



# PROGRAMMATION RÉSEAU – CHAP I : INTRODUCTION

# OBJECTIF

- Acquérir les éléments de base qui permettent d'appréhender la programmation réseau en java en particulier :
  - la Prise en charge de l'adressage en java



## RÉSEAU LOCAL (LAN):

- Les machines sont identifiées par une adresse physique MAC.
- Sur Internet les machines ne peuvent pas être identifiées par leur adresse MAC (Machines mobiles, adresse physique dépend de la technologie utilisée dans la couche d'accès au réseau, l'adressage physique n'est pas hiérarchisé).
- Java n'est pas concernée par l'adressage physique



# INTERNET

- Internet est un réseau public (différents équipements utilisant des technologies différentes: PC, Server, Mobiles, Console de jeux, ....).
- Chaque équipement connecté à Internet doit posséder un identificateur unique: adresse IP (Internet protocol).
- Cette adresse est louée (adressage dynamique) ou achetée (adressage statique)



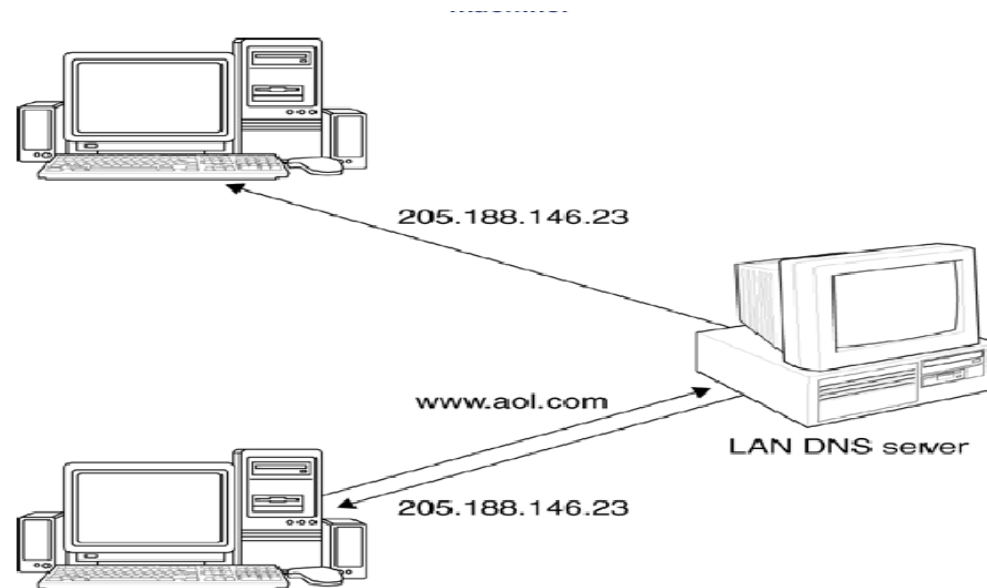
# ADRESSAGE IP

- Deux versions :
  - IP4 (ancienne mais encore très utilisée)
  - IP6 (nouvelle mais pas encore totalement imposée).
- IP4 (4 octets en notation décimale pointée) est hiérarchisée en classes A, B, C, D, E
- IP6 (16 octets en notation hexadécimale séparée par :) n'est pas hiérarchisée en classes



# DNS

- en alternative à l'adressage numérique brut IP, les machines peuvent être localisées grâce à des noms.
- Une relation biunivoque doit alors être maintenue entre les adresses IP et les noms des machines, par des machines jouant le rôle de serveur de noms comme les serveurs DNS



# JAVA ET TCP/IP

- java prend en charge la pile de protocoles TCP/IP et permet de développer des applications distribuées fonctionnant sur tout réseau TCP/IP.
- le package **java.net.\*** contient plusieurs classes et interfaces pour les fonctionnalités suivantes :
  - Adressage IP
  - Envoi et réception de paquets de type Datagramme UDP.
  - Etablissement de connections TCP.
  - Localisation et identification des ressources réseau.
  - ... etc



# ADRESSAGE IP ET JAVA

- La classe **java.net.InetAddress** joue un rôle central pour l'adressage IP et la résolution des noms d'hôtes.
- Deux classes plus spécifiques sont dérivées de cette classe:
  - **java.net.Inet4Address**,
  - **java.net.Inet6Address**.
- Un objet de cette classe contient les informations :
  - adresse IP,
  - nom de domaine si possible





# ADRESSAGE IP ET JAVA

- Cette classe ne possède pas de constructeur, les objets sont créés par des **méthodes statiques** :
- **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.
  - `InetAddress adr = InetAddress.getByName("www.isga.ma");`
  - Lorsque la recherche DNS n'aboutit pas, une exception `UnknownHostException` est levée.
- **public static InetAddress [ ] getAllByName (String nom\_hote)**
  - Cette méthode renvoie toutes les adresses Internet de la machine de nom nom\_hote.
- **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")`.



# ADRESSAGE IP ET JAVA

- Cette classe possède les méthodes suivantes :
- **public String getHostName ()**
  - Renvoie le nom de la machine hôte, ou bien l'adresse IP si la machine n'a pas de nom.
- **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).



Byte tab[ ]

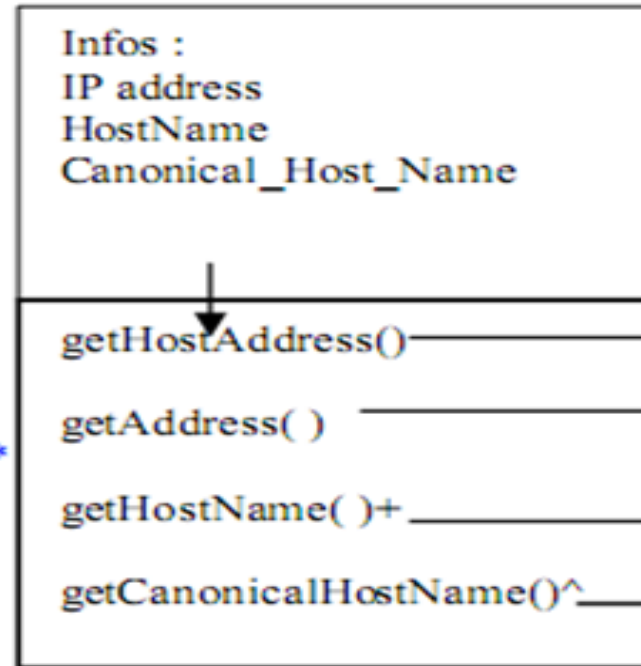
192
168
1
112

`InetAddress.getByAddress( tab ) !`  
ou  
`InetAddress.getByName( "poste1", tab ) :`

Création d'un objet InetAddress

`InetAddress.getByName( "poste1" ) *`  
ou  
`InetAddress.getByName( "www.abc.com" ) *`  
ou  
`InetAddress.getByName( "212.13.12.3" )`

InetAddress object



Consultation des infos contenues dans l'objet InetAddress

192
168
1
112



## EXAMPLE 1

```
public class localhost {  
    public static void main(String[] args) {  
        try {  
            InetAddress adresse=InetAddress.getLocalHost();  
            String hostadresse=adresse.getHostAddress();  
            String hostname=adresse.getHostName();  
            System.out.println(hostadresse);  
            System.out.println(hostname);  
        } catch (UnknownHostException ex) {  
            System.out.println("Exception : "+ex);  
        }  
    }  
}
```



- **Résultat :**
- Cas d'une machine configurée avec TCP/IP mais média déconnecté, affichage de l'adresse de boucle
  - 127.0.0.1
  - OUTZOURHIT-PC
- Cas de machine configurée avec TCP/IP et ayant une adresse
  - 192.168.1.7
  - OUTZOURHIT-PC



## EXEMPLE 2

```
Scanner sc=new Scanner(System.in);
System.out.println("saisir le nom d'hote :");
String host =sc.nextLine();
System.out.println ("Resolving " + host);
try{
    // Résoudre le nom d'hôte en objet InetAddress
    InetAddress addr = InetAddress.getByName(host);
    //Extraire de l'objet addr l'adresse IP sous
    // forme String et l'afficher
    System.out.println ("IP address : " +addr.getHostAddress() );
    //Extraire de l'objet addr le nom d'hôte et l'afficher
    System.out.println ("Hostname : " + addr.getHostName() );
}
catch (UnknownHostException ex)
{
    System.out.println ("unable to resolve hostname");
}
```



Résultat :

saisir le nom d'hôte :

www.google.com

Resolving www.google.com

IP address : 173.194.66.103

Hostname : www.google.com



## EXAMPLE 3

```
System.out.println("saisir le nom d'hote :");
Scanner sc=new Scanner(System.in);
String host=sc.nextLine();
System.out.println ("Resolving " + host);
try{
    // Résoudre le nom d'hôte en objet IP adresse, remarquez le tableau
    InetAddress[ ] addr= InetAddress.getAllByName( host );
    //addr tableau d'objets InetAddress
    for (int i = 0; i < addr.length; i++) {
        System.out.println("IP address:" +addr[i].getHostAddress());
        System.out.println ("Hostname : " + addr[i].getHostName());
    }
}
catch (UnknownHostException uhe){
    System.out.println ("Error - unable to resolve hostname" );
}
```





- Resultat:

saisir le nom d'hote :

isga.ma

Resolving isga.ma

IP address : 165.22.91.46

Hostname : isga.ma





## CHAP II : PROTOCOLE UDP

# OBJECTIF

- Appréhender la programmation réseau en java avec le protocole UDP
  - la Prise en charge du protocole UDP en java



# INTRODUCTION

- UDP : Protocole de transport sans connexion :
  - Ne garantie pas l'arrivée des paquets à destination
  - Ne garantit pas qu'ils arrivent dans le bon ordre
  - Protocole rapide mais peu fiable
- Java prend en charge le transport des paquets en utilisant le protocole UDP à travers 2 classes :
  - `java.net.DatagramPacket`
  - `java.net.DatagramSocket`



# PORTS ET SOCKETS

- Ports : utilisés par la couche de transport pour dispatcher les paquets reçus vers les applications
- Le port est un identifiant numérique sur 2 octets (valeur possible de 0 à 65535)
- Certaines valeurs sont réservées pour des services standards :
  - DNS port 53
  - HTTP port 80
  - SMTP port 25 ... etc
- Les autres valeurs sont disponibles pour les applications utilisateurs et peuvent être attribués automatiquement par le SE (au-delà de 1024)



# SOCKET

- Une socket est un mécanisme qui permet à une application d'envoyer et de recevoir des paquets sur le réseau
- Une sorte de prise logicielle ou point terminal d'une connexion
- Elle est caractérisée par :
  - Un et un seul port (auquel elle est ou sera liée)
  - Une et une seule adresse IP
  - Un buffer (tampon) pour stocker, temporairement, les données reçues ou à envoyer

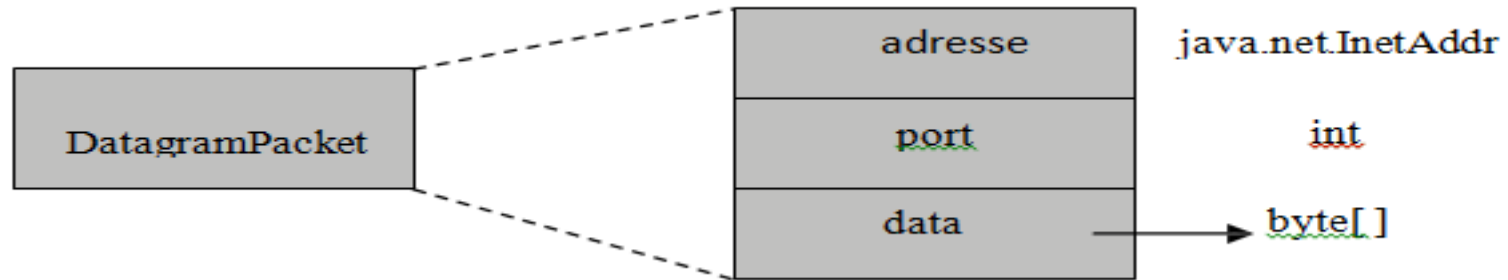


- Pour pouvoir communiquer sur le réseau, une application doit :
  - Créer une socket
  - Lier cette socket à un port (et éventuellement à une adresse IP)
  - Envoyer des paquets à travers cette socket (**send**) ou recevoir des paquets à partir de cette socket (**receive**)
- Java prend en charge le transport des paquets en utilisant le protocole UDP à travers 2 classes :
  - `java.net.DatagramPacket`
  - `java.net.DatagramSocket`



# LA CLASSE DATAGRAMPACKET

- Un objet **DatagramPacket** peut être représenté de la façon suivante :



- Si le datagramme a été construit à partir d'un paquet UDP reçu, l'adresse (adresse) et le port (port), sont ceux de l'expéditeur du paquet (machine distante).
- Si le datagramme est utilisé pour envoyer un paquet UDP, l'adresse (adresse) et le port (port), sont ceux du récepteur du paquet (machine distante).



# DATAGRAMPACKET – LES CONSTRUCTEURS

- **DatagramPacket(byte[] buf, int length)**
  - Construit un *DatagramPacket* pour recevoir un paquet de longueur *length*.
- **DatagramPacket(byte[] buf, int offset, int length)**
  - Construit un *DatagramPacket* pour recevoir un paquet de longueur *length*, en spécifiant un offset dans le *buf* (offset=décalage par rapport au début) .
- **DatagramPacket(byte[] buf, int length, InetAddress address, int port)**
  - Construit un *DatagramPacket* pour envoyer des paquets de longueur *length* vers le *port* de la destination *address*.



# DATAGRAMPACKET – LES CONSTRUCTEURS

- **DatagramPacket(byte[] buf, int offset, int length, InetAddress address, int port)**
  - Construit un *DatagramPacket* pour envoyer des paquets de longueur *length* vers le *port* de la destination *address*, en spécifiant l'*offset* dans le *buf*.



## DATAGRAMPACKET – MÉTHODES

- **InetAddress getAddress ()** : retourne l'adresse du serveur
- **byte[] getData()** : retourne les données contenues dans le paquet
- **int getPort ()** : retourne le port
- **int getLength ()** : retourne la taille des données contenues dans le paquet
- **setData(byte[])** : Mettre à jour les données contenues dans le paquet
- Le format des données échangées est un tableau d'octets, il faut donc correctement initialiser la propriété `length` qui représente la taille du tableau pour un paquet émis et utiliser cette propriété pour lire les données dans un paquet reçu.



# LA CLASSE `DatagramSocket` - CONSTRUCTEURS

## ○ `DatagramSocket()`

- Construit une *DatagramSocket* et lui attribue un port disponible.
- Utile pour un *DatagramSocket* utilisé par un client pour envoyer des paquets UDP (requêtes) / recevoir des paquets UDP (réponses).

## ○ `DatagramSocket(int port) throws SocketException`

- Construit une *DatagramSocket* et lui attribue le port fournit, sur la machine locale
- Utile pour un *DatagramSocket* utilisé par un Serveur pour écouter sur un port et y recevoir (requêtes) /envoyer des (réponses) UDP.



# LA CLASSE DATAGRAMSOCKET - CONSTRUCTEURS

- **DatagramSocket(int port, InetAddress addr) throws SocketException**
  - Construit une *DatagramSocket* et lui attribue le port et l'adresse fournis, sur la machine locale
  - Utile pour un *DatagramSocket* utilisé par un Serveur (ayant plusieurs adresses) pour écouter sur un port et adresse et y recevoir des requêtes UDP.



# LA CLASSE DATAGRAMSOCKET - METHODES

- **void receive(DatagramPacket p)**
  - recevoir un paquet dans le datagramPacket p, à partir de la socket.
- **void send(DatagramPacket p)**
  - envoyer le datagramPacket p via la socket.
- **void close()**
  - ferme cette socket et libère les ressources système correspondantes.
- **void connect(InetAddress address, int port)**
  - Connecte la socket à une adresse et port distants spécifiques.



# LA CLASSE DATAGRAMSOCKET - METHODES

- **void disconnect( )**
  - Déconnecte la socket.
- **InetAddress getAddress( )**
  - Retourne l'adresse distante vers laquelle la socket est connectée.
- **InetAddress getLocalAddress( )**
  - Donne l'adresse locale à laquelle la socket est liée.
- **int getLocalPort( )**
  - Donne le port local au quel la socket est liée.
- **int getPort()**
  - Donne le port distant vers lequel est connecté cette socket, -1 si la socket n'est pas connectée.



# UN SERVEUR UDP

- Un serveur est une application qui sert d'autres applications (clients) en répondant à leurs requêtes.
- Le client initie toujours la communication.
- Pour pouvoir recevoir les requêtes de ses clients, un serveur UDP doit:
  - Créer une *DatagramSocket* et la relier à un port (libre et connu des clients).
  - Créer un *DatagramPacket* et l'utiliser comme conteneur pour y enregistrer les paquets reçus à partir de la *DatagramSocket*.
  - Se mettre à l'écoute sur le port choisi, en attendant l'arrivée d'un paquet, le charger dans le conteneur *DatagramPacket*.
  - Extraire les informations du *DatagramPacket* (IPAddress, Port de l'émetteur et Data reçues).
  - Traiter les données reçues.





# EXEMPLE DE SERVEUR

```
boolean stop=false;
try {
    DatagramSocket sock=new DatagramSocket(8000);
    DatagramPacket pack=null;
    while(!stop){
        Thread.sleep(100);
        pack=new DatagramPacket(new byte[512], 512);
        sock.receive(pack);
        byte[] data=pack.getData();
        String s=new String(data).trim();
        System.out.println("vous avez dis : "+s);
        if(s.equals("fin"))    stop=true;
    }
    sock.close();
} catch (Exception e) {
}
```



# UN CLIENT UDP

- Le client est une application qui envoie des requêtes à un serveur; il initie l'échange.
- Pour pouvoir envoyer ses requêtes vers un serveur UDP, le client doit:
  - Créer une *DatagramSocket*, la spécification du port UDP est ici facultative.
  - Créer un *DatagramPacket* et l'initialiser avec les Données à envoyer, ainsi que l'adresse IP et le port du serveur.
  - Envoyer le paquet à travers la *DatagramSocket* en lui passant le *Datagrampacket* initialisé.
  - Répéter si nécessaire les deux étapes précédentes pour d'autres paquets.



## EXEMPLE DE CLIENT

```
Scanner sc=new Scanner(System.in);
DatagramSocket sock=new DatagramSocket();
boolean stop=false;
while(!stop){
    String s=sc.nextLine();
    byte[] data=s.getBytes();
    if(s.equals("fin")) stop=true;
    DatagramPacket pack=new DatagramPacket(data, data.length);
    pack.setAddress(InetAddress.getByName("localhost"));
    pack.setPort(8000);
    sock.send(pack);
}
    sock.close();
} catch (Exception ex) {
}
```



# DATAGRAMPACKET – LES CONSTRUCTEURS

- **DatagramPacket(byte[] buf, int offset, int length, InetAddress address, int port)**
  - Construit un *DatagramPacket* pour envoyer des paquets de longueur *length* vers le *port* de la destination *address*, en spécifiant l'*offset* dans le *buf*.
- **DatagramPacket(byte[] buf, int length, SocketAddress address)**
  - Construit un *DatagramPacket* pour envoyer des paquets de longueur *length*, vers la destination pointée par la socket *address*.
- **DatagramPacket(byte[] buf, int offset, int length, SocketAddress address)**
  - Construit un *DatagramPacket* pour envoyer des paquets de longueur *length*, en spécifiant l'*offset* dans le *Buf*, vers la destination pointée par la socket *address*.



## CHAP III : PROGRAMMATION TCP

# OBJECTIF

- Appréhender la programmation réseau en java avec le protocole TCP
  - la Prise en charge du protocole TCP en java



# TRANSFERT CONTROL PROTOCOL TCP

- TCP garantie l'arrivée des données vers la destination et dans l'ordre d'envoi
- TCP utilise pour cela :
  - le mécanisme d'accusé de réception
  - l'étiquetage des données
  - La retransmission de données en cas de non arrivée d'accusés de réception
- Toutefois, en cas d'anomalies graves sur le réseau (le destinataire a été réinitialisé, le réseau est en panne...), et au bout d'un certain nombre de tentatives non fructueuses, le TCP déclare l'impossibilité de transmettre les données



- La programmation réseau utilisant UDP : manipulation de paquet à travers des sockets
- la programmation réseau utilisant TCP : utilisation des flux
- Le programmeur aura moins de travail à faire en utilisant TCP : les données qu'il manipule en flux sont converties en paquets et gérées automatiquement par TCP



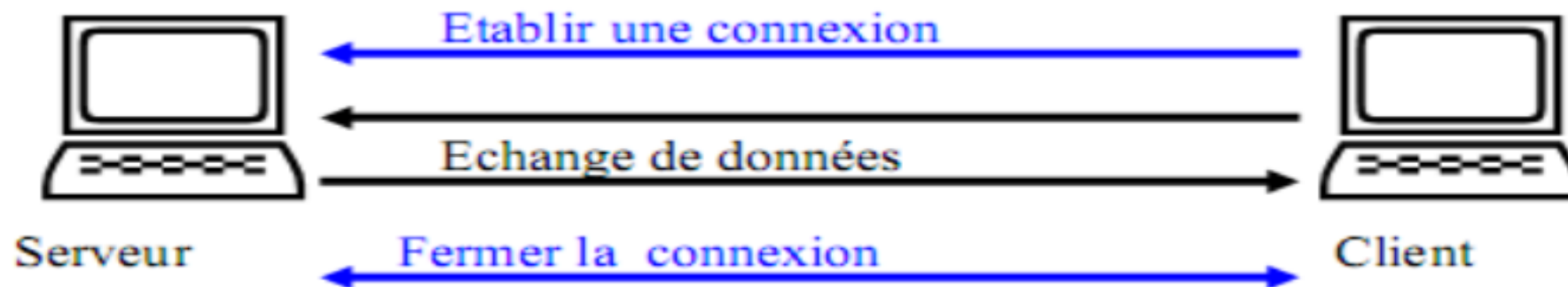


- UDP utilise de manière symétrique la même classe de socket au niveau client et serveur **DatagramSocket**
- TCP utilise deux classes de socket différentes, une pour le serveur (**ServerSocket**) et une pour le client (**Socket**).
- UDP utilise les sockets en mode non connectées : une socket peut échanger des paquets avec différentes destinations
- TCP utilise les sockets en mode connectées, une socket est forcément connectée à une seule destination lors de sa création et ne peut échanger des données qu'avec cette seule destination



# SOCKETS POUR TCP


- En TCP, l'échange de données ne peut commencer qu'après avoir établi une connexion, et doit se terminer par une fin de connexion, comme le résume la figure suivante:

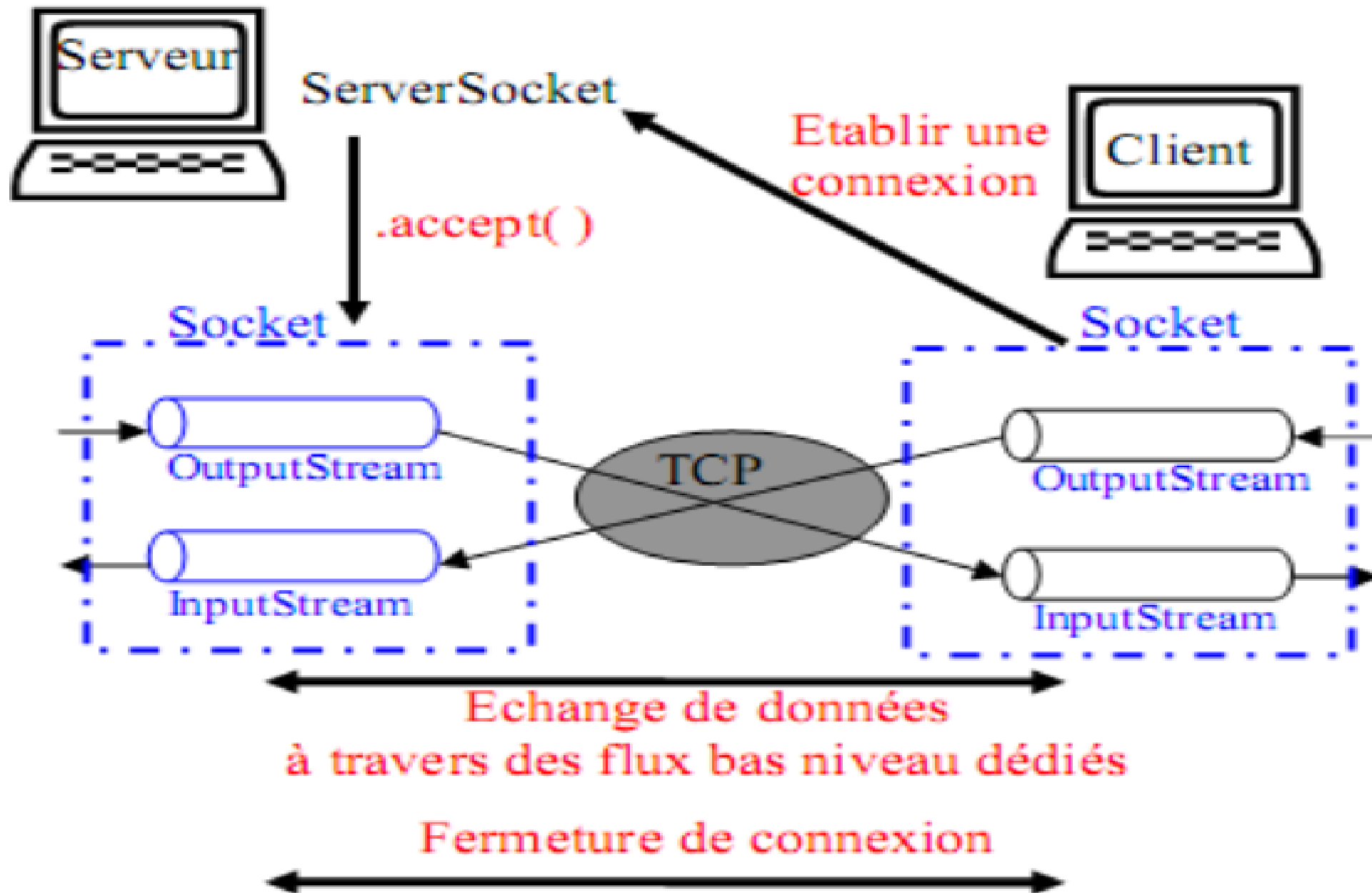


## Côté serveur:

- Le serveur utilise une socket de type **ServerSocket**, liée à un port local, qui permet:
  - d'accepter les connexions.
  - de créer une socket de type **Socket** connectée au client pour l'échange de données, ayant deux flux d'entrée/sortie.

## Côté client:

- Le client utilise une socket de type **Socket**, qui permet :
    - d'établir une connexion à un serveur distant.
    - envoyer et recevoir des données vers le serveur à travers deux flux (entrée/sortie) dédiés à la socket.
    - de terminer la connexion.
- 



# CRÉATION D'UN SERVEUR TCP (MONOTÂCHE)

- Le serveur TCP (monotâche) fonctionne de la manière suivante:
  - Crée une socket de type **ServerSocket** liée à un port.
  - Attend la demande de connexion d'un client et l'accepte.
  - Pour chaque demande de connexion acceptée une socket de type **Socket** connectée au client est créée.
  - Le serveur peut échanger avec le client via les deux flux standard dédiés à la socket créée.
  - Le serveur peut décider de fermer la connexion au client.




# SERVEUR TCP

```
// tentative de Création d'une ServerSocket liée au port 9000
ServerSocket sockServer = new ServerSocket ( 9000 );

// Attente d'une demande de connexion
Socket sock = sockServer.accept( ); //acceptation d'une demande et
// création d'une socket pour l'échange de donnée avec le
client

// Récupérer le flux d'entrée de la socket
InputStream input=sock.getInputStream();
// Récupérer le flux de sortie de la socket
OutputStream output=sock.getOutputStream();
/*
    Echange et traitement de données à travers ces deux flux
*/
sock.close(); // fermeture de la connexion au client
```



# CRÉATION D'UN CLIENT TCP

- Le client TCP fonctionne de la manière suivante:
  - Crée une socket de type **Socket** connectée à un serveur (IP, Port)
  - Une fois la socket créée il peut échanger avec le serveur via les deux flux standard dédiés à la socket créée
  - Le client peut décider de fermer la connexion au serveur




# CLIENT TCP

```
// Création d'une socket connectée à un port sur un serveur
Socket sock = new Socket ( "localhost", 9000);

// Récupérer le flux d'entrée de la socket
InputStream input=sock.getInputStream();

// Récupérer le flux de sortie de la socket
OutputStream output=sock.getOutputStream();
/*
    Echange de donnée avec le client à travers ces deux flux
*/
//fermeture de la socket
sock.close( );
```





# LA CLASSE SERVERSOCKET

- Les méthodes de la classe **ServerSocket**:
- **public Socket accept()**
  - permet l'acceptation d'une connexion demandée par un client,
  - Cette méthode est bloquante mais l'attente peut être limitée dans le temps par la méthode:  
**public void setSoTimeout(int timeout) throws SocketException**
- **public InetAddress getInetAddress()**
  - fournit l'adresse IP de la socket d'écoute
- **public int getLocalPort()**
  - fournit le port de la socket d'écoute
- **public void close()**
  - ferme la connexion et libère les ressources du système associées à la socket



# LA CLASSE SOCKET – CONSTRUCTEURS

- Coté serveur, on utilise la méthode **accept** de la classe **ServerSocket**
- Coté client, on utilise les constructeurs suivants :
  - **Socket(String host, int port) :**
  - **Socket(InetAddress address, int port) :**
    - Ces 2 constructeurs créent une socket connectée à l'adresse et au port spécifiés
  - **Socket(String host, int port , InetAddress localAddress, int localPort )**
  - **Socket(InetAddress address, int port , InetAddress localAddress, int localPort )**
    - Les 2 derniers constructeurs permettent en plus de fixer l'adresse IP et le numéro de port utilisés coté client (au lieu d'utiliser un port disponible quelconque)

# LA CLASSE SOCKET – MÉTHODES

- La communication effective sur une connexion par socket utilise des flux de données
  - **java.io.OutputStream** (flux d'écriture)
  - **java.io.InputStream** (flux de lecture)
- Ces flux sont obtenus à l'aide des méthodes suivantes :
  - **public InputStream getInputStream () throws IOException**
  - **public OutputStream getOutputStream () throws IOException**
- Les flux obtenus sont des flux de base (flux primaires), on peut leur associer des flux de traitement
  - **DataInputStream et DataOutputStream**
  - **PrintStream et Scanner**
  - **BufferedInputStream et BufferedOutputStream**



- Une opération de lecture sur ces flux est bloquante tant que les données ne sont pas disponibles.
- Cependant, il est possible de fixer un délai d'attente de données en utilisant la méthode :

**public void setSoTimeout(int timeout) throws SocketException**



# LES MÉTHODES DE LA CLASSE **Socket**

- La classe Socket possède les méthodes suivantes :
  - **public InetAddress getAddress()**
    - fournit l'adresse IP distante
  - **public InetAddress getLocalInetAddress()**
    - fournit l'adresse IP locale
  - **public int getPort()**
    - fournit le port distant
  - **public int getLocalPort()**
    - fournit le port local
  - **public void close()**
    - ferme la connexion et libère les ressources du système associées à la socket
- 