine);  
}  
catch (Exception e) {  
    System.err.println("Erreur : "+e);  
}
```

71

L Zenaidi-Belkhodja  
Chapitre2:Sockets Java

## Exécution de l'exemple Client /Serveur UDP

```
C:\ExemplesCours\UDP>java ServeurUDP  
serveur en attente:  
recu : Bonjour  
ca vient de : /127.0.0.1:61135
```

```
C:\ExemplesCours\UDP>
```

```
C:\ExemplesCours\UDP>java ClientUDP  
recu du serveur : bien recu
```

```
C:\ExemplesCours\UDP>
```

Un serveur UDP peut –il commencer par un send ?

L Zenaidi-Belkhodja  
Chapitre2:Sockets Java

72

## Sockets en mode multicast

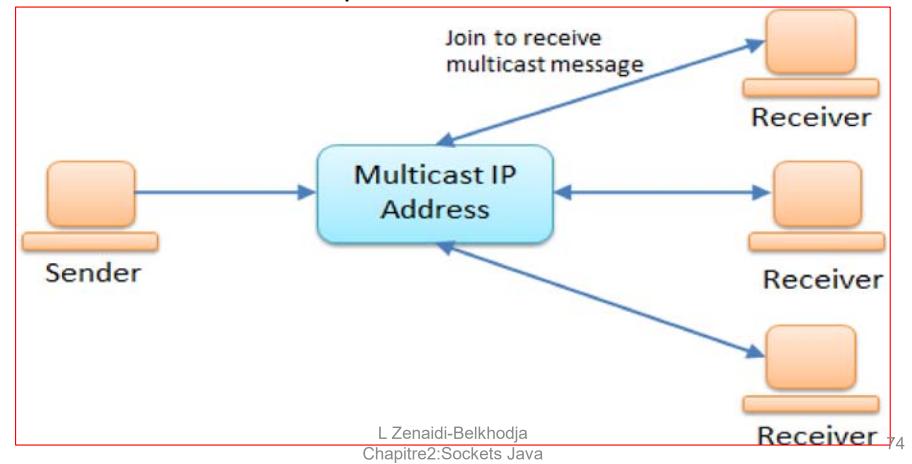
- UDP offre un mode de communication, **multicast** :plusieurs récepteurs pour une seule émission d'un paquet.
- **Broadcast** (diffusion intégrale) : envoi de données à toutes les machines du réseau. S'effectue, en pratique, en envoyant un paquet à la « dernière » adresse possible du réseau (255.255.255.255).
- **Multicast** (multidiffusion): envoi de données à **un sous-groupe** des éléments d'un réseau. Les switchs et les routeurs intermédiaires établissent une route depuis l'émetteur vers les récepteurs.
- **Multicast IP** : Envoi d'un datagramme sur une adresse IP particulière. Plusieurs éléments lisent à cette adresse IP.  
Une adresse IP multicast n'identifie pas une machine sur un réseau mais un *groupe multicast*.

L Zenaidi-Belkhodja  
Chapitre2:Sockets Java

73

## Sockets en mode multicast

- 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**.



## Sockets en mode multicast

Ce mode de communication se caractérise par:

- Une adresse IP de classe D (de **224.0.0.1 à 239.255.255.255**)
- Une diffusion de messages vers un groupe de destinataires.
- Des messages émis sur une adresse et reçus par tous les récepteurs "écoutant" sur cette adresse.
- plusieurs émetteurs possibles vers la même adresse.
- Même propriétés que UDP : taille des messages limitée à 64K, perte de messages possible, pas de contrôle de flux, ordre des messages non garanti, pas de connexion.
- Une même socket peut s'abonner à plusieurs adresses multicast simultanément.
- Il n'est pas nécessaire de s'abonner à une adresse multicast si on veut juste envoyer des datagrammes à cette adresse.

L Zenaidi-Belkhodja  
Chapitre2:Sockets Java

75

## Sockets en mode multicast

public class **MulticastSocket** extends DatagramSocket

- Constructeurs: **MulticastSocket ( port )**, **MulticastSocket ()**
- Méthodes:
- envoi : **send(DatagramPacket)**
- réception : **receive(DatagramPacket)**
- se lier à un groupe : **joinGroup(InetAddress adresseMulticast )**
- quitter un groupe : **leaveGroup(InetAddress adresseMulticast)**
- Limiter la portée des messages multicast ,en fixant le TTL :  
**setTimeToLive(int)** : le nombre de routeurs que le paquet peut traverser avant d'être arrêté :  
0 : ne dépasse pas la machine  
1 : ne dépasse pas le réseau local (par défaut)  
127 : monde entier

L Zenaidi-Belkhodja  
Chapitre2:Sockets Java

76

## Exemple : Sockets en mode multicast

```
import java.net.*;
import java.io.*;
class MulticastingIP
{
public static void main(String argv[])
throws Exception
{
String msg = "Bonjour le monde!";
// on traveillera avec l'adresse multicast 228.5.6.7
InetAddress groupe = InetAddress.getByName("228.5.6.7");
// crée la socket utilisé pour émettre et recevoir les datagrammes
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
```

L Zenaidi-Belkhodja  
Chapitre2:Sockets Java

77

## Exemple : Sockets en mode multicast (suite)

```
DatagramPacket envoi = new DatagramPacket(msg.getBytes(),
msg.length(),groupe, 50000);
// envoie le datagramme a tout le monde
s.send(envoi);
byte[] tampon = new byte[1024];
while (true) {
DatagramPacket reception = new DatagramPacket(tampon,tampon.length);
s.receive(reception);
//String texte = new String(envoi.getData());
String texte=new String(tampon);
System.out.println("Reception de la machine "+
reception.getAddress().getHostName()+" sur le port"
+reception.getPort()+" :\n"+texte );
}
// si la boucle n'était pas infinie s.leaveGroup(groupe);
}
```

L Zenaidi-Belkhodja  
Chapitre2:Sockets Java

78

## Exécution de l'exemple : Sockets en mode multicast

```
C:\ExemplesCours\multicast>java MulticastingIP
Reception de la machine mini-PC sur le port 50000 :
Bonjour le monde!
```

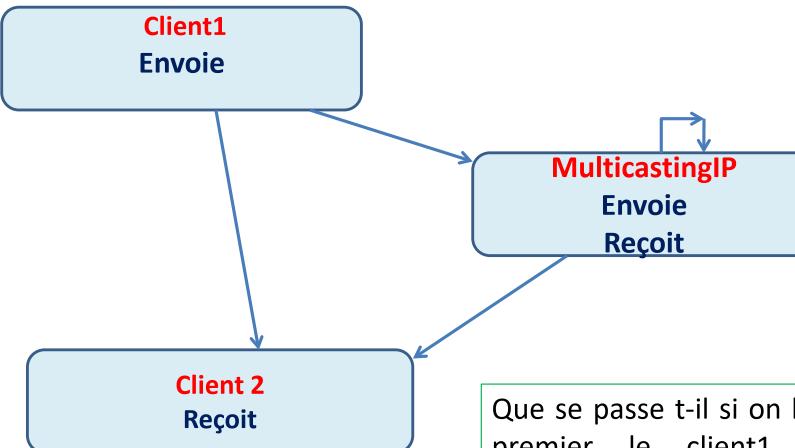
### Exercice :

- Ajouter à l'exemple (Sockets en mode multicast) un client1 qui envoie un message au groupe et un client2 qui reçoit les messages.
- L'ordre du lancement des 3 programmes est-il important?

L Zenaidi-Belkhodja  
Chapitre2:Sockets Java

79

## Schéma général de l'exercice: communication multicast



Que se passe t-il si on lance en premier le client1 ensuite MulticatingIP et enfin Client2 ?

L Zenaidi-Belkhodja  
Chapitre2:Sockets Java

80

### Exemple : Sockets en mode multicast (Client1)

```
import java.net.*;
import java.io.*;
class Client1 // Emetteur
{public static void main(String argv[]) throws SocketException,
IOException
{ String msg = "Je suis le client 1";
InetAddress groupe = InetAddress.getByName("228.5.6.7");
MulticastSocket s = new MulticastSocket(50000);
// MulticastSocket s=new MulticastSocket(); ?
//DatagramSocket s=new DatagramSocket(); ?
// n'a pas besoin de s'abonner 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 à tout le monde
s.send(envoi);
1
```

L Zenaidi-Belkhodja  
Chapitre2:Sockets Java

81

### Exemple : Sockets en mode multicast (Client2)

```
import java.net.*;
import java.io.*;
class Client2 // Recepteur
{ public static void main(String argv[]) throws SocketException,
IOException
{ InetSocketAddress groupe = InetSocketAddress.getByName("228.5.6.7");
MulticastSocket s = new MulticastSocket(50000);
s.joinGroup(groupe);
byte[] tampon = new byte[1024];
while (true) {
DatagramPacket reception = new DatagramPacket(tampon,
tampon.length);
s.receive(reception);
String texte=new String(tampon);
System.out.println("Reception de la machine "+
reception.getAddress().getHostName()+" sur le port "+
+reception.getPort()+" :\n"+texte );
}}
```

L Zenaidi-Belkhodja  
Chapitre2:Sockets Java

82

### Exécution de l'exemple avec 3 membres

```
C:\ExemplesCours\multicast\Client2>java Client2
Reception de la machine mini-PC sur sur le port 50000 :
Bonjour le monde!
Reception de la machine mni-PC sur le port 50000 :
Je suis le client 1
```

```
C:\ExemplesCours\multicast>java MulticastingIP
Reception de la machine mini-PC sur le port 50000 :
Bonjour le monde!
Reception de la machine mini-PCsur le port 50000 :
Je suis le client 1
```

```
C:\ExemplesCours\multicast\Client1>java Client1
C:\ARSR2017\ExemplesCours\multicast\Client1>
```

L Zenaidi-Belkhodja  
Chapitre2:Sockets Java

### Conclusion

Les sockets java permettent :

- une communication client / serveur entre machines distantes, en mode TCP et UDP
  - une communication unicast ou multicast.
  - Un traitement concurrents des requêtes en utilisant les threads;
  - L'envoie d'objet en utilisant la sérialisation;
  - L'envoi de fichiers en utilisant FileInputStream et FileOutputStream;
- Nécessitent l'utilisation des flux pour la transmission des données en mode TCP et de datagrammes pour le mode UDP.
- Constituent une base pour la majorité des middlewares.

### Fin du Chapitre 2

L Zenaidi-Belkhodja  
Chapitre2:Sockets Java

84