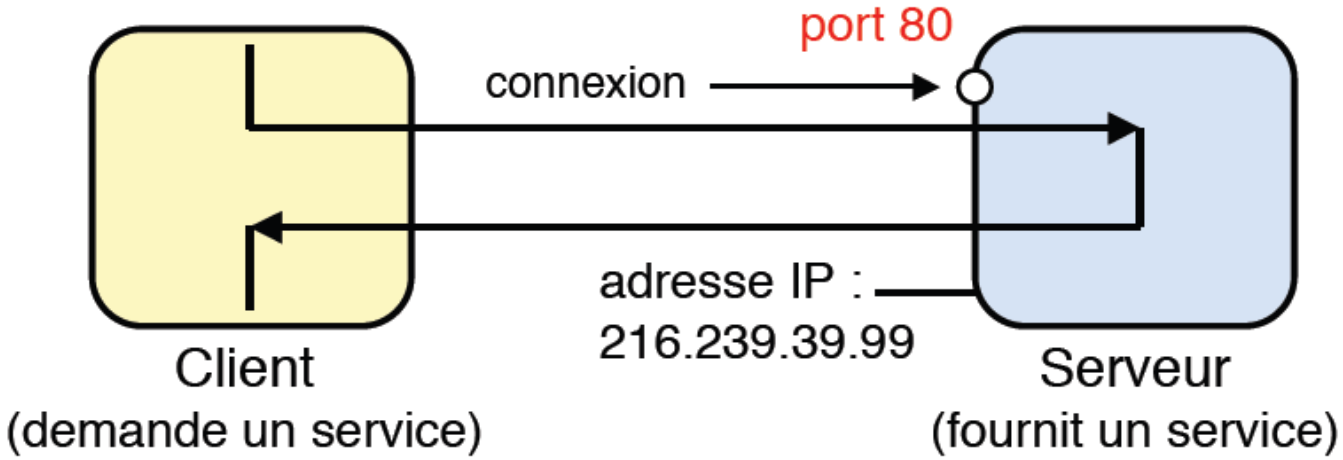
# Communication par sockets

Souad Hadjres

# Introduction

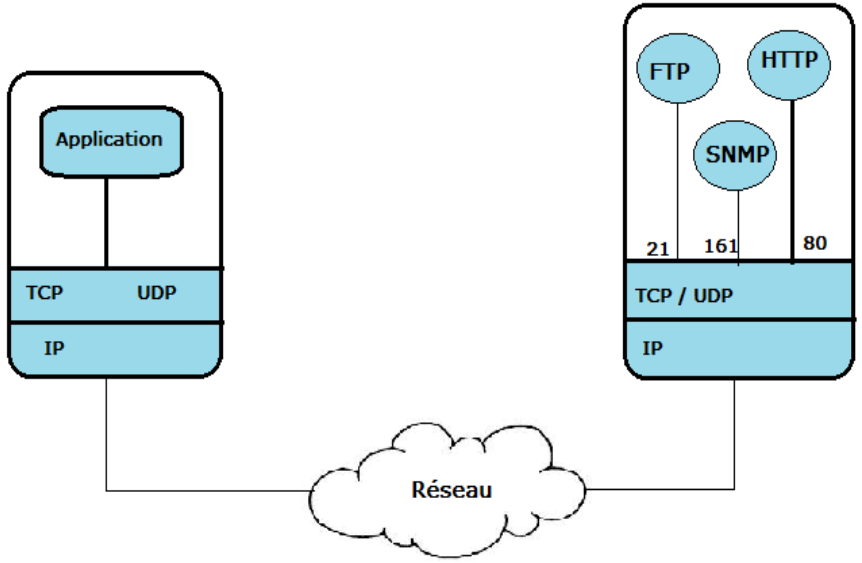
- Un service est souvent désigné par un nom symbolique (par exemple mail, http://..., telnet, etc.).
- Ce nom doit être converti en une adresse interprétable par les protocoles du réseau.
- La conversion d'un nom symbolique (par ex. http://www.google.com) en une adresse IP (216.239.39.99) est à la charge du service DNS



# Introduction

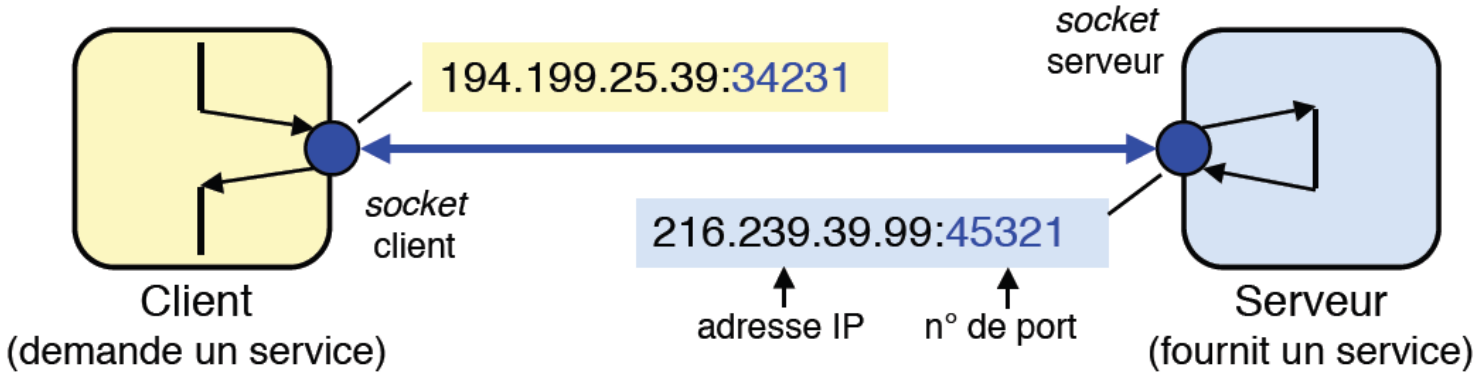
- l'adresse IP du serveur ne suffit pas car le serveur peut comporter différents services; il faut préciser le service demandé au moyen d'un numéro de port, qui permet d'atteindre un processus particulier sur la machine serveur.
- Un numéro de port comprend 16 bits (0 à 65 535). Les numéros de 0 à 1023 sont réservés, par convention, à des services spécifiques.

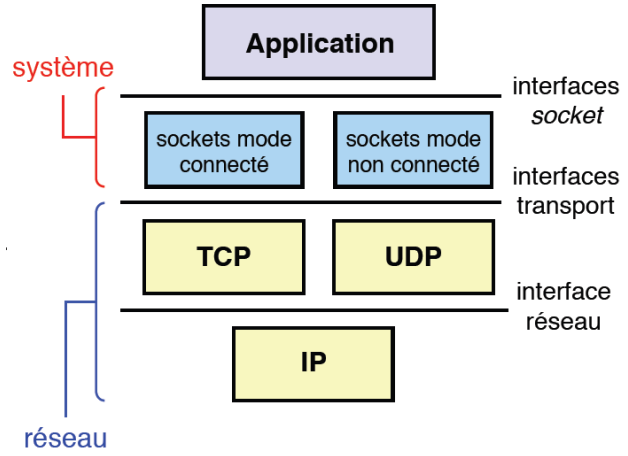
(Exemples; 7: echo, 23 : telnet, 80 : serveur web, 25: mail)



# Qu'est ce qu'un socket

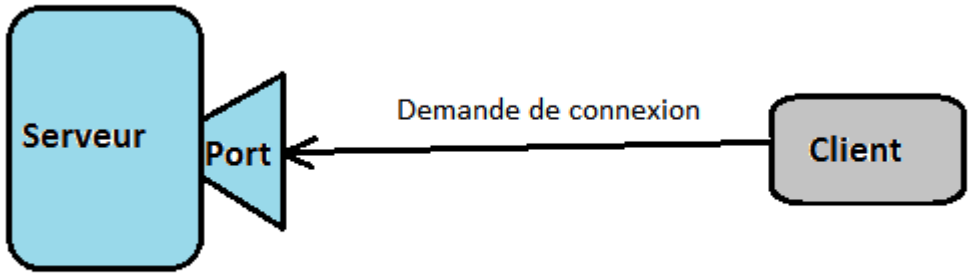
- Un *socket* est simplement un moyen de désigner l'extrémité d'une connexion, côté émetteur ou récepteur ( une adresse IP), en l'associant à un port.
- Pour programmer une application client-serveur, il est commode d'utiliser les *sockets*. Les *sockets* fournissent une interface qui permet d'utiliser facilement les protocoles de transport TCP et UDP
- Lorsque vous choisissez un numéro de port pour votre serveur, sélectionnez en un qui est supérieur à 1023



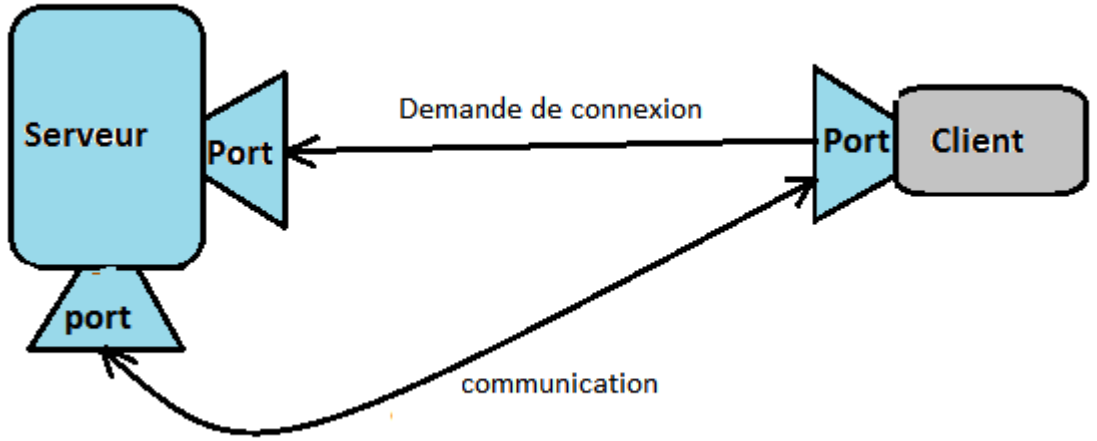
# Communication par sockets

- Un serveur fournit un service à des clients. Il doit donc attendre une demande, puis la traiter.
- Le serveur doit avoir pour chaque service, un socket lié à un port spécifique associé à ce service.
- Le serveur écoute sur le socket et attend qu'un client fasse une demande de connexion.



# Communication par sockets

- Le serveur accepte la demande de connexion entrante.
- Le serveur crée un nouveau socket liée a un numéro de port différent de celui sur lequel il a reçu la demande.



# Communication par sockets

Deux principaux protocoles de communication peuvent être utilisés pour la programmation des sockets

1. Le  protocole TCP: Communication par flux (de données) continu (ou stream) en utilisant des sockets de flux (stream sockets).
2. Le protocole UDP: Communication par datagramme en utilisant des sockets datagrammes (datagram sockets)

# Communication par sockets

- Pour les sockets de flux: Une connexion est établie via des *sockets* entre un processus client et un processus serveur, et ensuite les messages sont échangés entre eux sous forme de flux d'octets, à l’aide de primitives telles que celles utilisées par les gestionnaires de fichiers (*read*, *write...)*.
- Pour les sockets de datagrammes: aucune liaison n'est établie. Les messages sont échangés individuellement sous forme de paquets (datagrammes).

# Implémentation du client et du serveur
# Les principales classes

Le package java.net fourni les classes nécessaires pour la manipulation des sockets. Entre autres:

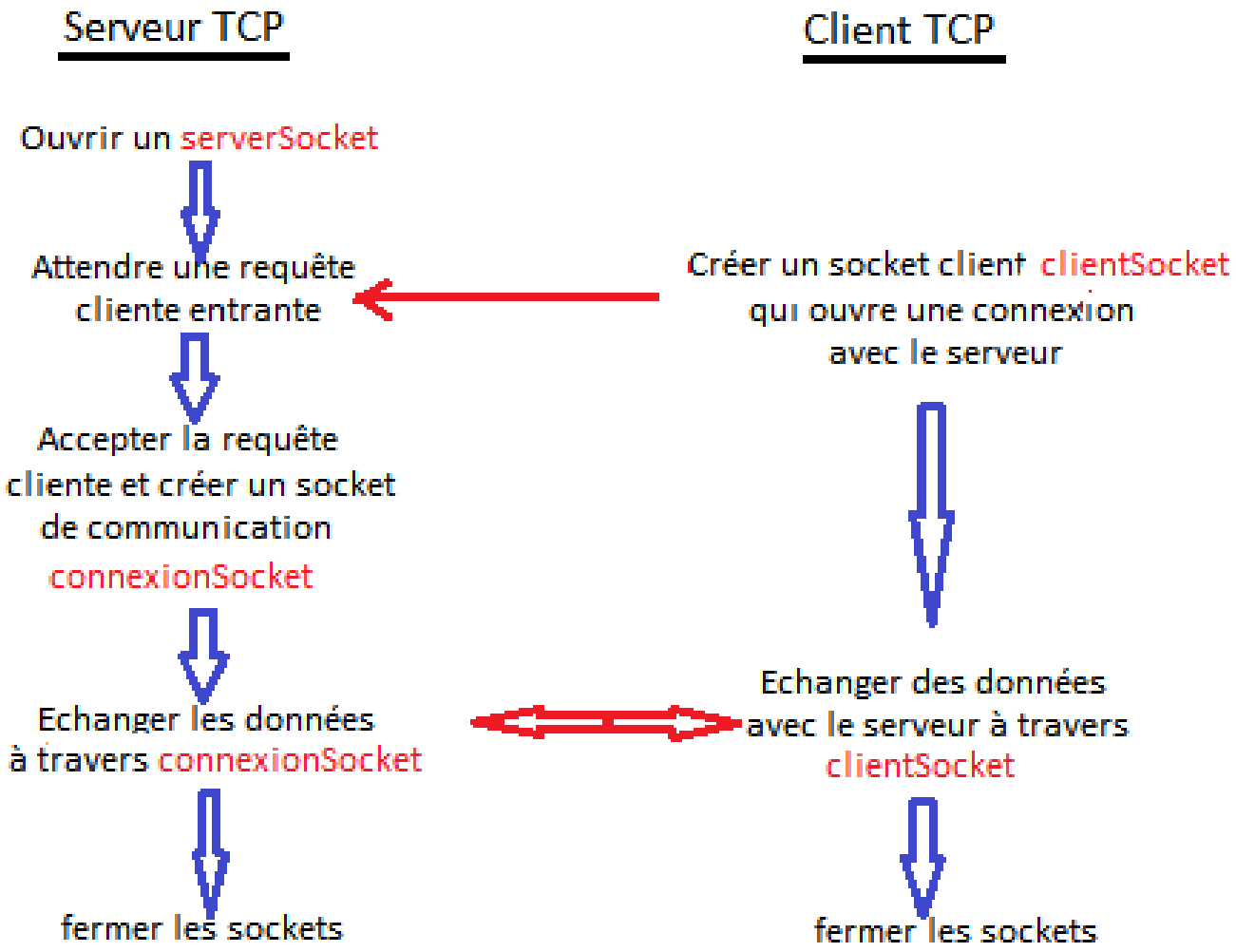
Pour la création des sockets:

- Socket – pour l'implémentation d'un client TCP
- ServerSocket – pour l'implémentation d'un serveur TCP
- DatagramSocket – pour l'implémentation à la fois d'un client et d'un serveur UDP

Pour l'échange de données:

- TCP: InputStream and OutputStream
- UDP: DatagramPacket

# Implémentation d'un client/serveur TCP



# Implémentation d'un serveur TCP

1. Créer le socket du serveur lié à un port donné:

```
ServerSocket serveur;
try {
   serveur = new ServerSocket(portNumber);
} catch (IOException e) { System.out.println(e); }
```

2. Mettre le socket à l'écoute de demande de connexion avec la méthode bloquante *accept():*

```
try {
   Socket connexionSocket = serveur.accept();
} catch (IOException e) { .. }
```

# Implémentation d'un serveur TCP

3. Échanger des données avec le client à travers le socket *connexionSocket*:

```
String data = "Hello from server";
try {
  OutputStream out = connexionSocket.getOutputStream();
  out.write(data.getBytes() );
} catch (IOException e) {…}
```

(le message *Hello from server* est envoyé au client)

# Implémentation d’un serveur TCP

4. Fermer les sockets:

```
try {
   connexionSocket.close(); // Close the socket. We are
                            // done with this client!
   serveur.close(); // close the server socket
} catch (IOException e) { System.out.println(e); }
```

# Implémentation d’un client TCP

1. Ouverture d’un socket client:

```
try {
   Socket clientSocket = new Socket(serverIP/Name, serverPort);
} catch (IOException e) { System.out.println(e); }
```

# Implémentation d'un client TCP

2. Échanger des données avec le serveur à travers le socket *clientSocket*:

```
int MAXLENGTH=256;
byte[ ] buff = new byte[MAXLENGTH];
try {
  InputStream in =  clientSocket .getInputStream();
  in.read(buff);
} catch (IOException e) {System.out.println(e);
```

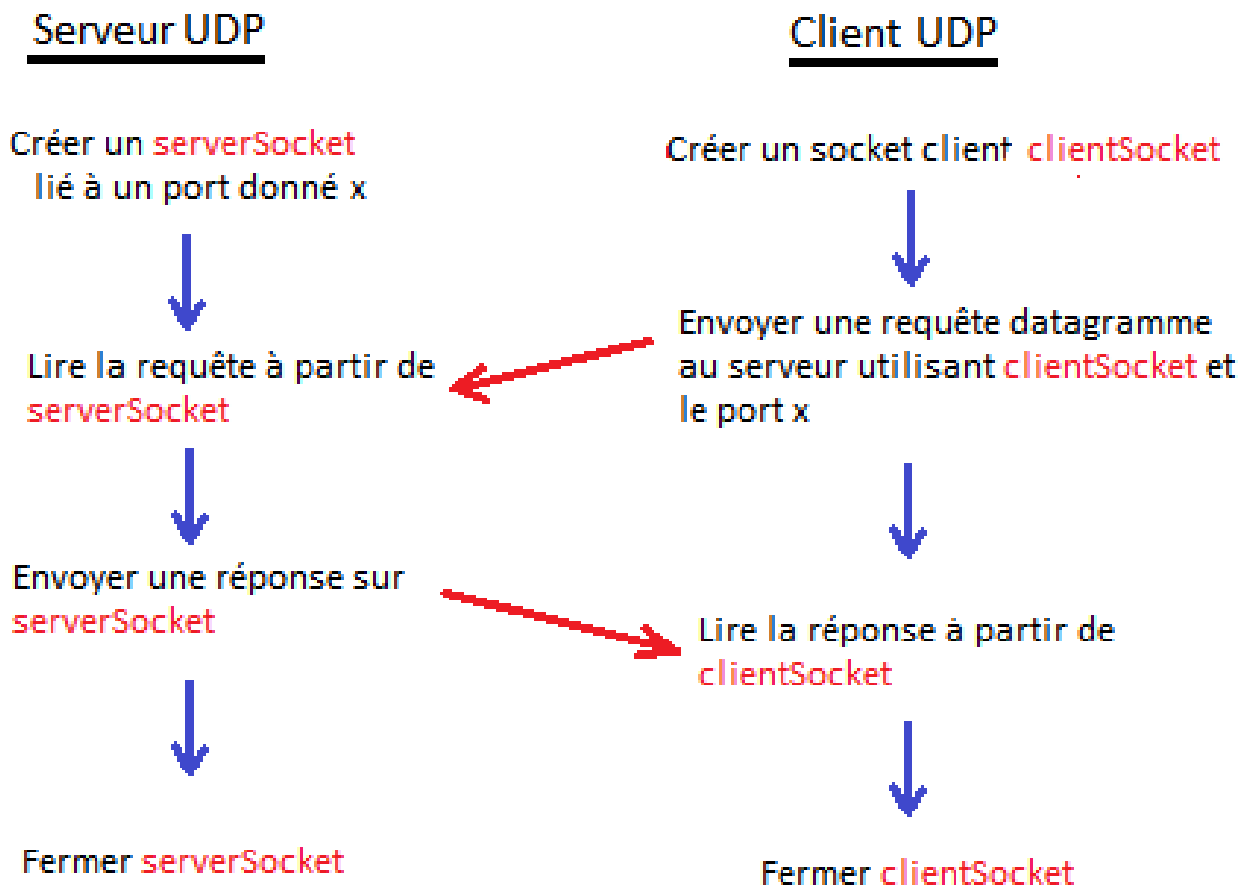
(le message *Hello from server* est récupéré par le client dans le tableau *buff*)

# Implémentation d’un client TCP

3. Fermer les sockets:

```
try {
   clientSocket.close();
} catch (IOException e) { System.out.println(e); }
```

# Implémentation d'un client/serveur UDP



# Implémentation d'un serveur UDP

1. Créer *serverSocket* un socket serveur lié à un port donné:

```
DatagramSocket serveur;
try {
  serveur = new DatagramSocket (portNumber);
} catch (IOException e) { System.out.println(e); }
```

2. Lire le datagramme requête à partir de *serverSocket*:

```
byte[ ] buff = new byte[PACKETLENGTH];
try {
  DatagramPacket rcvPacket = new DatagramPacket (buff, buff.length);
  serveur.receive(rcvPacket );
} catch (IOException e) { .. }
```

# Implémentation d'un serveur UDP

3. Obtenir l'adresse IP et le port du client puis envoyer la réponse sur *serverSocket*:

```
InetAddress clientIP = rcvPacket.getAddress();
int clientPort = rcvPacket.getPort();
String data = "Hello from server";
try {
  DatagramPacket sendPacket = new
    DatagramPacket(sendData,sendData.length, clientIP, clientPort );
  serveur.send(sendPacket);
} catch (IOException e) {…}
```

4. Fermer le socket *serveur*.

# Implémentation d'un client UDP

1. Créer un socket client clientSocket:

```
try {
  DatagramSocket clientSocket = new DatagramSocket();
} catch (IOException e) { System.out.println(e); }
```

2. Envoyer un datagramme (requête) au serveur:

```
int PACKETLENGTH= 256;
byte[ ] data = new byte[PACKETLENGTH];
try {
  DatagramPacket packet = new
                    DatagramPacket(data,data.length,  serverIP ,  serverPort );
  clientSocket .send(packet);
} catch (IOException e) {System.out.println(e); }
```

# Implémentation d'un client UDP

3. Récupérer la réponse du serveur:

```
byte[ ] rcvData = new byte[PACKETLENGTH];
Try {
  DatagramPacket receivePacket = new
                        DatagramPacket (rcvData, rcvData.length);
  clientSocket.receive(rcvPacket);
  String rcvString = new String(rcvPacket.getData());
  System.out.println("The received packet is: "+rcvString);
} catch (IOException e) {System.out.println(e); }
```

4. Fermer le socket:

```
try {
clientSocket.close(); // Close the socket
} catch (IOException e) {…}
```

# La classe InetAddress

- Les objets de cette classe modélisent les adresses IP. Ils sont utilisés par exemple comme argument des constructeurs de la classe Socket.
- Cette classe donne aussi des renseignements sur l'adresse IP à l'aide de méthodes statiques. Par exemple pour obtenir l'adresse IP de la machine locale on utilise : InetAddress.getLocalHost()

Exemple: pour lancer un client et un serveur sur la même machine locale. On écrit alors :

```
Socket serveur = new Socket(InetAddress.getLocalHost(), 4567);
```

# La classe InetAddress

Autres méthodes de la classe InetAddress:

- public static synchronized InetAddress getByName(String nom_hote) throws UnknownHostExeception
- public static synchronized InetAddress[] getAllByName (String nom_hote) throws UnknownHostExeception
- public String getHostName ()
- public byte[] getAddress ()
- public String getHostAddress ()

# Références

1. “All About Sockets”
http://java.sun.com/docs/books/tutorial/networking/sockets/

2. “Java Sockets”
http://www.cs.bgu.ac.il/~spl141/PracticalSession10