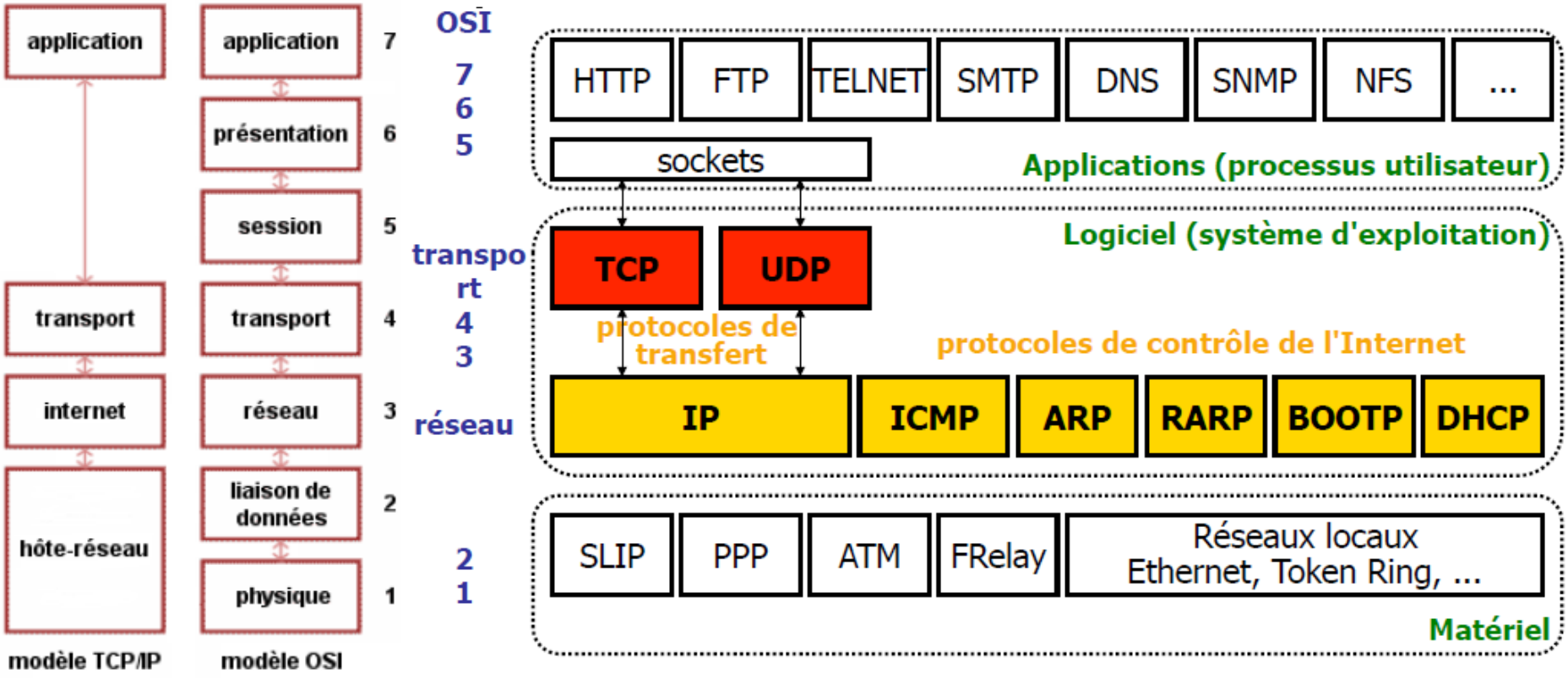
# Développement d'Application Client-Serveur

## Chapitre 4 : Les sockets

# Rappel sur les réseaux

# Rappel sur les réseaux

## Modèle OSI-TCP/IP



# Rappel sur les réseaux

## Couche réseau (internet) :

**Fonctions principales :**

- ✓ Routage et adressage des paquets.
- ✓ Contrôle de flux des paquets.

➢ **IP (Internet Protocol) :** Protocole de commutation de paquets.

Chaque machine reliée à un réseau IP se voit attribuer une adresse, dite « adresse IP ».

**Exemple : 172.168.13.4**

➢ **Paquet IP :**


L'en-tête contient :

- ✓ L'adresse IP de la machine source.
- ✓ L'adresse IP de la machine destination.
- ✓ La taille du paquet.

# Rappel sur les réseaux

IP ne garantit pas que les paquets émis soient reçus par leur destinataire. Deux protocoles évolués ont été bâtis au-dessus d'IP :

- ✓ UDP (User Datagram Protocol)
- ✓ TCP (Transmission Control Protocol)

## Couche Transport :

S'occupe de l'acheminement de l'information. Ce transport doit être transparent.

**Le protocole UDP :**

- ✓ Protocole très simple.
- ✓ Non fiable (les segments UDP peuvent être perdus).
- ✓ Sans connexion (chaque segment UDP est traité indépendamment des autres).

Ajoute deux fonctionnalités au-dessus d'IP :

- ✓ L'utilisation de numéros de ports par l'émetteur et le destinataire.
- ✓ Contrôle d'intégrité sur les paquets reçus.

# Rappel sur les réseaux

**Le protocole TCP :**

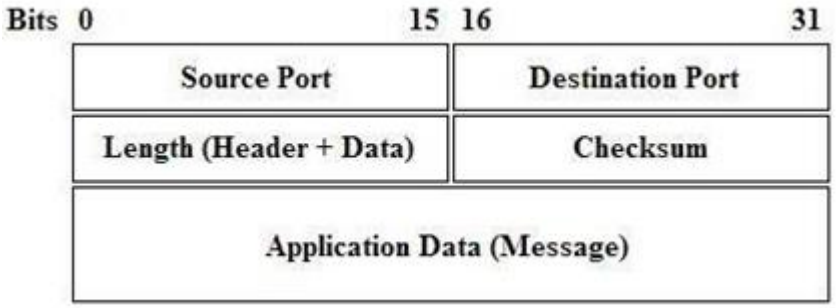
- Etablis un canal bidirectionnel entre deux processus.
  - ✓ Protocole fiable.
  - ✓ Orienté connexion.
  - ✓ Permet l'acheminement sans erreur de paquets.
- Contrairement à UDP,
  - ✓ TCP gère lui-même le découpage des données en paquets, le recollage des paquets dans le bon ordre et la retransmission des paquets perdus si besoin.
  - ✓ TCP s'adapte également à la bande passante disponible en ralentissant l'envoi des paquets.

# Rappel sur les réseaux

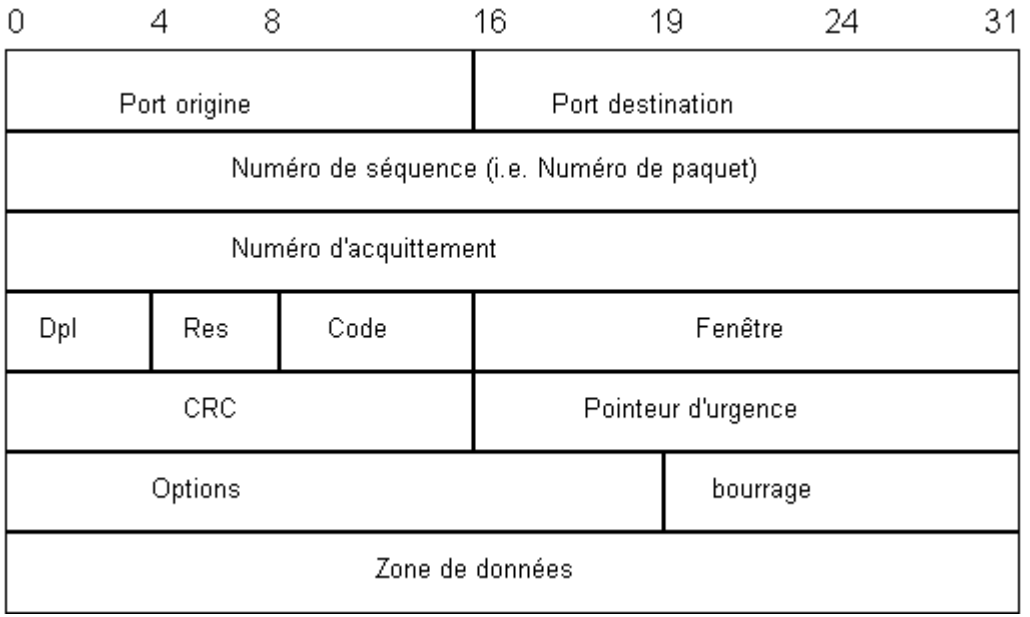
## Paquet UDP ou TCP :

Composé de deux parties :

- ✓ L'en-tête TCP ou UDP : contient les numéros de ports ainsi que le code de contrôle d'intégrité.
- ✓ Le contenu du paquet.




**Datagramme UDP**




**Datagramme TCP**

# Rappel sur les réseaux

**UDP :** Pourquoi un service non fiable sans connexion?

- Simple donc rapide (pas de délai de connexion).
- Petit en-tête donc économie de bande passante.
- Sans contrôle de congestion donc UDP peut émettre aussi rapidement qu'il le souhaite.

Souvent utilisé pour les applications multimédias :

- Tolérantes aux pertes
- Sensibles au débit

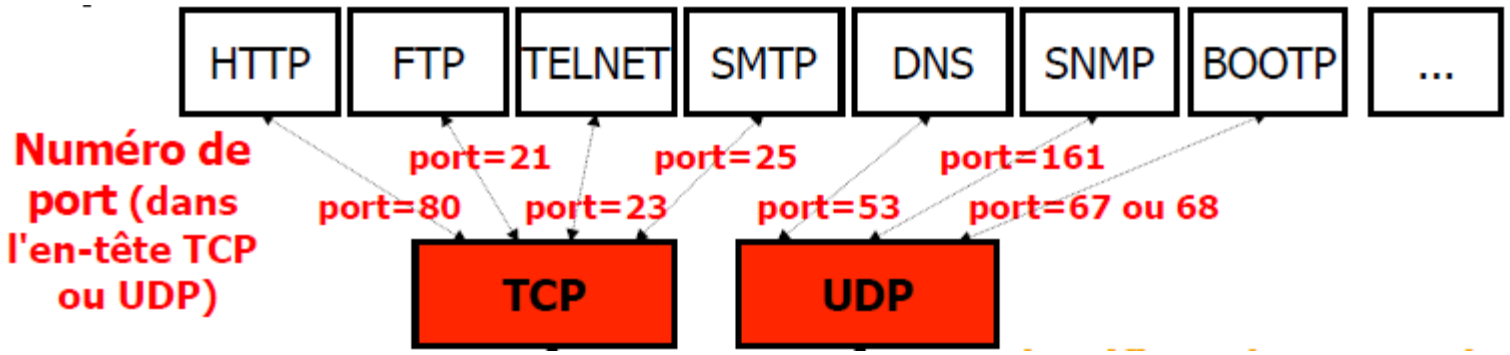
# Rappel sur les réseaux

**Ports :**

Un port réseau c'est un canal de communication qui permet de recevoir des informations en provenance de l'extérieur. Les numéros de port TCP et UDP sont codés sur 16 bits (0 à 65535).

Chaque type d'application utilise son propre port.

Les ports de 0 à 1023 (ports réservés) sont utilisés sur internet (services standards), les autres (ports utilisateurs) sont utilisés uniquement sur un réseau local (service applicatif quelconque).



# Rappel sur les réseaux

Une adresse IP sert donc à identifier de façon unique un ordinateur sur le réseau tandis que le numéro de port indique l'application à laquelle les données sont destinées.

# Rappel sur les réseaux

## La couche application :

Contient tous les protocoles de haut niveau.

- ✓ **HTTP** (HyperText Transport Protocol) : Protocole du web (Echange de requête/réponse entre un client et un serveur web)
- ✓ **FTP** (File Transfer Protocol) : Protocole de manipulation de fichiers distants (Transfert, suppression, création, …)
- ✓ **TELNET** (TELetypewriter Network Protocol) : Système de terminal virtuel (ouverture d'une session distante)
- ✓ **SMTP** (Simple Mail Transfer Protocol) : Service d'envoi de courrier électronique
- ✓ **DNS** (Domain Name System) : Assure la correspondance entre un nom symbolique et une adresse IP
- ✓ **SNMP** (Simple Network Management Protocol) : Protocole d'administration de réseau (interrogation, configuration des équipements, …)

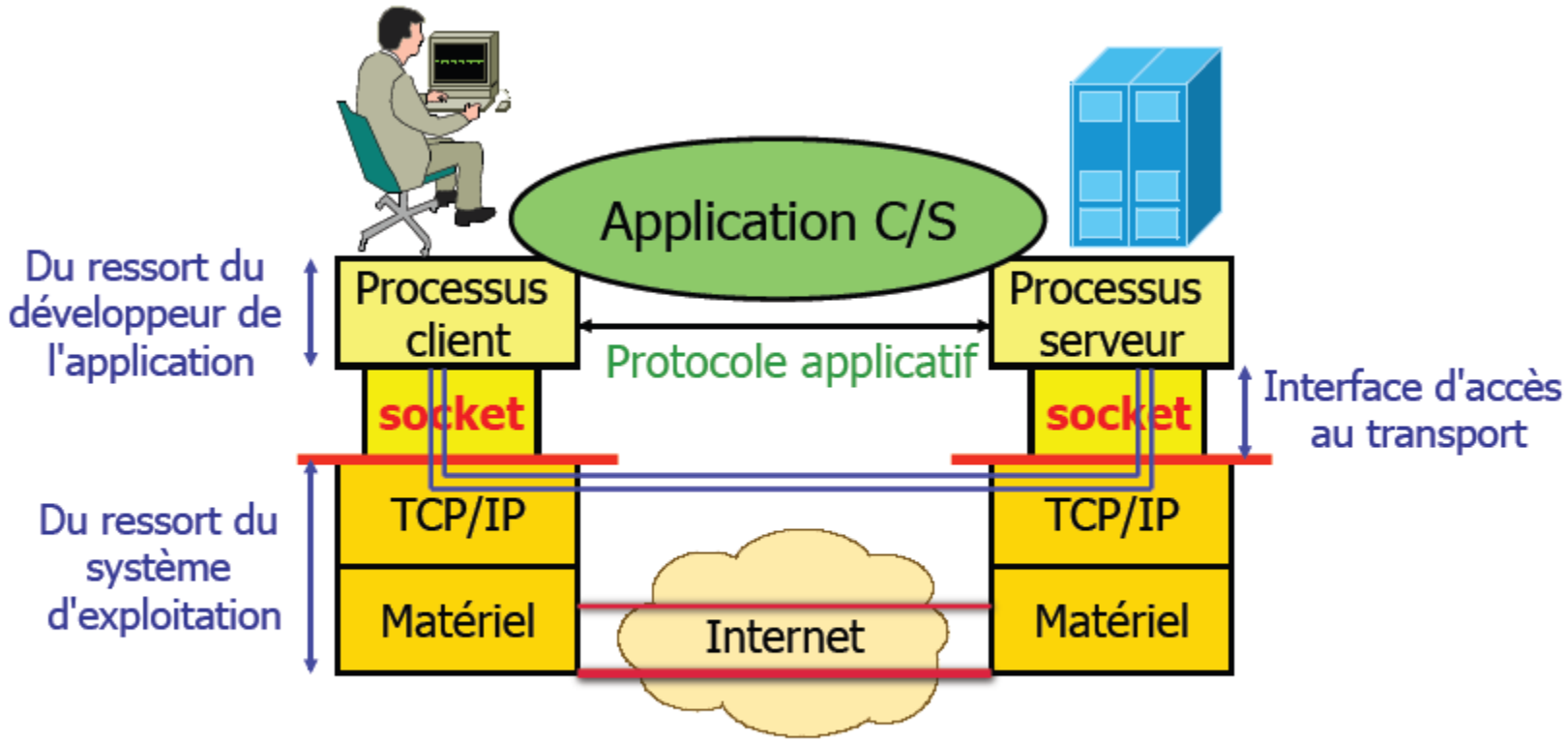
# Les sockets

# Introduction aux sockets

Une **Socket** (prise) : un **canal** de communication (élément logiciel) permettant à un processus **d'envoyer** ou **recevoir** des données.

Deux processus communiquent en émettant et recevant des données via les sockets sur une même machine ou à travers un réseau TCP/IP.

# Introduction aux sockets




**Une socket : interface locale à l'hôte, créée par l'application, contrôlée par l'OS**
**Porte de communication entre le processus client et le processus serveur**

# Introduction aux sockets

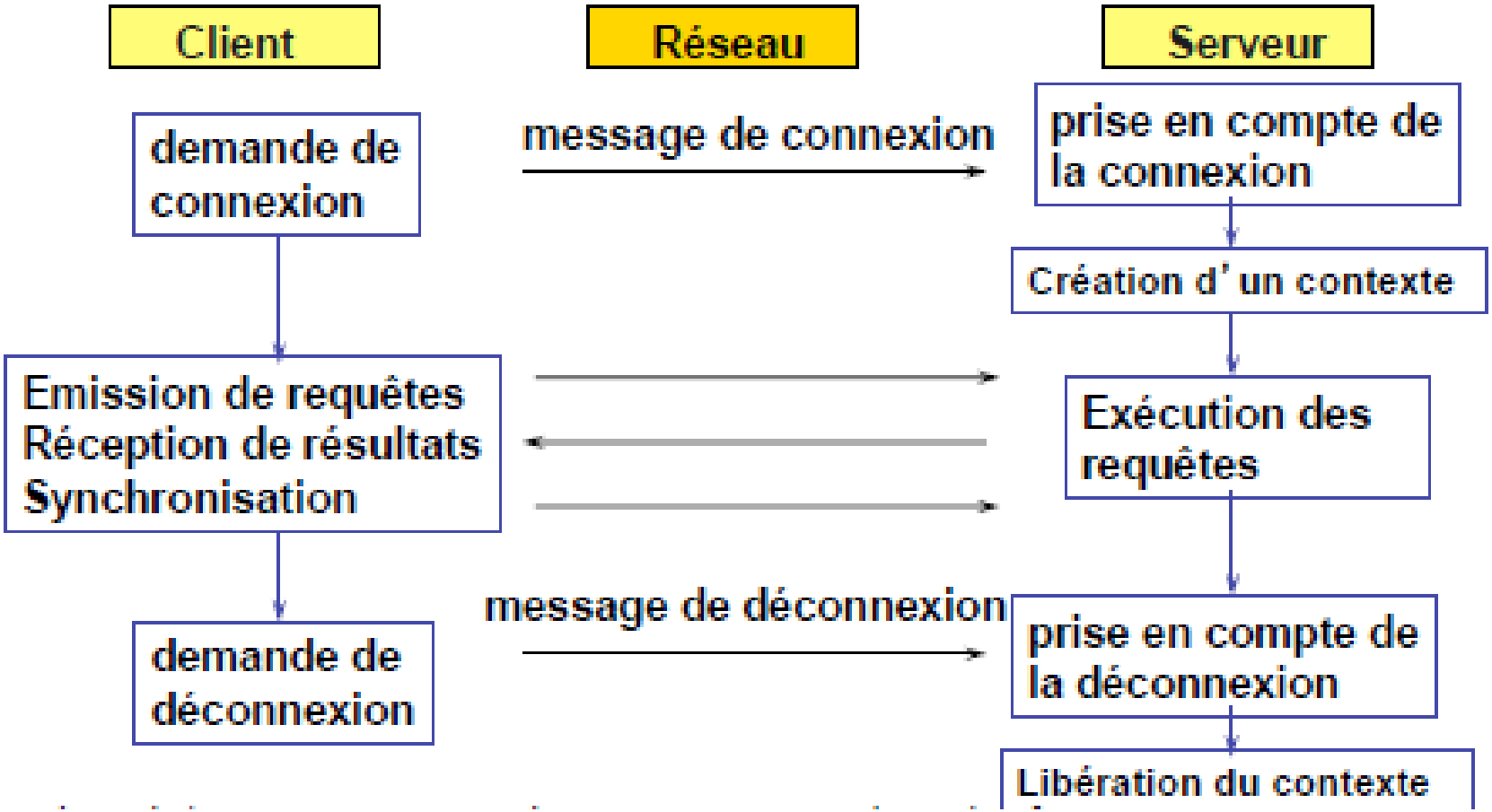
Une socket est une paire: [ adresse IP, numéro de port ].

- ✓ Disponible dans différents langages (C, Java, VB, ...)
- ✓ Cette interface permet la programmation d'applications client/serveur.

Il existe différents types de sockets associées aux différents services de transport:

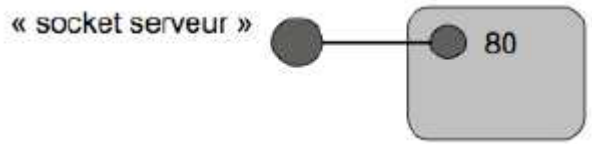
- ➢ **Stream sockets (TCP) :**
  - ✓ Etablir une communication en mode connecté.
  - ✓ Si connexion interrompue : applications informées.
- ➢ **Datagram sockets (UDP) :**
  - ✓ Etablir une communication en mode non connecté.
  - ✓ Données envoyées sous forme de paquets indépendants de toute connexion.
  - ✓ Plus rapide, moins fiable que TCP.
- ➢ **Raw sockets :**
  - ✓ Utilise directement IP ou ICMP (ex. ping).

# Utilisation de Socket avec TCP (mode connecté)



# Utilisation de Socket avec TCP (mode connecté)

1. Pour que le client puisse contacter le serveur:
   - Le processus serveur doit déjà tourner.
   - Le serveur doit avoir créé au préalable une socket pour recevoir les demandes de connexion des clients.



# Utilisation de Socket avec TCP (mode connecté)

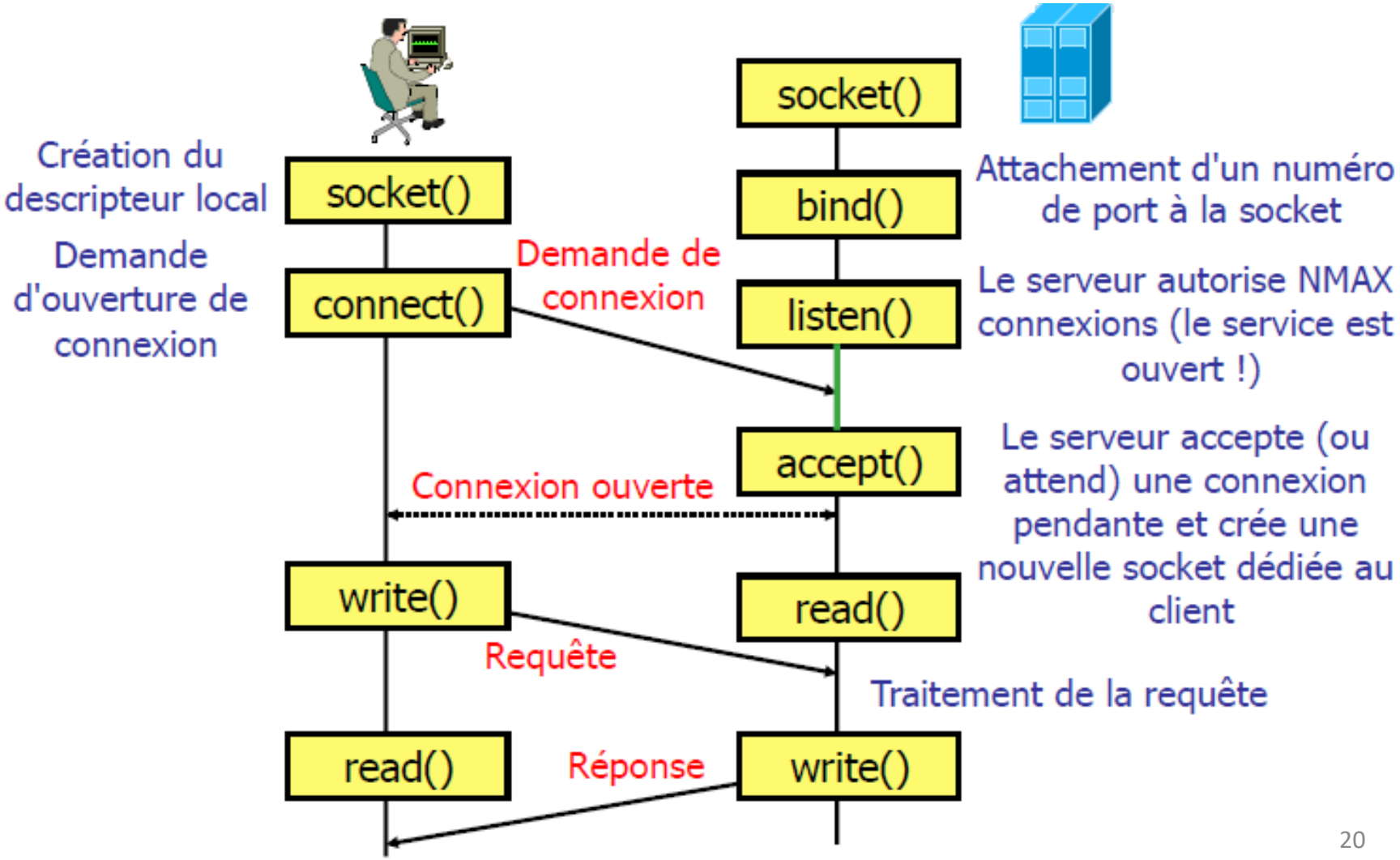
2. Le client contacte le serveur :
   - En créant une socket locale au client.
   - En spécifiant une adresse IP et un numéro de port pour joindre le processus serveur.

# Utilisation de Socket avec TCP (mode connecté)

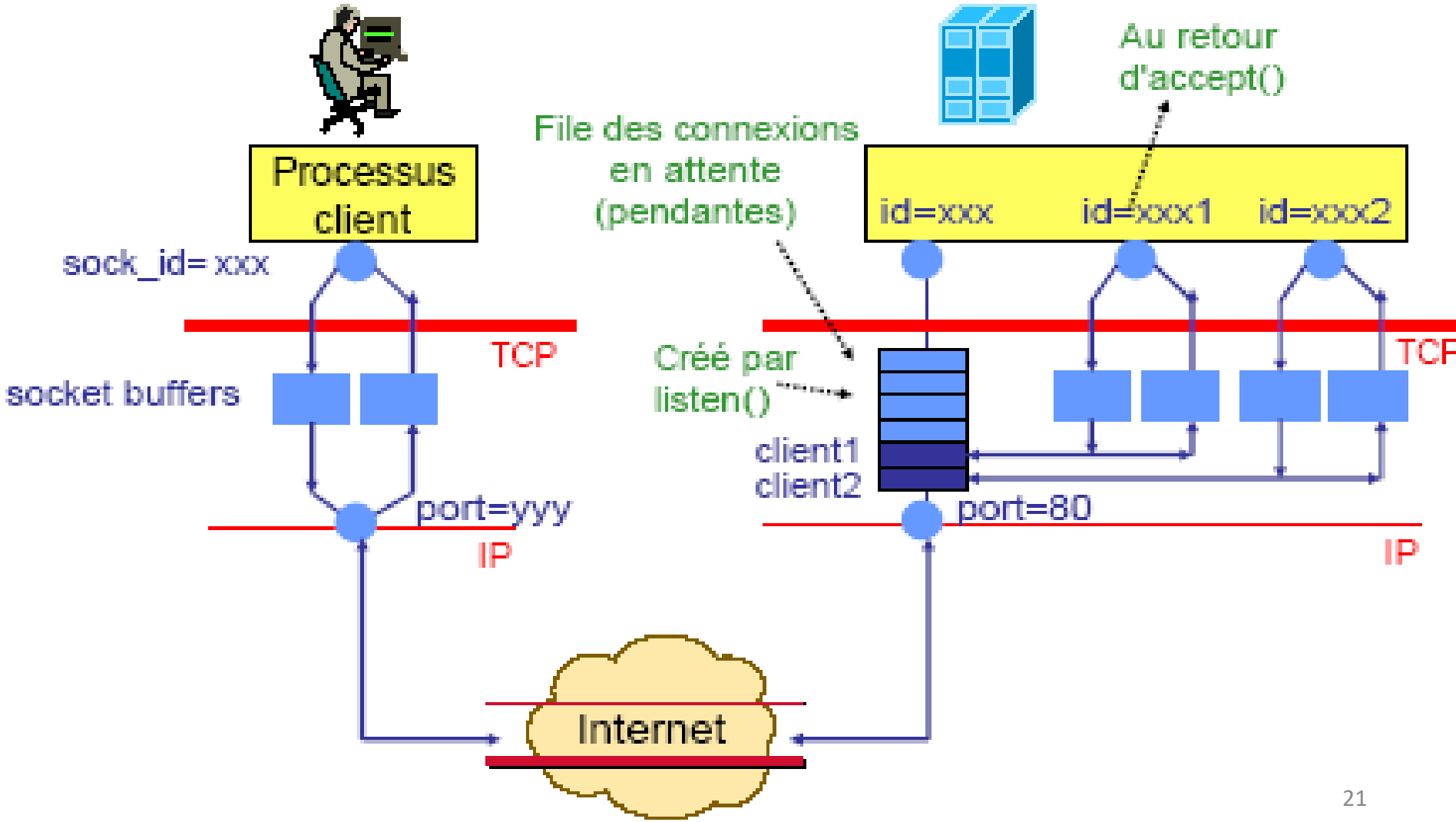
3. Le client demande alors l'établissement d'une connexion avec le serveur.
4. Si le serveur accepte la demande de connexion :
   - Il crée une nouvelle socket "socket service client" permettant le dialogue avec ce client.

     Ceci permet au serveur de dialoguer avec plusieurs clients.
   - Les deux sockets client et socket service clients sont connectées entre elles.

# Utilisation de Socket avec TCP (mode connecté)



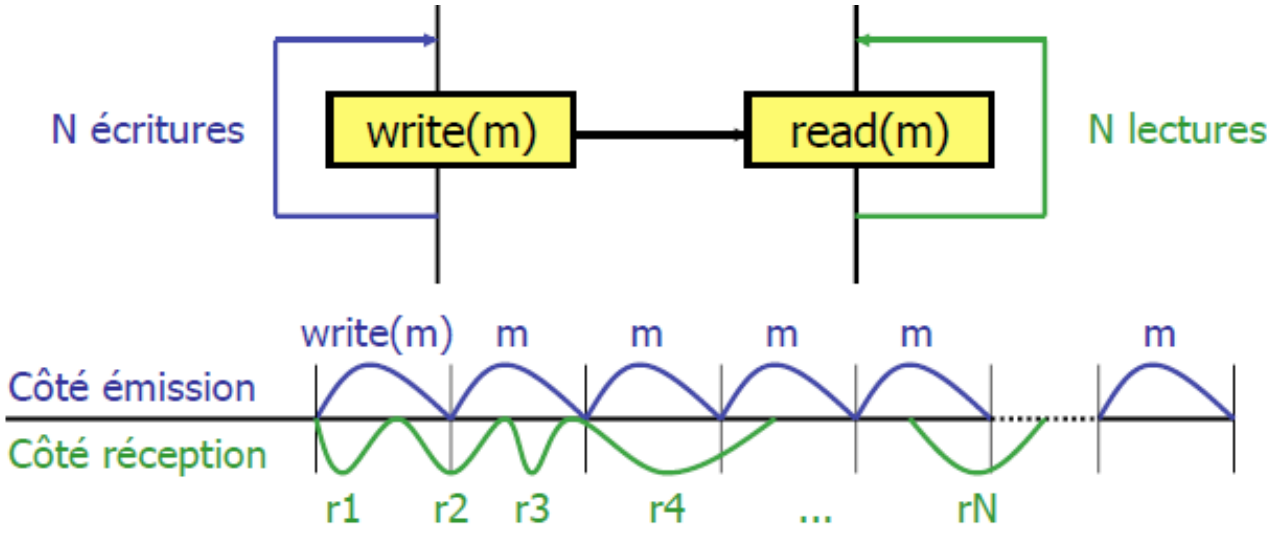
# Utilisation de Socket avec TCP (mode connecté)



# Utilisation de Socket avec TCP (mode connecté)

**Attention : les émissions/réceptions ne sont pas synchrones**

- ✓ read(m) : lecture **d'au plus** m caractères
- ✓ write(m) : écriture de m caractères




r1+r2+r3+r4+...+rN <= N*m

# Utilisation de Socket avec UDP (mode non connecté)

## Étapes de communication:

1. Pour que le client puisse contacter le serveur :
   - il doit connaître l'adresse du socket du serveur
   - le serveur doit avoir créé la socket de réception
2. Le client envoie sa requête en précisant, lors de chaque envoi, l'adresse du socket destinataire.
3. Le datagramme envoyé par le client contient l'adresse du socket émettrice (port, @IP).
4. Le serveur traite la requête et répond au client en utilisant l'adresse du socket émettrice de la requête.

# Utilisation de Socket avec UDP (mode non connecté)

Création du descripteur local

socket()

socket()

Attachement d'un numéro de port à la socket

bind()

recv_from()

Le serveur est en attente d'une requête cliente

Envoi de la requête

send_to()

Requête

Traitement de la requête

Attente de la réponse

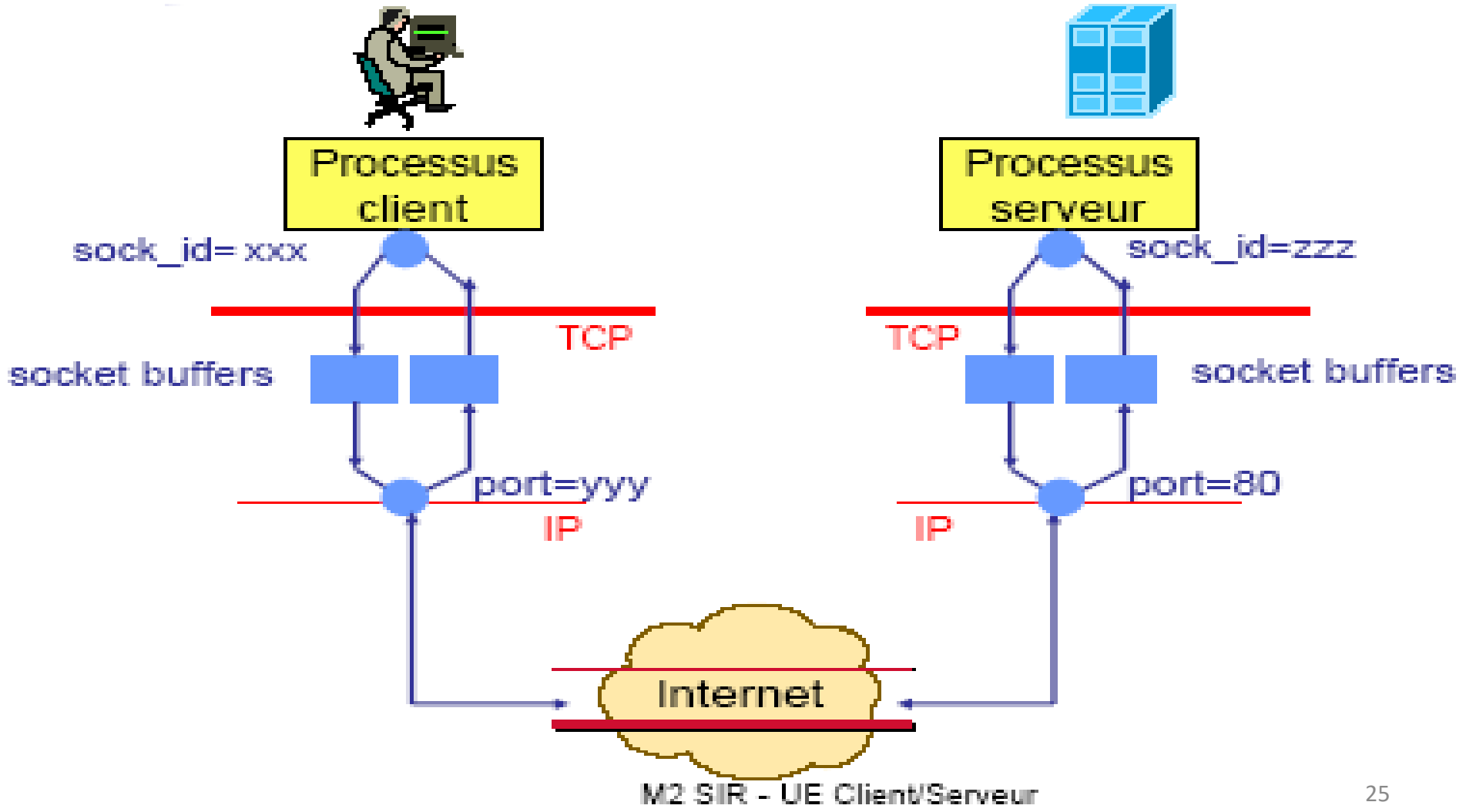
recv_from()

Réponse

send_to()

Le serveur envoie la réponse

# Utilisation de Socket avec UDP (mode non connecté)



# Récupérer une adresse IP en Java ( classe InetAddress)

Le package java.net de la plate-forme Java fournit une classe **InetAddress** qui permet de récupérer et manipuler des adresses IP. Cette classe n'a pas de constructeur, pour pouvoir avoir une instance de cette classe on a besoin des méthodes de classe suivantes :

- ✓ **getLocalHost() :** retourne un objet qui contient l'adresse IP de la machine locale. Equivalent à getByName (null) ou getByName ("localhost").
- ✓ **getByName(String nom_machine) :** retourne un objet qui contient l'adresse IP de la machine dont le nom est passé en paramètre.
- ✓ **getAllByName(String nom_machine) :** retourne un tableau d'objets qui contient l'ensemble d'adresses IP de la machine qui correspond au nom passé en paramètre.

Méthodes applicables à un objet de cette classe :

- ✓ **getHostName() :** retourne le nom de la machine dont l'adresse est stockée dans l'objet.
- ✓ **getAddress() :** retourne l'adresse IP stockée dans l'objet sous forme d'un tableau de 4 octets.

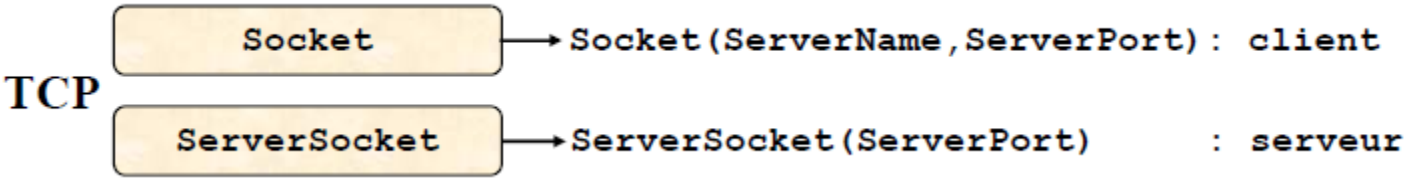
# Récupérer une adresse IP en Java ( classe InetAddress)

**Exemple de programme:**

```
public class Adressage {
public static void main(String[] args) {
  InetAddress LocaleAdresse ;
  InetAddress ServeurAdresse;
  try {
     LocaleAdresse = InetAddress.getLocalHost();
     System.out.println("L'adresse locale est : " +
     LocaleAdresse );
     ServeurAdresse = InetAddress.getByName("www.google.fr");
     System.out.println("L'adresse du serveur du site google
     est : " + ServeurAdresse);
     } catch (UnknownHostException e) {
       e.printStackTrace();
     }
  }
}
```

# Création de socket TCP en JAVA

# Création de socket TCP en Java




**Coté serveur (classe ServerSocket) :**

java.net comprend la classe **ServerSocket**, qui met en œuvre des sockets que les serveurs peuvent utiliser pour écouter et accepter les connexions des clients.

✓Création d'un objet de la classe ServerSocket sur un port spécifique :

```
ServerSocket so_server = new ServerSocket (num_port);
```

si num_port = 0, le socket est créée sur n'importe quel port libre.

✓Il existe deux autres constructeurs :

```
ServerSocket so_server = new ServerSocket (num_port, nbr_max);
```

**nbr_max** est le nombre maximal de connexions traitées simultanément.

```
ServerSocket so_server = new ServerSocket (num_port, nbr_max, adresse_locale);
```

**adresse_locale** est l'adresse du serveur ( de type **InetAddress**).

# Création de socket TCP en Java

## Coté client (classe Socket)

Java.net fournit une classe **Socket** qui met en œuvre un côté d'une connexion bidirectionnelle entre le programme Java du client et celui du serveur.

La création d'un socket pour le client nécessite un des constructeurs suivants :

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

- ✓ Crée un socket et tente de s'y connecter.
- ✓ Lève les exceptions :

  UnkownHostException = si le nom de la machine est inconnu.

  IOException = problèmes de connexion.

**Paramètres:**

- ✓ hote = chaine de caractère contenant le nom de l'hôte
- ✓ port = port de destination (port auquel la socket doit se connecter).

**Exemple :**

```
Socket s = new Socket ("ElKendy-PC", 4567);
```

# Création de socket TCP en Java

**Coté client (classe Socket) (suite)**

**`public Socket (InetAddress adr, int port)`** `throws IOException`

- ✓ Crée un socket et tente de s'y connecter.
- ✓ Lève l'exception IOException

**Paramètres :**

- ✓ adr = InetAddress contenant l'adresse de l'hôte
- ✓ port = port de destination.

**`public Socket(String hote, int port, InetAddress adr_loc, int port_loc)`** `throws IOException`

**`public Socket(InetAddress adr, int port, InetAddress adr_loc, int port_loc)`** `throws IOException`

- ✓ Les deux derniers paramètres permettent de choisir l'interface réseau (et le port) d'où doit partir la connexion.

# Création de socket TCP en Java

En mode connecté, l'attente de connexion provenant du client se fait du coté serveur par la création d'une socket (socket service client) à l'aide de la méthode **`accept()`**. La méthode accept() reste bloquante tant qu'elle n'a pas détecté de connexion.

**Exemple :**

```
Socket soc = so_server.accept();
```

**`setSoTimeout(int)`** : spécifie un délai maximal d'attente (exception de type InterruptedIOException).

**`getSoTimeout()`** : lecture de la valeur d'attente maximale.

# Création de socket TCP en Java

## Fermeture de la socket

**public void close() throws IOException**

Même si Java ferme les sockets à la fin du programme, il est de bon usage de fermer explicitement les sockets via la méthode close.

Valable pour tous les types de Sockets

- ✓ Sockets de service
- ✓ Sockets clients
- ✓ Sockets serveurs (ServerSockets)

## Exemple :

```
so_server.close() ;
s.close() ;
```

# Création de socket TCP en Java

**Exemple de programmes:**

Etablir une communication entre un client et un serveur

**Serveur1.java:**

```
public class Serveur1 {
public static void main(String[] args) {
  ServerSocket s ;
  Socket soc ;
  try {
     s = new ServerSocket(2022);
     soc = s.accept();
     System.out.println("Un Client s'est connecté !");
     soc.close();
     s.close();
     } catch (IOException e) {
       e.printStackTrace();
  }
  }
}
```

# Création de socket TCP en Java

Client1.java:

```
import java.io.IOException;
import java.net.InetAddress;
import java.net.Socket;
import java.net.UnknownHostException;
public class Client1 {
public static void main(String[] args) {
  Socket s1;
  try {
  s1 = new Socket(InetAddress.getLocalHost(),2022);
  s1.close();
  } catch (UnknownHostException e) {
        e.printStackTrace();
  } catch (IOException e) {
        e.printStackTrace();
  }
}
}
```

# Socket TCP en Java
# échange de messages

Une fois la connexion établie et les sockets en possession, il est possible de récupérer le flux d'entrée et de sortie de la connexion TCP vers le serveur. Il existe deux méthodes (de la classe Socket) pour permettre la récupération des flux :

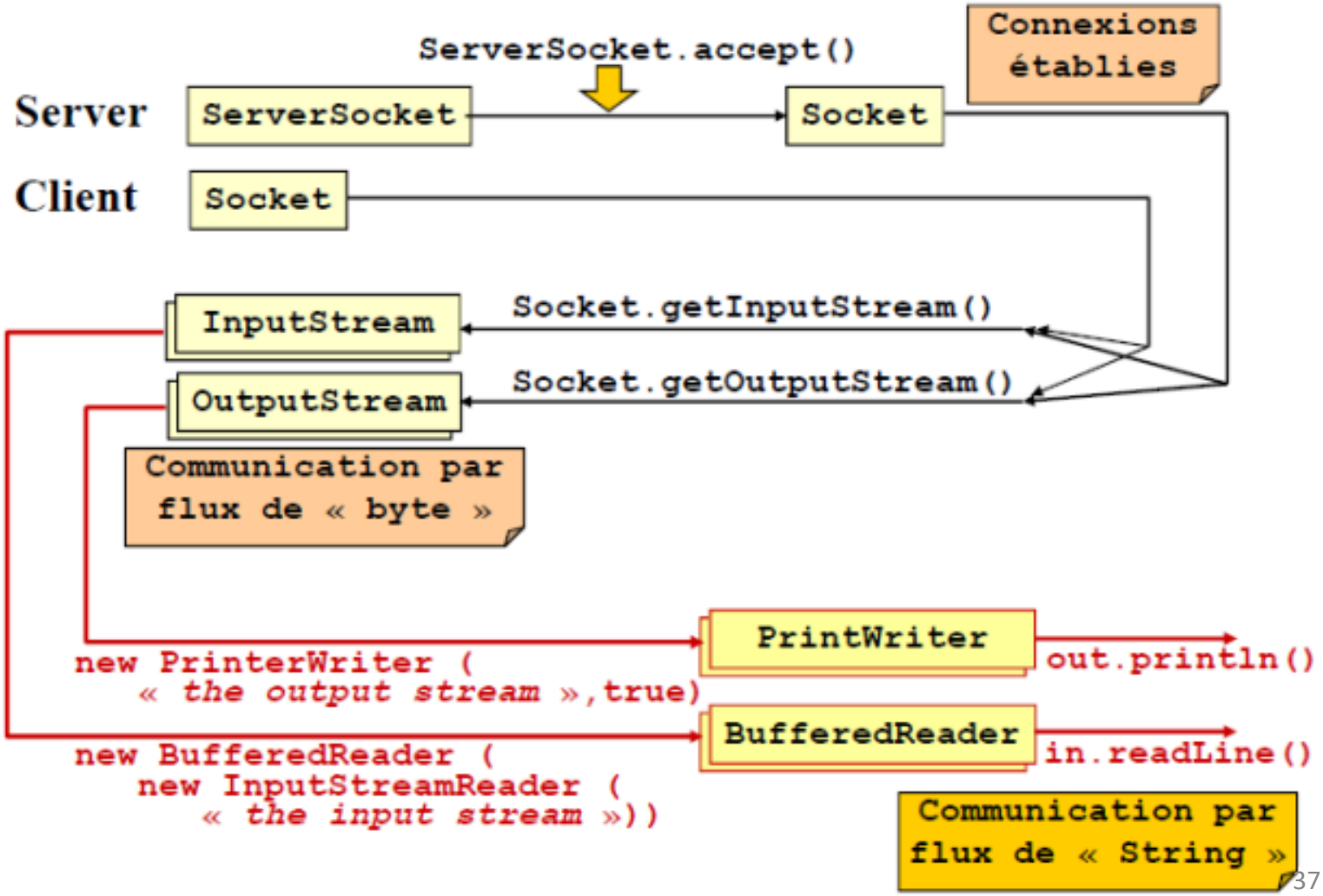
- ✓ **getInputStream()**: permet de récupérer un flot de données en provenance d'une socket (flux entrant).
- ✓ **getOutputStream()**: permet d'envoyer un flot de données à destination d'une socket (flux sortant) .

En général le type d'entrée et sortie utilisé est **BufferedReader** et **InputStreamReader** pour la lecture, **PrintWriter** pour l'écriture.

**Exemple :**

```
BufferedReader entree = new BufferedReader(new
InputStreamReader(s.getInputStream()));
PrintWriter sortie = new
PrintWriter(soc.getOutputStream(),true);
```

# Socket TCP en Java
# échange de messages



# Socket TCP en Java
# échange de messages

**Exemple de programme** (échange de message entre le client et le serveur)

**Serveur2.java**

```
public class Serveur2 {
public static void main(String[] args) {
          ServerSocket s ;
          Socket so;
          PrintWriter out;
          try {
          s = new ServerSocket(2022);
          System.out.println("Le serveur est à l'écoute du port
"+s.getLocalPort());
          so = s.accept();
          System.out.println("Un Client s'est connecté");
          out = new PrintWriter(so.getOutputStream());
          out.println("Vous êtes connecté Client !");
          out.flush();
          So.close();
          s.close();
          }catch (IOException e) {
                    e.printStackTrace();
          }
}
}
```

# Socket TCP en Java
# échange de messages

Client2.java

```
public class Client2 {
public static void main(String[] args) {
        Socket s1;
        BufferedReader in;
        try {
                s1 = new Socket(InetAddress.getLocalHost(),2022);
                System.out.println("Demande de connexion");
        in = new BufferedReader (new InputStreamReader
(s1.getInputStream()));
                System.out.println(in.readLine());
                s1.close();
        } catch (UnknownHostException e) {
                e.printStackTrace();
        }catch (IOException e) {
                e.printStackTrace();
        }
}
}
```

# Socket TCP en Java

## Serveurs concurrents (Utilisation des threads)

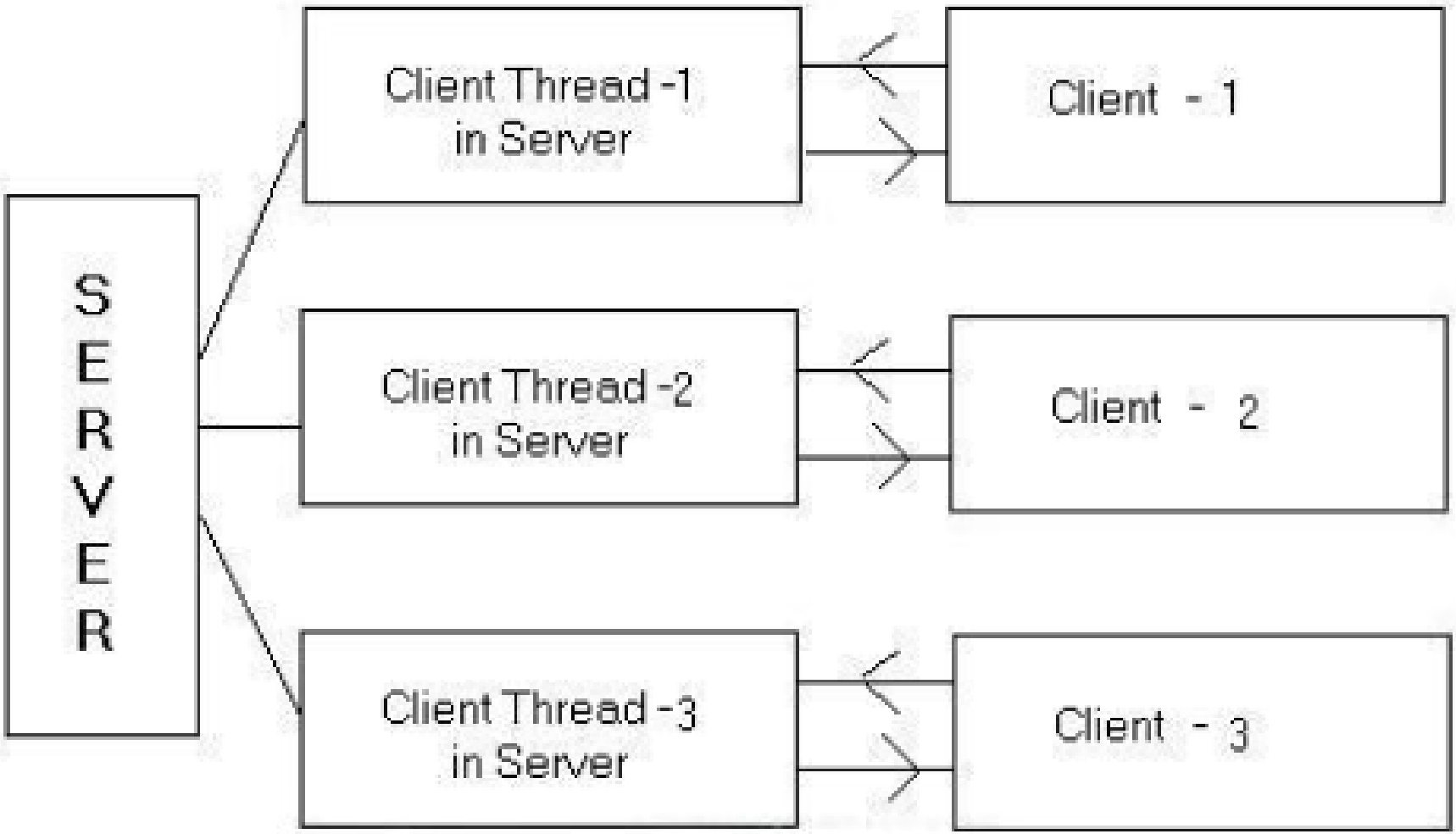
Le type du serveur précédent est dit itératif (un client à la fois), parce que les demandes de connexion sont satisfaites l'une après l'autre. Si chaque connexion doit être durable (avec un dialogue entre client et serveur, ou bien un temps de calcul du serveur important), il faut plutôt réaliser un serveur concurrent (plusieurs clients simultanément) à l'aide de threads.

Après avoir créer un objet **ServerSocket** par le serveur, on le place comme paramètre à un constructeur de la classe qui implémente l'interface **Runnable** ou étend la classe **Thread**.

Chaque demande de connexion acceptée par le serveur provoque la création et le lancement d'un nouveau thread qui prend en charge la communication avec le client (on parle de socket TCP non bloquant):

# Socket TCP en Java

## Serveurs concurrents (Utilisation des threads)



# Socket TCP en Java

## Serveurs concurrents (Utilisation des threads)

**Exemple:**

Serveur3.java

```
public class Serveur3 {
        public static void main(String[] args) {
        ServerSocket s;
        try{
                s = new ServerSocket(2022);
                Thread t = new Thread(new AccepterClients(s));
                t.start();
                System.out.println(" Serveur concurrent prêt :");
        }catch (IOException e){
                e.printStackTrace();
        }
        }
}
```

# Socket TCP en Java

## Serveurs concurrents (Utilisation des threads)

**Exemple: (suite)**

```
public class AccepterClients implements Runnable{
        private ServerSocket so;
        private Socket sock;
        private int nbrclient = 1;
        public AccepterClients(ServerSocket s1){
                so = s1;
        }
        public void run() {
        try{
        while(true){
                sock = so.accept();
                System.out.println("Le client numéro "+nbrclient+ " est
connecté !");
        nbrclient++;
        sock.close();
        }
        }catch(IOException e){
                e.printStackTrace();
        }
        }
}
```

# Socket TCP en Java

## Serveurs concurrents (Utilisation des threads)

**Exemple:**

Client3.java

```
public class Client3 {
	public static void main(String[] args) {
	Socket soc;
	try {
		soc = new Socket("localhost",2022);
		soc.close();
	} catch (IOException e) {
		e.printStackTrace();
	}
	}
}
```

# Socket TCP en Java
# échange d'objets

- Une classe **Sérializable**:

```
public class Produit implements Serializable  {
    private int num;
    private String designation;
    private float prix;
    public Produit(int num, String designation, float prix) {
        this.num=num;
        this.designation=designation;
        this.prix=prix;
    }
    public String toString() {
        return num+" -- "+designation+" -- "+prix;
    }
}
```

# Socket TCP en Java
# échange d'objets

- Pour **sérialiser** un objet (envoyer un objet vers le client)

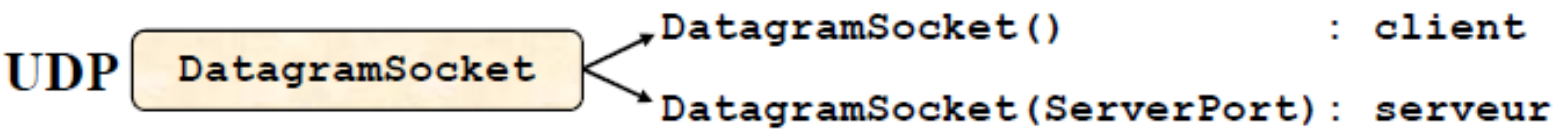
```
OutputStream os=s.getOutputStream();
ObjectOutputStream oos=new ObjectOutputStream(os);
Produit p=new Produit(1, "Table", 500);
oos.writeObject(p);
```

- Pour lire un objet envoyé par le serveur( **désérialisation**)

```
InputStream is=s.getInputStream();
ObjectInputStream ois=new ObjectInputStream(is);
Produit p=(Produit)ois.readObject();
```

# Création de socket UDP en JAVA

# Création de Socket UDP en Java




- Pas de différence de classe entre le client et le serveur.
- Constructeurs :

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

  - ✓ Crée une nouvelle socket en la liant à un port quelconque libre.
  - ✓ Exception levée en cas de problème.

**Exemple :** Création d'un socket coté émetteur.

```
DatagramSocket dgSocket = new DatagramSocket ( ) ;
```

# Création de Socket UDP en Java

- Constructeurs (suite) :

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

  - ✓ Crée une nouvelle socket en la liant au port local précisé par le paramètre port.
  - ✓ Exception levée en cas de problème : notamment quand le port est déjà occupé

**Exemple :** Création d'un socket liée à un port d'écoute coté récepteur.

DatagramSocket dgSocket = new DatagramSocket (4030) ;

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

  - ✓ Analogue au précédent, mais permet en plus de choisir l'adresse IP liée au port (utile en cas de plusieurs interfaces réseaux).

# Création de Socket UDP en Java

✓ **Classe DatagramPacket :**

Permet de manipuler les datagrammes (structure des données en mode UDP) échangés entre les correspondants.

➢ **Constructeurs :**

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

Crée un objet employé pour **l'envoi** d'un datagramme.

**Paramètres :**

- ✓ **buf** : tableau d'octets contenant les données à envoyer.
- ✓ **length** : taille du tableau ou des données à envoyer. (ne pas préciser une taille supérieure à celle de buf).
- ✓ **address** : adresse IP de la machine destinataire du datagramme.
- ✓ **port** : port de destination.

# Création de Socket UDP en Java

**Exemple :** (coté émetteur).

```
String message="BTS DSI";
byte [] data = message.getBytes( );
InetAddress adr ="192.168.1.22" ;
DatagramPacket paquet = new DatagramPacket(data,
data.length, adr, 2000);
```

**Constructeur: public DatagramPacket (byte[ ] buf, int length)**

Crée un objet employé pour la **réception** d'un datagramme.

**Paramètres :**

- ✓ **buf** : tableau d'octets permettant la réception des données.
- ✓ **length** : taille du tableau ou des données à recevoir (si la taille du tableau est insuffisante, les données seront tronquées).

**Exemple :** Préparation du paquet de réception.

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

# Création de Socket UDP en Java

- **Méthodes de la classe DatagramPacket :**

**public synchronized InetAddress getAddress ()**
Retourne l'adresse stockée dans le paquet.
**public synchronized int getPort ()**
Retourne le port stocké dans le paquet.
**public synchronized byte[] getData ()**
Retourne les données stockées dans le paquet.
**public synchronized int getLength ()**
Retourne la taille des données stockées dans le paquet.
**public synchronized void setAddress(InetAddress iaddr)**
Modifie ou affecte l'adresse de destination.
**public synchronized void setPort(int iport)**
Modifie ou affecte le port de destination.
**public synchronized void setData(byte ibuf[])**
Modifie ou affecte la référence de la zone contenant les données.
**public synchronized void setLength(int ilength)**
Modifie ou affecte la taille de la zone contenant les données.

# Création de Socket UDP en Java

- **Emission/Réception de datagrammes :** (méthodes de la classe DatagramSocket)

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

- ✓ Envoie un datagramme passé en paramètre.
- ✓ Lève l'exception IOException en cas d'erreur.

**Paramètre :**

- ✓ DatagramPacket p : datagramme prêt à l'envoi.

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

- ✓ Attends la réception d'un datagramme et remplie le datagramme passé en paramètre par les données reçues (cette méthode est bloquante).
- ✓ Lève l'exception IOException en cas d'erreur.

**Paramètre :**

- ✓ DatagramPacket p : datagramme **vide** prêt à recevoir les données

Il est possible de spécifier un délai d'attente maximal en réception. Pour cela, il faut positionner une variable de timeout sur la socket au moyen de la méthode setSoTimeout (int timeout).

# Création de Socket UDP en Java

- **Fermeture de la Socket:**

**public void close () throws IOException**

Ferme la socket et libère les ressources qui lui sont associées. La socket ne pourra plus être utilisée ni pour envoyer, ni pour recevoir des datagrammes.

- ✓ **Exemple de programmes :** (communication UDP entre un client et un serveur)

Dans cet exemple :

- Le serveur attend une chaîne de caractères et la retourne.
- Le client envoi une chaîne de caractères, attend que le serveur la lui retourne et l'affiche.

# Socket UDP en Java

## Serveur.java :

```
public class Serveur{
final static int port = 8532;
final static int taille = 1024;
final static byte buffer[] = new byte[taille];
public static void main(String args[ ]) throws Exception{
        DatagramSocket socket = new DatagramSocket(port);
        socket.setSoTimeout(10000);
        while(true){
        DatagramPacket data = new DatagramPacket(buffer,buffer.length);
        socket.receive(data);
        String msg = new String(data.getData(),0,data.getLength());
        if (msg.equals("fin")) break;
        System.out.println("Le message suivant : " + msg) ;
        System.out.println("à été reçu et renvoyé vers : " +
data.getAddress());
        socket.send(data);
        }
        socket.close() ;
}
}
```

# Socket UDP en Java

Client.java :

```
public class Client{
final static int taille = 1024;
static byte buffer[] = new byte[taille];
public static void main(String args[]) throws Exception{
BufferedReader InClavier = new BufferedReader(new InputStreamReader(System.in));
InetAddress adrServeur = InetAddress.getByName("127.0.0.1");
System.out.println("Tapez votre message : ");
String message = InClavier.readLine();
buffer = message.getBytes();
int lg = buffer.length;
DatagramPacket dataSent = new DatagramPacket(buffer,lg,adrServeur,8532);
DatagramSocket socket = new DatagramSocket();
System.out.println("Port local utilisé : " + socket.getLocalPort());
socket.send(dataSent);
DatagramPacket dataRecieved = new DatagramPacket(new byte[lg],lg);
socket.receive(dataRecieved);
System.out.println("Données reçues : " + new String(dataRecieved.getData()));
System.out.println("De : " + dataRecieved.getAddress() + " : " +
dataRecieved.getPort());
socket.close();
}
}
```

# Socket UDP en Java

**Remarque :**

Si le serveur doit envoyer un message propre au client, Il doit tirer les informations nécessaires (adresse et port du client) à partir du datagramme reçu du client.

Serveur.java :

```
class Serveur{
final static byte buffer[] = new byte[1024];
public static void main(String args[]) throws Exception{
DatagramSocket socket = new DatagramSocket(8532);
while(true){
DatagramPacket data = new DatagramPacket(buffer,buffer.length);
socket.receive(data);
String msg = new String(data.getData(),0,data.getLength());
System.out.println("Le message : " +msg+" à été reçu à partir de : " +
data.getAddress());
String confirmation = "Message reçu";
byte [ ]tampon = confirmation.getBytes();
InetAddress adr = data.getAddress();
int portClient = data.getPort();
DatagramPacket dataConf = new DatagramPacket(tampon,tampon.length,adr,portClient);
socket.send(dataConf);
}
}
}
```

# Création de socket UDP Multicast en Java

- **La classe MulticastSocket**

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

- ✓ **Constructeurs :**

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

- ✓ **Abonnement/résiliation :**

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

# Création de socket UDP Multicast en Java

- **public void joinGroup(InetAddress adresseMulticast) throws IOException**

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

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

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

- **public void leaveGroup(InetAddress adresseMulticast) throws IOException**

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

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

# Création de socket UDP Multicast en Java

- **Choix du TTL :**

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

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

**public int getTimeToLive() throws IOException**

**public void setTimeToLive(int ttl) throws IOException**

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

# Création de socket UDP Multicast en Java

**Exemple**: programme qui envoie et reçoit des datagrammes avec le multicasting IP

```
public class MulticastingIP{
public static void main(String args[ ]) throws SocketException, IOException{
String msg = "Bonjour ";
InetAddress groupe = InetAddress.getByName("228.5.6.7");
MulticastSocket s = new MulticastSocket(50000);
s.joinGroup(groupe);
DatagramPacket envoi = new DatagramPacket(msg.getBytes(), msg.length(),
groupe, 50000);
s.send(envoi);
while (true){
byte[] buf = new byte[1024];
DatagramPacket reception = new DatagramPacket(buf, buf.length);
s.receive(reception);
String texte=new String(buf, 0, reception.getLength());
System.out.println("Reception de : "+ reception.getAddress().getHostName()+
" sur le port "+reception.getPort()+" :\n"+ texte );
}
}
}
```

# Création de socket UDP Multicast en Java

**Remarque :**

Il est possible de créer un programme qui :

- Offre le même service en UDP et en TCP sur le même numéro de port.
- Ou encore, d'inventer un protocole client/serveur qui dialogue en UDP, et qui ouvre de temps en temps des liaisons TCP pour transmettre de longs flux ; ou inversement, qui ouvrent un canal TCP, et qui échangent des informations de signalisation en UDP.

**Exemple** : (programme serveur recevant un message en mode TCP, envoi confirmation vers l'expéditeur et transmet le message reçu aux autres clients en mode multicast UDP).

# Création de socket UDP Multicast en Java

```
public class Serveur{
public static void main(String[ ] args) {
ServerSocket s ;
Socket soc ;
try {
s = new ServerSocket(2000);
soc = s.accept();
PrintWriter out = new
PrintWriter(soc.getOutputStream(),true);
BufferedReader in = new BufferedReader(new
InputStreamReader(soc.getInputStream()));
String message = in.readLine();
System.out.println(" Message reçu du client "+
soc.getInetAddress()+" : " + message);
out.println("Message reçu.");
soc.close();
s.close();
```

# Création de socket UDP Multicast en Java

```
byte[ ] buffer = message.getBytes();
InetAddress groupe =
InetAddress.getByName("228.5.6.7");
MulticastSocket sm = new MulticastSocket(50000);
sm.joinGroup(groupe);
DatagramPacket paquet = new
DatagramPacket(buffer,buffer.length,groupe,50000);
sm.send(paquet);
}catch (IOException e){
e.printStackTrace();
}
}
}
```

# Merci pour votre attention