



Université Mohamed V - Rabat
Ecole Nationale Supérieure d'Informatique
et d'Analyse des Systèmes

Programmation Objet Avancée

PROF. GUERMAH HATIM
EMAIL: GUERMAH.ENSIAS@GMAIL.COM

Socket Java

Plan du cours

- Introduction
- Gestion des Flux et Sérialisation
- Socket : Définition et Principe de fonctionnement
- Servir plusieurs clients et Adressage
- Socket UDP

Introduction

Système centralisé : rassemble les ressources nécessaires à un traitement sur un hôte central avec différents mécanismes de contrôle et de coordination entre les périphériques et terminaux connectés.

Evolution de la technologie

- Plus de Performances
- Moins de Coûts
- Banalisation des réseaux de communication
- Performances des voix de communication

Evolution des Besoins

- Communication et Partage entre systèmes de plus de ressources d'information

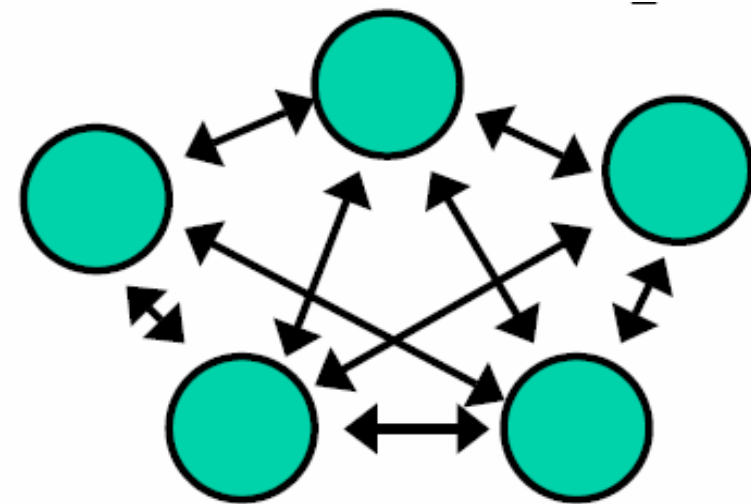
Systèmes Réparties

Systeme Répartie

- Un Systeme Réparti (distribué) est un ensemble d'entités autonomes de calcul (ordinateurs, processeurs, processus, processus léger etc.) interconnectées et qui peuvent communiquer par un réseau de communication et qui communiquent par envoi de messages.

Exemples :

- WWW, FTP, Mail
- Guichet de banque, agence de voyage.
- Téléphones portables (et bornes)
- Télévision interactive
- Agents intelligents
- Robots footballeurs



Construction du système réparti

Conception de l'architecture du système

Programmation des entités logicielles

- Utilisation d'un mécanisme de communication avec un modèle d'exécution
- Programmation en fonction du modèle d'exécution.

Configuration des entités de diverses provenances

- Méthodes de communication et d'échange des données.
- Modèle d'échange des informations de contrôle
- Médiation entre entités (permettre de se comprendre.)

Installation, déploiement et Administration

Problématique

- Créer des Systèmes ou Applications réparties en utilisant les protocoles de transport.
 - Application client-serveur :
 - un serveur fournit un service
 - un client utilise ce service
- Le client et le serveur doivent établir une connexion entre eux, ils doivent utiliser des mécanismes de communication et une interface de transport afin d'échanger des messages.
- Les communications entre les deux machines doivent être sûres, les données ne doivent pas se perdre, et arriver dans le même ordre d'émission.

Gestion du Flux

Socket

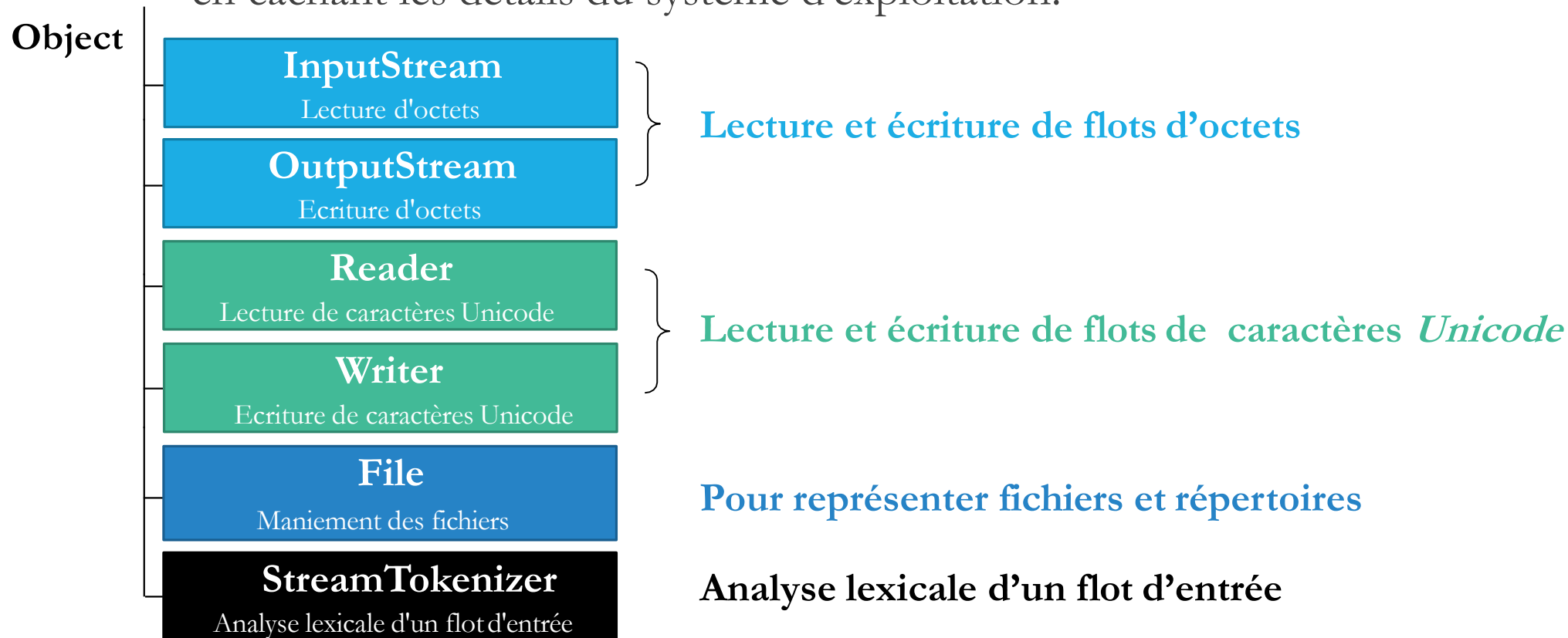
Gestion d'entrées/ Sorties

- Une entrée/sortie : un échange de données entre le programme et une autre source : lecture d'informations émises par une source externe, ou envoi d'informations à une destination externe.
- L'API d'E/S Java comprend deux types de flux :
 - Les flux de communication: représentent des destinations et des sources telles que des fichiers ou des sockets réseau.
 - Les flux de traitement qui ne fonctionnent que s'ils sont chaînés à d'autres flux.

→ En général, il est nécessaire de chaîner au moins deux flux : l'un pour représenter la connexion et l'autre pour les appels de méthodes.
- La possibilité de mélanger différentes combinaisons de flots de communication et de traitement procure une très grande souplesse et permet de personnaliser l'utilisation de ces chaînages.

Le package Java.io

- Le package java.io:
 - fournit les classes nécessaires à la création, lecture, écriture et traitement des flux
 - en cachant les détails du système d'exploitation.



Les fichiers

- Le package `java.io` regroupe l'ensemble des classes pour traiter les fichiers.
- Ce package définit une hiérarchie de classes représentant les différentes sortes de flux disponibles pour manipuler les fichiers.
- La plupart des opérations réalisées par les classes de `java.io` sont " dangereuses " , c'est pourquoi ce paquet utilise abondamment les exceptions.
- les classes manipulant les fichiers se trouvent dans la classe `java.io.File`.

Gestion du nommage et de location

- `String getName()`
- `String getPath()`
- `String getAbsolutePath()`
- `String getParent()`
- `boolean renameTo(File newName)`
- ...

Droits d'accès

- `boolean exists()`
- `boolean canWrite()`
- `boolean canRead()`
-

Autres informations

- `long length()`
- `long lastModified()`
- `boolean delete()`
-

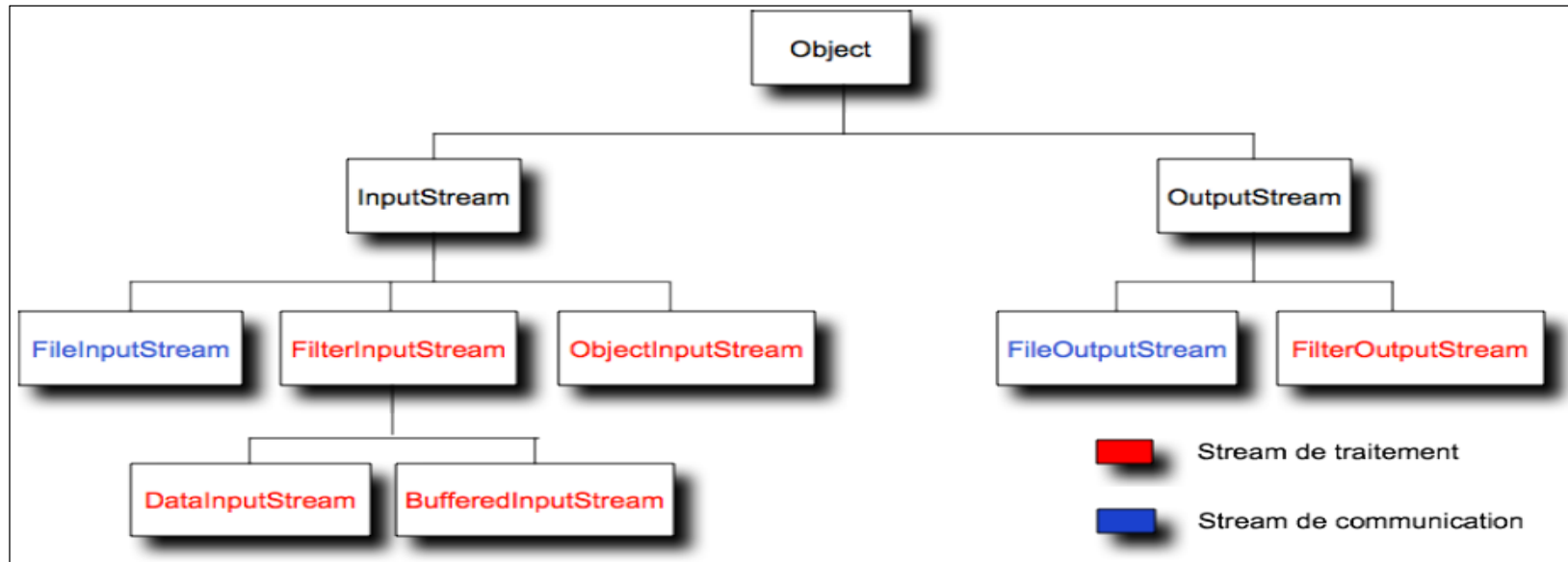
Répertoires

- `boolean mkdir()`
- `String[] list()`
-

Gestion d'entrées/ Sorties

- En java, les entrées sorties sont gérées par les objets de Flux « *Stream* » : un médiateur entre la source des données et sa destination.
- Java propose deux catégories d'objets désignant les flux :
 - les objets traitant les flux d'entrée (in) : la lecture de flux.
 - les objets traitant les flux de sortie (out) : l'écriture de flux.
- les classes d'entrées/sorties sont définies dans le paquetage java.io. :
 - Classes manipulant des octets (InputStream, OutputStream et leurs classes dérivées). Les flux binaires (octets) peuvent servir à charger en mémoire des images, les enregistrer /charger des objets ou sur le disque (sérialisation/ désérialisation).
 - Classes manipulant des caractères (Reader, Writer et leurs classes dérivées) : les caractères en java sont codés sur 16 bits (Unicode). Ces flux servent à gérer les jeux de caractères (conversion, etc.).

Byte Stream



- Un programme peut utiliser un byte stream pour réaliser entrée et sortie de bytes (8-bits). Les classes byte stream sont descendantes de `InputStream` et `OutputStream`.
- Il y a plusieurs classes byte stream. Ici, nous allons utiliser `FileInputStream` et `FileOutputStream`.

Byte Stream

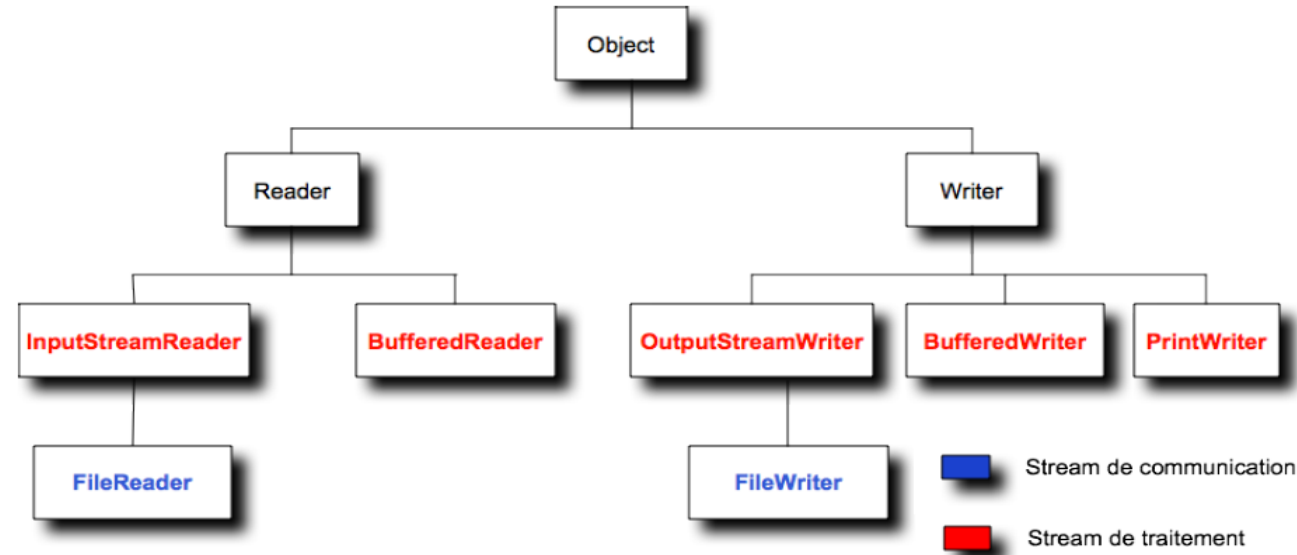
Classe pour entrée	Classe pour sortie	Fonctions fournies
InputStream	OutputStream	Classes abstraites de base pour lecture et écriture d'un flux de données
FilterInputStream	FilterOutputStream	Classe mère des classes qui ajoutent des fonctionnalités à Input/OutputStream
BufferedInputStream	BufferedOutputStream	Lecture et écriture avec buffer
DataInputStream	DataOutputStream	Lecture et écriture des types primitifs
FileInputStream	FileOutputStream	Lecture et écriture d'un fichier

Byte Stream

- Il faut toujours fermer les byte streams quand ils ne sont plus nécessaires. Cela évite les fuites de ressources.
- CopyBytes.java utilise une sorte de lecture et écriture de bas niveau. Cela n'est pas toujours approprié.
- Par exemple, si le fichier contient du texte, il vaudrait mieux utiliser un character stream.

```
import java.io.*;
public class CopyBytes {
    public static void main(String[] args) throws
        IOException {
        FileInputStream in = null;
        FileOutputStream out = null;
        try {
            in = new FileInputStream("xanadu.txt");
            out = new FileOutputStream("out.txt");
            int c;
            while ((c = in.read()) != -1)
                out.write(c);
        }
        finally {
            if (in != null) in.close();
            if (out != null) out.close();
        }
    }
}
```

Character stream



- La classe **Object** qui est la classe de base en Java, dont toutes les autres héritent ;
- Les classes abstraites **Reader** et **Writer** qui concernent respectivement les flux de caractères pour les lectures et les écritures ;
- Les classes **InputStreamReader** et **OutputStreamWriter** qui permettent de faire la traduction des données brutes en caractères UNICODE et inversement ;

Character stream

- Les classes `BufferedReader` et `BufferedWriter` qui permettent l'utilisation d'une mémoire tampon pour les entrées-sorties. Cette mémoire est indispensable pour l'utilisation des périphériques standards (écran et clavier) ;
- Les classes `FileReader` et `FileWriter` qui permettent l'utilisation de fichiers ;
- La classe `PrintWriter` qui permet l'écriture de données formatées semblables aux affichages à l'écran.
- Cette liste n'est pas exhaustive mais correspond aux principales classes que nous serons amenés à utiliser dans la suite.

```
import java.io.*;

public class CopyLines {
    public static void main(String[] args) throws
        IOException {
        BufferedReader inputStream = null;
        PrintWriter outputStream = null;
        try {
            inputStream = new BufferedReader(new
                FileReader("xanadu.txt"));
            outputStream = new PrintWriter (new
                FileWriter ("charoutput.txt"));
            String l;
            while ((l = inputStream.readLine()) != null)
                outputStream.println(l);
        }
        finally {
            if (inputStream != null) inputStream.close();
            if (outputStream != null) outputStream.close();
        }
    }
}
```

Character stream : Buffered stream

- Les buffered streams sont plus efficaces : le programme fait moins d'appel au système d'exploitation.
- La lecture et l'écriture est faite par grands blocs de données, appelées buffers .

```
InputStream = new BufferedReader(new FileReader("xanadu.txt"));  
OutputStream =new PrintWriter(new FileWriter("charoutput.txt"));
```

→ la conversion d'un unbuffered stream dans un buffered stream.

- Quand on utilise des buffers, il faut savoir que si l'exécution termine de façon inattendue, il se peut que les buffers ne soient pas encore vides.
 - Pour éviter cela, il faut parfois utiliser la méthode flush.

Sérialisation

- Certains streams (ObjectInputStream/ ObjectOutputStream) permettent d'enregistrer sur disque (sérialiser) des objets Java
 - Conserver l'état des objets entre deux exécutions d'un programme
 - Echanger des objets entre programmes
- Sérialisation : consiste à écrire des données présentes en mémoire vers un flux de données binaires, c'est donc la représentation sous forme binaire d'un objet Java
- Java a introduit des outils permettant de sérialiser les objets de manière transparente et indépendante du système d'exploitation.
- La sérialisation peut s'appliquer facilement à tous les objets.



Sérialisation

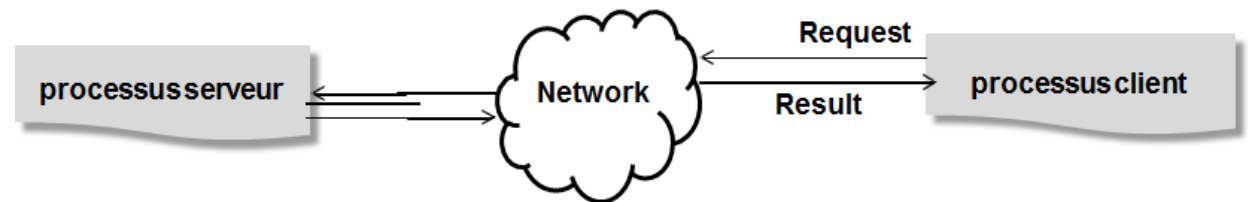
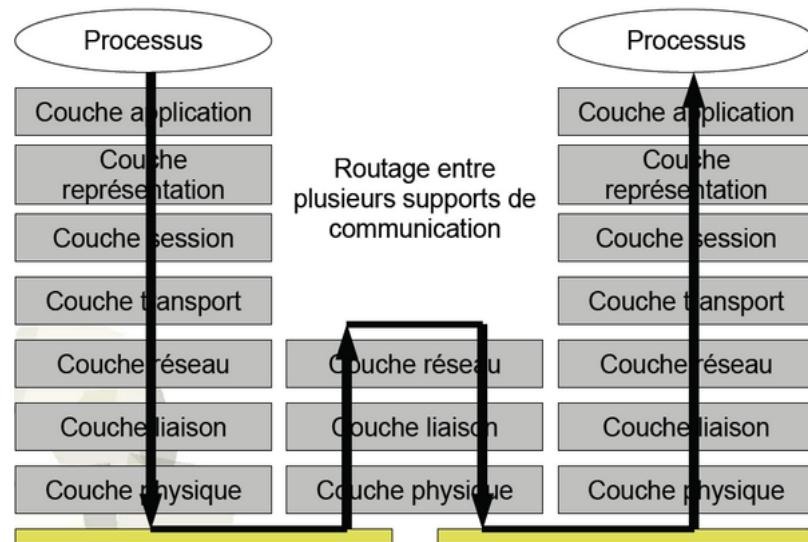
- Une classe serializable est une classe qui implémente l'interface `java.io.Serializable`. Ses instances peuvent alors être placées/lues dans/depuis un fichier
- L'interface déclare deux méthodes : `readObject()` & `writeObject(Object)`
- Note : un objet `String` est sérializable par définition

```
class Voiture implements Serializable {  
    Moteur moteur;  
    Carrosserie carrosserie;  
    transient int essence;  
    ...  
}
```

```
import java.io.* ;  
public class ExSerialise {  
    public static void main(String[] args) throws IOException {  
  
        /*On sérialise dans le fichier «garage» l'objet voiture à l'aide  
        du stream o enveloppé dans le stream f */  
        Voiture voiture = new Voiture(«V6», «Cabriolet»);  
        voiture.setCarburant(50);  
        // Stream de communication  
        FileOutputStream f = new FileOutputStream(«garage»);  
  
        // Stream de traitement  
        // f : Emboitement des streams  
        ObjectOutputStream o = new ObjectOutputStream(f);  
  
        //Sérialisation  
        o.writeObject(voiture);  
  
        // Fermeture du stream  
        o.close();  
  
    }  
}
```

Communication systèmes répartis : Client/Serveur

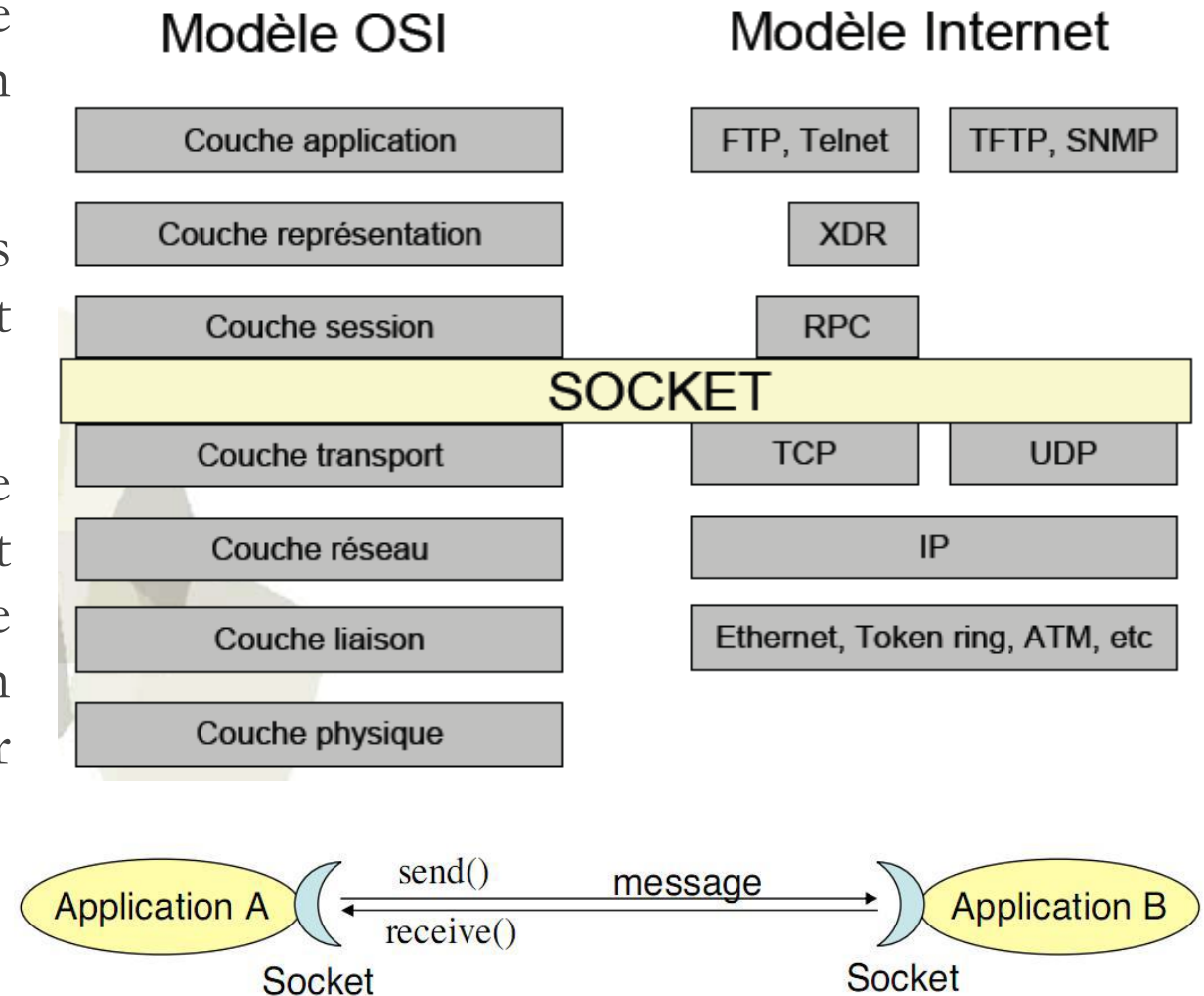
- L'architecture client serveur désigne un mode de communication entre plusieurs ordinateurs d'un réseau qui distingue un ou plusieurs postes clients du serveur.
- Chaque client peut envoyer des requêtes à un serveur.
- Un serveur peut être spécialisé en serveur d'applications, de fichiers, de terminaux, ou encore de messagerie électronique.



Clients et **serveurs** impliquent des services réseau fournis par la **couche de transport**

Les Sockets

- Une socket 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 (un serveur par exemple exploite une socket par client connecté).
- Un port est identifié par un entier (16 bits).



Mode de communication

- Il existe deux modes de communication :
 - Mode connecté : la communication entre un client et un serveur est précédée d'une connexion et suivi d'une fermeture.
 - Facilite la gestion d'état
 - Meilleurs contrôle des arrivées/départs de clients
 - Uniquement communication unicast
 - Plus lent au démarrage
 - Mode non connecté : les messages sont envoyés librement
 - Plus facile à mettre en œuvre.
 - Plus rapide au démarrage.

→ **TCP : Transmission Control Protocol**

➕ **Protocole sûr** puisqu'il garantit l'ordre d'arrivée des données envoyées : Téléphonie

➖ **Vitesse de connexion lente** : Un nombre important d'A/R.

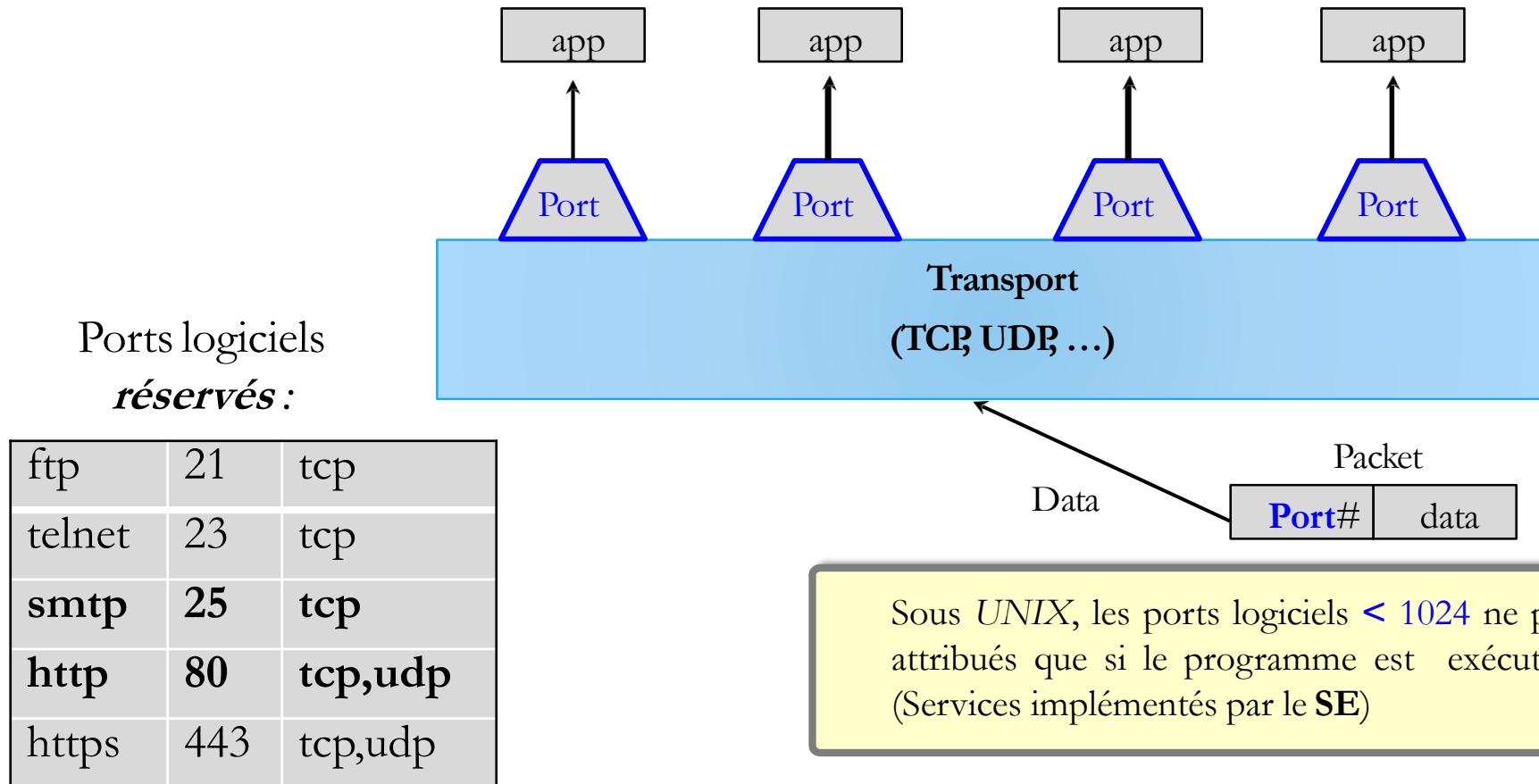
→ **UDP : User Datagram Protocol**

➕ **Rapide** (peu d'A/R) : Utilisé pour les échanges dont la perte de paquets n'est pas important (vidéocast, VoIP ...)

➖ **Peu fiable** (poste) : Il n'est pas sûr que les données arrivent à destination ni dans le bon ordre.

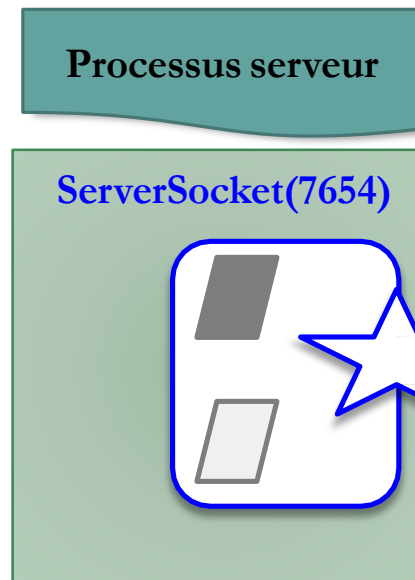
Port d'accès

- TCP ou UDP utilisent des numéros de ports pour mapper les données entrantes à un service particulier



Principe de Fonctionnement

1. le serveur enregistre son service sous un numéro de port, indiquant le nombre de clients qu'il accepte de faire buffériser à un instant T : `(new serverSocket(...))`

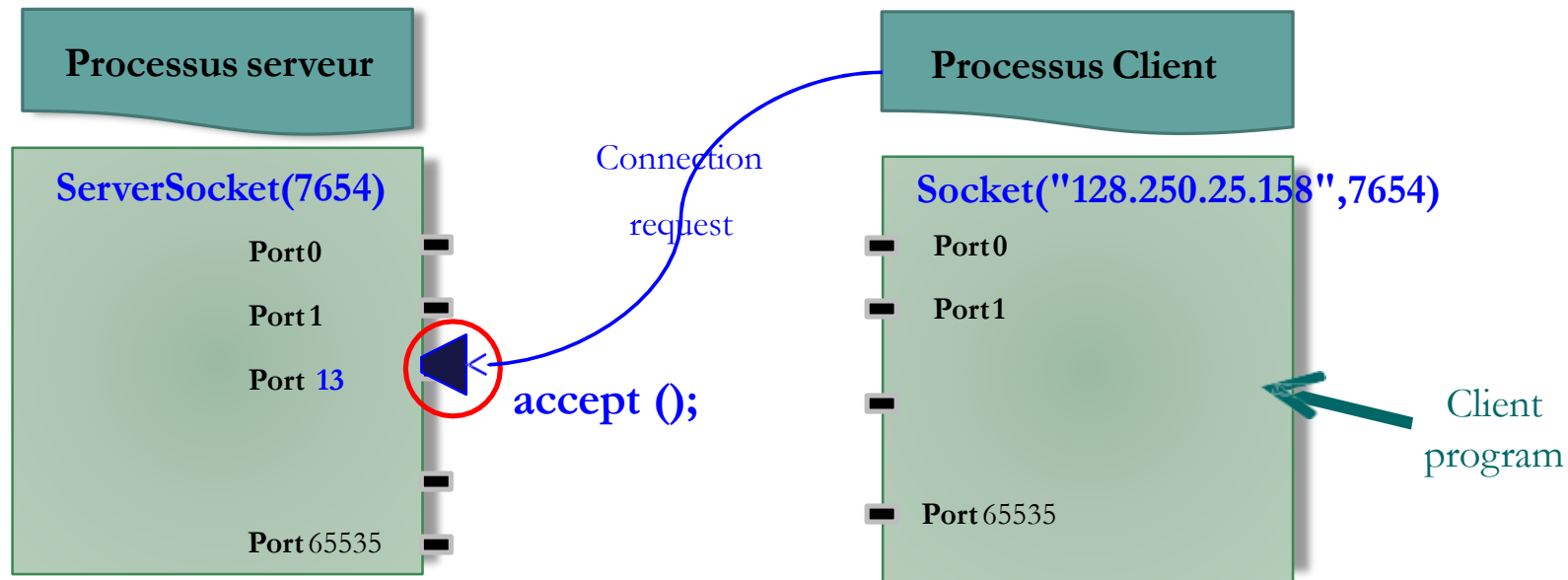


Création de la **socket** : l'extrémité de la connexion entre deux programmes en cours d'exécution sur le réseau.

Principe de Fonctionnement

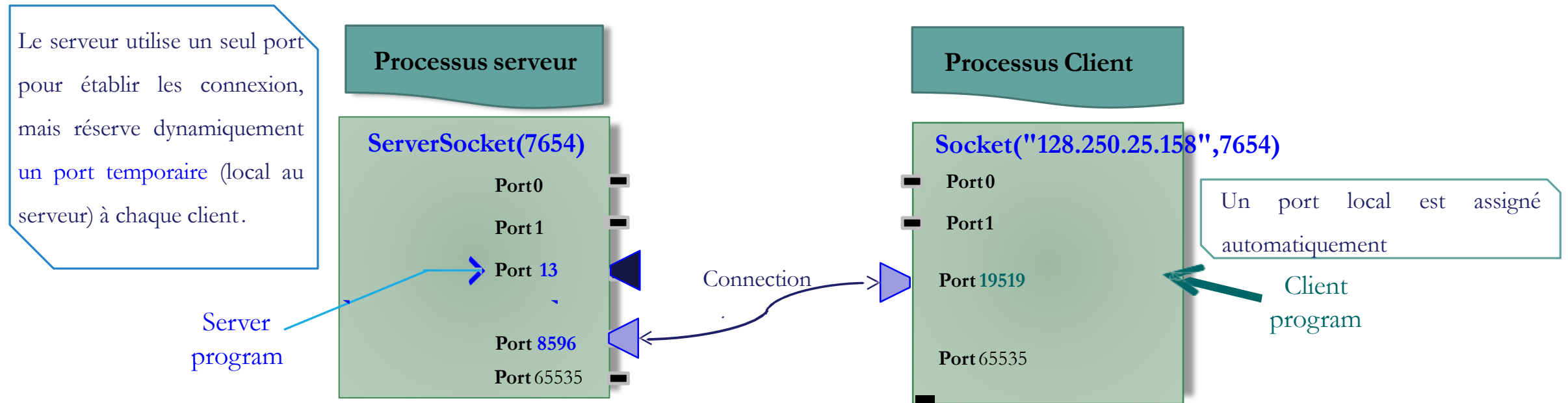
2. le serveur se met en attente d'une connexion (méthode `accept()` de son instance de `ServerSocket`)

3. 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.



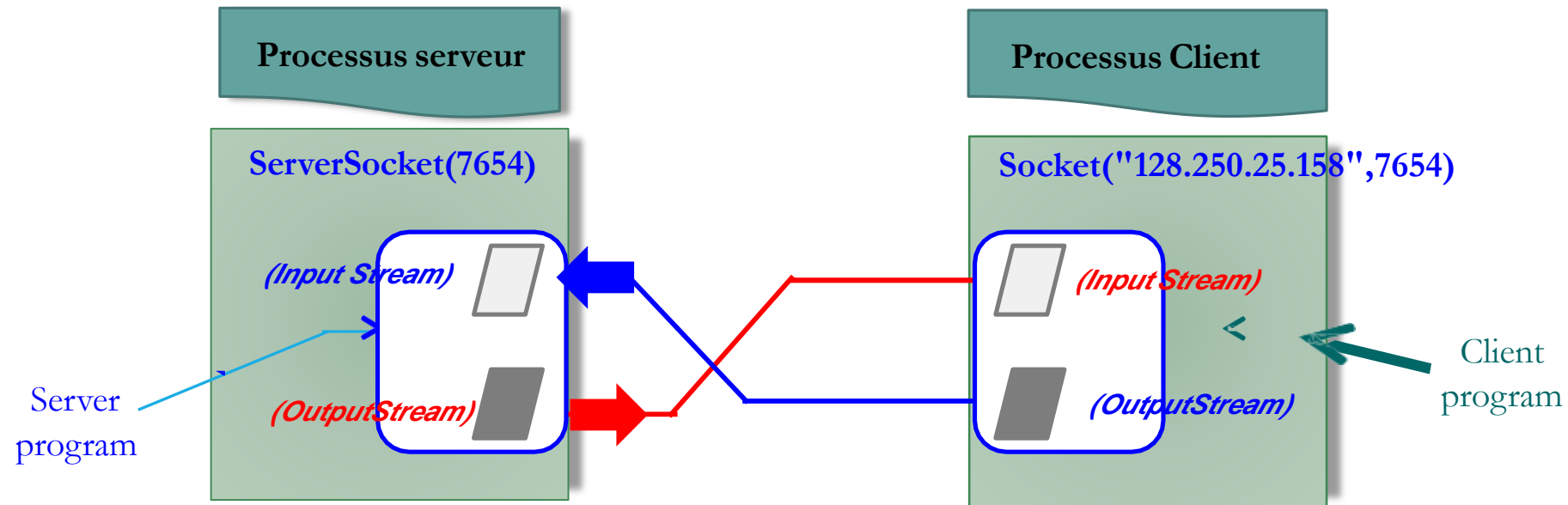
Principe de Fonctionnement

4. Le serveur accepte la connexion: une session est établie: le serveur sort de son `accept()` et récupère une Socket de communication avec le client



Principe de Fonctionnement

5. le client et le serveur peuvent alors utiliser des `InputStream` et `OutputStream` pour échanger les données : Les flux d'entrée et de sortie sont connectés aux sockets



Exemple : Couche de transport

- Ecrire un programme client qui vérifie la mise en place des sockets dans la couche de transport.

```
import java.net.*;
import java.io.*;
public class PortScanner {
    public static void main(String[] args) {
        String host = "localhost";
        try {
            InetAddress adr = InetAddress.getByName(host);
            for (int i = 1; i < 1024; i++) {
                try {
                    Socket laSocket = new Socket(adr,i);
                    System.out.println("Il y a un serveur sur le port " + i + " de " + host);
                } catch (IOException ex) { // ne doit pas y avoir de serveur sur ce port
                }
            } // end for
        } // end try
        catch (UnknownHostException e) {
            System.err.println(e);
        } // end main
    }
}
```

Ports sur écoute par
des Serveurs TCP

Il y a un serveur sur le port 21 de localhost
Il y a un serveur sur le port 22 de localhost
Il y a un serveur sur le port 23 de localhost
Il y a un serveur sur le port 25 de localhost
Il y a un serveur sur le port 37 de localhost
....

Sockets TCP

■ Coté Client :

A P I **java.net.Socket 1.0**

- **Socket(String host, int port) :**
constructs a socket to connect to the given host and port.
- **void close()**
closes the client socket

■ Coté Serveur :

A P I **java.net.ServerSocket 1.0**

- **ServerSocket(int port)**
creates a server socket that monitors a port.
- **Socket accept()**
Waits for a connection. This method blocks (that is, idles) the current thread until the connection is made. The method returns a Socket object through which the program can communicate with the connecting client.
- **void close()**
closes the server socket

■ Coté Client / Serveur :

A P I **java.net.Socket 1.0**

- **InputStream getInputStream()**
- **OutputStream getOutputStream()**
get streams to read data from the socket and write data to the socket

Sockets TCP : Client

1. Créer une socket

```
Socket client = new Socket(server, port_id);
```

2. Créer les flots d'E/S pour la communication

```
Scanner in = new Scanner(client.getInputStream());
```

```
PrintWriter out = new PrintWriter(client.getOutputStream(),true);
```

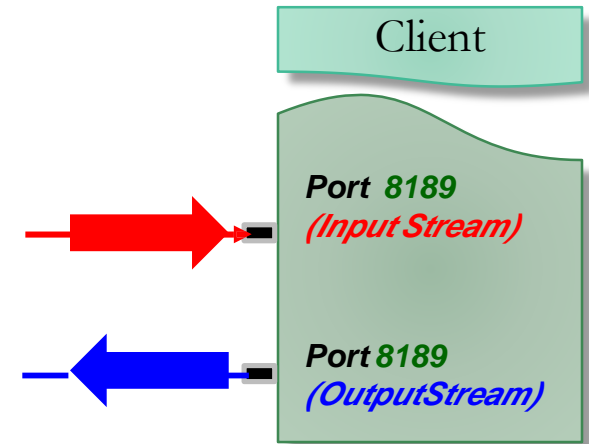
3. Communiquer avec le serveur

```
String line = in.readLine(); //Recevoir les données à partir du serveur
```

```
out.writeBytes("Hello\n"); //Envoyer les données au serveur
```

4. Fermer la socket

```
client.close();
```



Le client peut fonctionner sur tout ordinateur dans le réseau (LAN, WAN ou Internet) tant qu'il n'y a pas de pare-feu qui bloque la communication.

Client TCP: Récupérer Page Web

- Ecrire un programme client qui permet de récupérer la page d'index du serveur de l'ENSIAS et d'afficher le code source.

```
String g = "GET / HTTP/1.1\n" + "Host: www.ensias.ma\n\n";
Socket socket = new Socket("www.ensias.ma", 80);
OutputStream out = socket.getOutputStream();
out.write(g.getBytes());

InputStream in = socket.getInputStream();

byte[] b = new byte[1000]; //pour les données renvoyées

int nbBitsRecus = in.read(b); //effectivement recues

if(nbBitsRecus>0) {
    System.out.println(nbBitsRecus + " bits recus.");
    System.out.println("Recu: " + new String(b,0,nbBitsRecus));
}

socket.close();
```

g : pour récupérer la page d'index du serveur. Utilise le protocole HTTP

Client TCP: Récupérer Page Web

Résultat :

```
1000 bits recus.  
Recu: HTTP/1.1 200 OK  
Date: Wed, 09 Oct 2013 21:40:05 GMT  
Server: Apache/2.2.3 (CentOS)  
X-Powered-By: PHP/5.1.6  
...  
<meta http-equiv="Content-Type"  
content="text/html; charset=utf-8" />  
<meta content="text/html; Charset=UTF-8" http-  
equiv="Content-Type" />  
<title>Ecole Nationale SupÃ©rieure d'Informatique ...
```

Sockets TCP : Serveur

1. Créer une socket serveur

```
int port_d_ecoute = 8189;
```

```
ServerSocket listener = new ServerSocket(port_d_ecoute);
```

2. Répéter

i. Attendre une connexion avec un client

```
Socket socket_de_travail = listener.accept();
```

ii. Traiter la connexion

```
new ClasseDuTraitement(socket_travail);
```

3. Fermer la socket

```
listener.close();
```

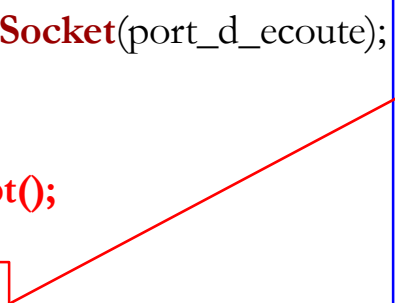
Si on ne ferme pas la socket et un autre programme utilise le même port, il sera impossible de la fermer car le port sera occupé

```
public ClasseDuTraitement(Socket s) {  
    // Créer les flots d'E/S pour la communication  
    Scanner in = new Scanner(s.getInputStream());  
  
    PrintWriter out = new PrintWriter(s.getOutputStream(), true  
        /*autoFlush*/);  
  
    // Communiquer avec le client  
  
    String line = in.readLine();    //Recevoir à partir du client  
  
    out.writeBytes("Hello\n");    // envoyer au client  
  
}
```


Serveur TCP: Exemple

- Ecrire un programme Serveur qui permet d'envoyer la date au clients

```
public class DateServer {  
    public static void main(String[] args) throws IOException {  
        int port_d_ecoute = 8189;  
        ServerSocket listener = new ServerSocket(port_d_ecoute);  
        try {  
            while (true) {  
                Socket client = listener.accept();  
                try {  
                    new DateToClient(client);  
                } finally {  
                    client.close();  
                }  
            }  
        } finally { listener.close(); }  
    }  
}
```



```
public class DateToClient {  
    public DateToClient(Socket socket) {  
        PrintWriter out = new  
        PrintWriter(socket.getOutputStream(), true);  
        out.println(new Date().toString());  
    }  
}
```

Servir plusieurs Clients : Problème

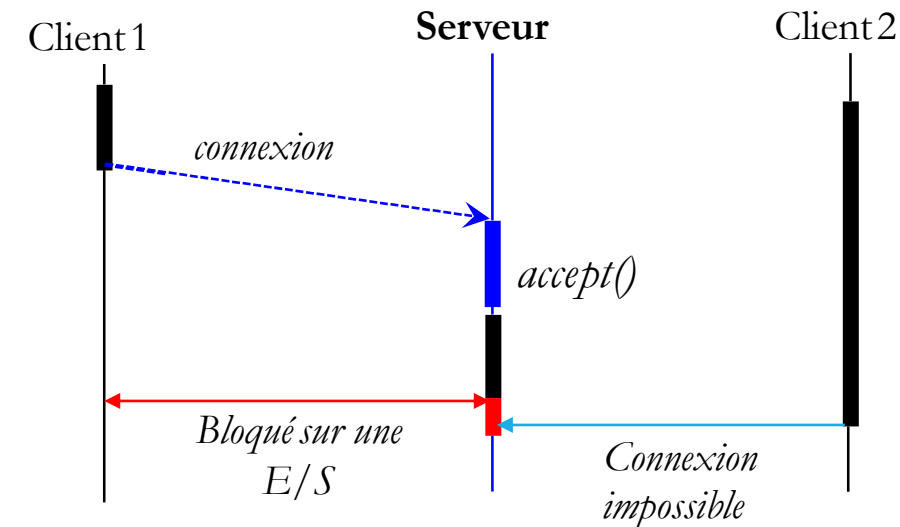
- Le serveur est bloqué jusqu'à ce qu'un client se connecte au serveur :

```
Socket s = listener.accept();
```

- Le **serveur** et le **client** seront **bloqués** si les données à partir de la socket ne sont pas disponibles :

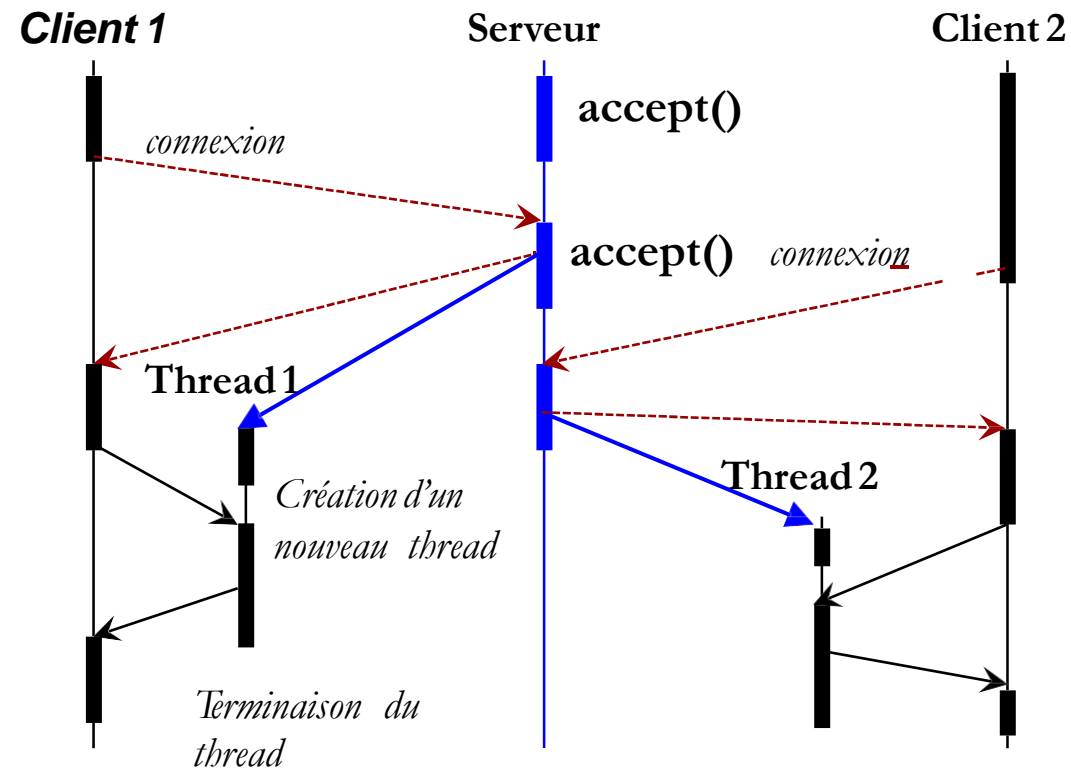
```
String line = in.readLine();
```

- Il est difficile de servir plusieurs clients : solution
 - Connexion avec le 1er client
 - Attente d'un 2eme client et devient non répondant aux requêtes du 1er : **NON**



Servir plusieurs Clients : Solution

- Placer l'appel à **accept()** dans une boucle
- Un nouveau **Thread** pour traiter chaque connexion



Servir plusieurs Clients : Solution

■ Exemple

```
public static void main(String[] args) {  
    ServerSocket listener = new ServerSocket(8189);  
    while(true){  
        Socket socket = listener.accept();  
        Runnable client= new ClientHandler(socket);  
        Thread thread = new Thread(client);  
        thread.start();  
    }  
}
```

```
class ClientHandler implements Runnable{  
    private Socket socket;  
    public ClientHandler(Socket socket) { ... }  
    // Le constructeur maintient une référence sur la socket  
    public void run() {  
        try {  
            InputStream inS = socket.getInputStream();  
            OutputStream outS = socket.getOutputStream();  
            Scanner in = new Scanner(inS);  
  
            PrintWriter out = new PrintWriter(outS,true);  
            echo(in,out); //echo sortie client par exemple  
        } finally {  
            socket.close();  
        }  
    }  
}
```

Exercice :

- On veut écrire une application client/serveur qui permet de savoir si une machine est active. Cette application utilise des sockets TCP.
 - Ecrivez le programme Serveur qui attend une requête du client sur le port 8182 et lui envoie un message (contenant l'heure locale). Le serveur doit être capable de gérer plusieurs clients simultanément.
 - Ecrivez le programme Client qui lit une adresse au clavier et envoie une requête au serveur de la machine correspondante pour savoir si elle est active. Dans l'affirmative, il affiche le message envoyé par le serveur.

Adressage

- Chercher l'adresse IP d'un Hôte:
 - Se fait automatiquement à l'aide d'un serveur DNS
 - Ne signifie pas que l'IP est accessible

A P I

java.net.InetAddress 1.0

- static InetAddress **getByName**(String host)
- static InetAddress[] **getAllByName**(String host)
constructs an InetAddress, or an array of all Internet addresses, for the given host name.

```
System.out.println("Hôte/IP : " +  
InetAddress.getByName("www.ensias.ma"));
```

 Hôte/IP : www.ensias.ma/196.200.135.4

➔ Pour Savoir si un nom d'hôte est accessible
boolean **isReachable** (int timeout)

Adressage

- Récupérer l'adresse IP d'un Hôte:

API	java.net.InetAddress 1.0
■ <code>static InetAddress getLocalHost ()</code>	constructs an InetAddress for the local host.
■ <code>byte[] getAddress()</code>	returns an array of bytes that contains the numerical address.
■ <code>String getHostAddress()</code>	returns a String with decimal numbers, separated by periods, for example, "132.163.4.102"
■ <code>String getHostName()</code>	returns a the hostname.

```
import java.net.InetAddress; ...

public class TestInetAddress {

    public static void main(String[] args) throws UnknownHostException {

        System.out.println("Hôte/IP : " + InetAddress.getLocalHost()); }}

```

Hôte/IP : HP-Pavilion /196.168.4.52

Adressage

- Chaque interface (carte réseau) peut disposer de plusieurs adresses IP
 - adresse IP local (127.0.0.1) → localhost
 - adresse IP réseau (192.168.x.x réseaux maisons)
 - adresse **IP internet**

A P I

java.net. NetworkInterface 1.0

- `Enumeration<NetworkInterface> getNetworkInterfaces()`
returns all the network interfaces.

```
public class TestNetworkInterface {  
    public static void main(String[] args) throws Exception {  
        Enumeration<NetworkInterface> lesInterfaces= NetworkInterface.getNetworkInterfaces();  
  
        while (lesInterfaces.hasMoreElements()) {    NetworkInterface interface = lesInterfaces.nextElement();  
                                                    Enumeration<InetAddress> lesAddresses = interface.getInetAddresses();  
  
            while (lesAddresses.hasMoreElements()) {  
                InetAddress address = lesAddresses.nextElement();  
                System.out.println(address.getHostAddress());    } } } }
```


Sockets UDP :

- Liaison par Datagram : (UDP)
 - **Non connecté** : pas de protocole de connexion (plus rapide)
 - **Avec perte** : l'émetteur n'est pas assuré de la délivrance
 - **Avec duplication** : un message peut arriver plus d'une fois
 - **Sans fragmentation** : les messages envoyés ne sont jamais coupés : soit un message arrive entièrement, soit il n'arrive pas.
 - **Ordre non respecté** : Communication de type courrier

- Les classes à utiliser :
 - Il faut utiliser les classes DatagramPacket et DatagramSocket
 - Ces objets sont initialisés différemment selon qu'ils sont utilisés pour envoyer ou recevoir des paquets

Sockets UDP



java.net. DatagramPacket 1.0

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

Constructs a datagram packet for sending packets of length length to the specified port number on the specified host.

- `DatagramPacket(byte[] buf, int offset, int length)`

Constructs a DatagramPacket for receiving packets of length length, specifying an offset into the buffer.

- `getPort()` : port de l'émetteur pour une réception ou port du récepteur pour une émission
- `getAddress()` : idem adresse
- `getData()` : les données reçues ou à envoyer
- `getLength()` : idem taille



java.net. DatagramSocket 1.0

- `DatagramSocket()`

Constructs a datagram socket and binds it to any available port on the local host machine.

- `DatagramSocket(int port)`

Constructs a datagram socket and binds it to the specified port on the local host machine.

- `DatagramSocket(int port, InetAddress laddr)`

Creates a datagram socket, bound to the specified local address.

- envoi : `send(DatagramPacket)`
- réception : `receive(DatagramPacket)`
- remarque : possibilité de "connecter" une socket UDP à une (@IP,port): `connect (InetAddress ,int)`
→ pas de connection réelle, juste un contrôle pour restreindre les send/receive

Sockets UDP: Envoi d'un Datagram

1. Créer un DatagramPacket en spécifiant :
 - les données à envoyer
 - leur longueur
 - la machine réceptrice et le port
2. Utiliser la méthode send(DatagramPacket) de DatagramSocket
 - pas d'arguments pour le constructeur car toutes les informations se trouvent dans le paquet envoyé

```
//Machine destinataire
InetAddress address = InetAddress.getByName(" ensias.um5.ac.ma");
static final int PORT = 4851;
//Création du message à envoyer
String s = new String (" Message à envoyer");
int longueur = s.length();
byte[] message = new byte[longueur];
s.getBytes(0,longueur,message,0);
//Initialisation du paquet avec toutes les informations
DatagramPacket paquet = new DatagramPacket (message,longueur,
address,PORT);
//Création du socket et envoi du paquet
DatagramSocket socket = new DatagramSocket();
socket.send(paquet);
```

Sockets UDP: Réception d'un Datagram

1. Créer un `DatagramSocket` : Il écoute sur le port de la machine du destinataire
2. Créer un `DatagramPacket` : le but est de recevoir les paquets envoyés par le serveur : dimensionner le buffer assez grand
3. Utiliser la méthode `receive()` de `DatagramPacket` : cette méthode est bloquante.

```
//Définir un buffer de réception
byte[] buffer = new byte[1024];
//On associe un paquet à un buffer vide pour la réception
DatagramPacket paquet = new DatagramPacket(buffer,buffer.length());
//On crée un socket pour écouter sur le port
DatagramSocket socket = new DatagramSocket(PORT);
while (true) {
    //attente de réception
    socket.receive(paquet);
    //affichage du paquet reçu
    String s = new String(buffer,0,0,paquet.getLength());
    System.out.println("Paquet reçu : + s);
}
```