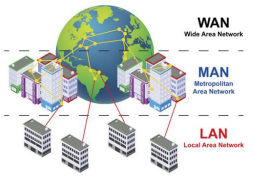
Communication via réseau

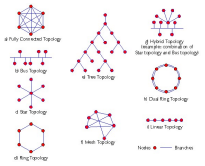
CHAPITRE 3

Introduction

❖Un **S.D.** est un ensemble de nœuds sur un réseau qui communiquent entre eux en échangeant des messages pour coordonner leurs actions afin d'effectuer un ensemble de tâches.

❖**Vision Matérielle:**
- **Machines** multi-processeurs avec mémoire partagée
- **connectées** entre eux (liaison filaire ou sans fil)
- Via un **réseau** (typologie (LAN, MAN, WAN), topologie (Bus, Etoile, anneau, etc.))





Introduction

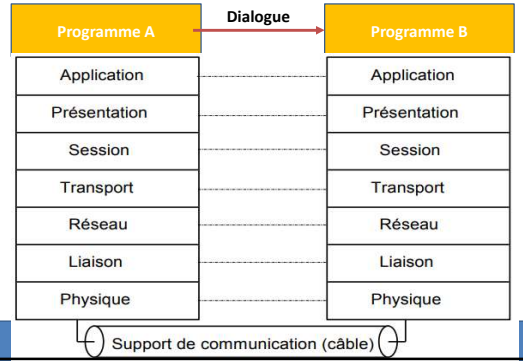
❖Vision logicielle:
- Ensemble **d'entités logiciels** s'exécutant indépendamment et en parallèle sur des **machines**
  - On parlera de processus (programme en cours d'exécution) plutôt que de logiciel.
- Ces processus **communiquent** entre eux via des **canaux** de communication

❖**La Communication est un point crucial**

Rappel sur les réseaux

❖**Système ouvert:** système qui désirant échanger des données avec un autre

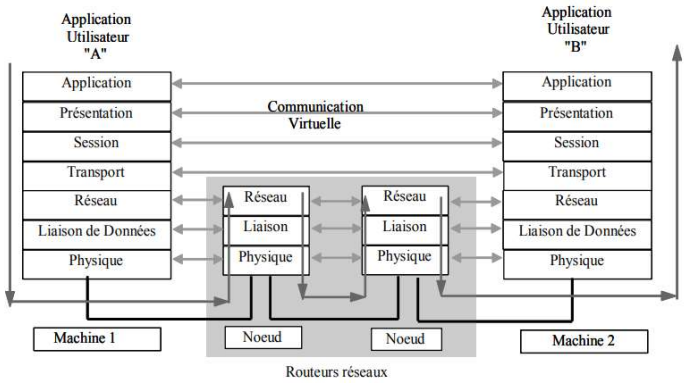
❖Norme OSI (**O**pen **S**ystem **I**nterconnection) de l'ISO : architecture en 7 couches



Rappel sur les réseaux

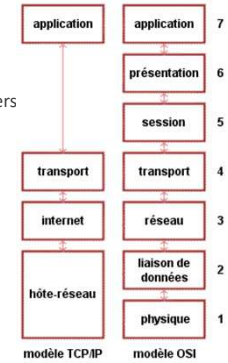
- Norme OSI de l'ISO : architecture en 7 couches
- **Application** : protocoles applicatifs (transfert de fichiers, messagerie, pages web, etc.)
- **Présentation** : représentation des données (entiers, chaînes de caractères...) et leurs codage/décodage
- **Session** : synchronisation et la gestion (ouverture/ fermeture) d'une connexion entre applications,.
- **Transport** : fournir une communication de bout en bout entre 2 applications (TCP/UDP)
- **Réseau** : construction des paquets, trouver les routes à travers un réseau pour atteindre la machine destinataire (IP adresse)
- **Liaison** : assemblage des blocs de données (trames) et gestion d'accès au support physique, assure que les données envoyées sur le support physique sont bien reçues par le destinataire
- **Physique** : transmission/réception des données binaires sur un support physique (radio,)

Rappel sur les réseau



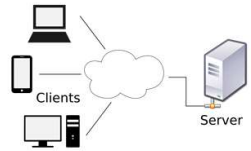
Rappel sur les réseaux

- **Modèle TCP/IP**: le plus utilisé en pratique (Internet)
- **Couche réseau** : IP (Internet Protocol)
  - Définition de l'adressage
  - Recherche des routes à travers le réseau pour acheminer les message vers leur destination prévue.
- **Couche transport** (TCP/UDP)
  - TCP : connexion virtuelle directe (fiable) entre 2 machines.
  - UDP : mode datagramme (non fiable)
    - Envoi de paquets de données
    - Pas de gestion de l'ordre d'arrivée, pas de gestion des paquets perdus

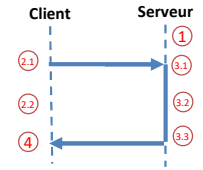


Architecture Client / Serveur (Rappel)

- Architecture très populaire dans les systèmes distribués et convient à une grande variété d'applications
- Deux composants majeurs qui interagissent via un réseau
  - **Serveur**: un composant (processus) **implémentant** un **service spécifique** (ex: fichiers, base de données).
  - **Client:** Composant qui **demande** un **service** à un serveur en lui envoyant une demande et en attendant la réponse du serveur.



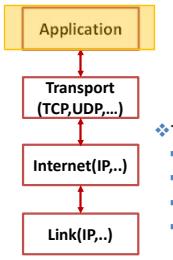
- **Scénario basique:**
  1. **Serveur:** en attente d'une requête ;
  2. **Client**: envoie une requête au serveur et attend une réponse
  3. **Serveur**: Apres réception d'une requête, il la traite et envoie une réponse
  4. **Client: reçoit la réponse** du serveur



- Architecture P2P????

❖Modèle TCP/IP est le plus répandu

❖Les applications (client et serveur) désirant réaliser une communication réseau doivent faire appel au services offerts par la couche transport



❖**TCP Transmission control Protocol**
- Protocole fiable (téléphonie)
- **Point à point**
- **Lent** (gestion de flux)
- **Applications: HTTP, FTP, SMTP**

❖**UDP: User Datagram Protocol**
- Protocole Peu Fiable (poste)
- **Emission multicast/Broadcast**
- **Rapide (**perte occasionnelle d'un paquet est tolérable)
- **Applications**: Streaming, VOIP

Adresse d'une application

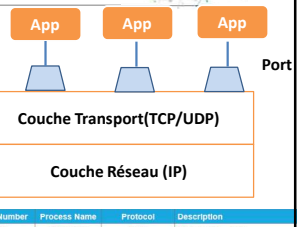
❖Comment identifier (communiquer) une application dans un réseau ?

❖**Adresse « réseau » application**: couple de 2 informations
- **Adresse IP** : identifiant de la machine sur laquelle tourne l'application
- **Numéro de port** : identifiant local réseau de l'application

❖**Couche réseau** : adresse IP
- Exemple : 192.168.1.2

❖**Couche transport :** numéro de port (TCP ou UDP)
- Entier codé sur 16 bits (65 536)
- Sur un port : réception ou envoi de données
- Ports < 1024 : réservés pour les applications ou protocoles systèmes (**Ex** : 80 = HTTP, 21 = FTP, ...)

❖Adresse notée : IP:port
- **Ex:** 172.217.19.132:80 : accès au serveur Web tournant sur la machine d'adresse IP 172.217.19.132



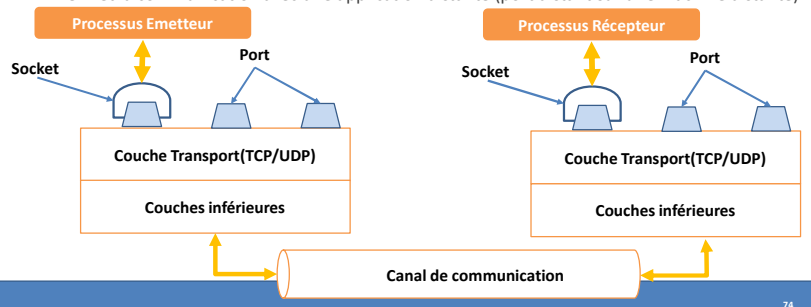
Port Number	Process Name	Protocol	Description
20	FTP-DATA	TCP	File transfer – data
21	FTP	TCP	File transfer – control
22	SSH	TCP	Secure shell
23	TELNET	TCP	Telnet
25	SMTP	TCP	Simple Mail Transfer Protocol
53	DNS	TCP and UDP	Domain Name System
69	TFTP	UDP	Trivial File Transfer Protocol
80	HTTP	TCP and UDP	Hypertext Transfer Protocol
110	POP3	TCP	Post Office Protocol 3
123	NTP	TCP	Network Time Protocol
143	IMAP	TCP	Internet Message Access Protoc
443	HTTPS	TCP	Secure implementation of HTTP

Les Sockets

❖Une socket (Prise) est le point de communication par lequel un processus peut émettre/recevoir des données à travers le réseau.

❖A chaque socket est associé un port local

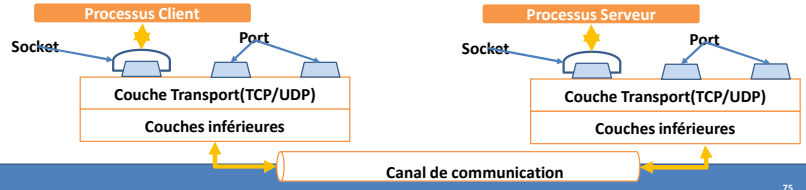
❖Permet la communication avec une application distante (port distant sur une machine distante)



Client/serveur avec Sockets

❖Architecture Client/serveur impose deux rôles différents:
- le client (demandeur de service) et le serveur (fournisseur de service)
- Possibilité que les processus communiquant jouent un autre rôle ou bien les 2 en même temps

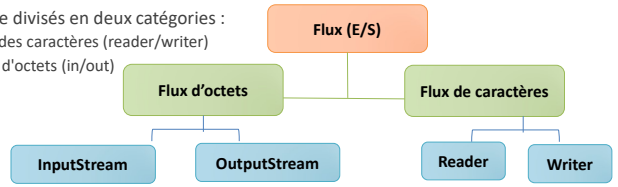
❖Différenciation pour plusieurs raisons:
- Identification : on doit connaître précisément la localisation du serveur
  - Le serveur communique via une socket liée à un port précis : port d'écoute
  - L'adresse du serveur (@IP et port) est connue du client
- Dissymétrie de la communication/connexion
  - Le client initie la connexion ou la communication



Flux d'entrée/sortie

Flux d'entrée/Sortie

- Un Processus (Programme en cours d'exécution) a besoin d'échanger des données:
  - Recevoir des données d'une source
  - Envoyer des données vers un destinataire
- En java, les entrées sorties sont gérés par des objets de Flux permettant d'encapsuler les processus d'envoi et de réception **(package java.io)**
- les flux peuvent être divisés en deux catégories :
  - les flux manipulant des caractères (reader/writer)
  - Les flux manipulant d'octets (in/out)

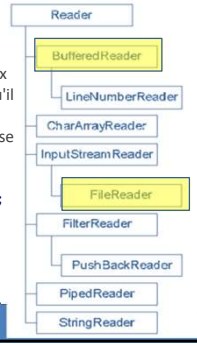


Flux de caractères

- Java gèrent les caractères avec le format Unicode (2 octets)
- Deux classes de base (abstraites) **Reader** ou **Writer**.
- De nombreuses sous-classes pour traiter les flux de caractères

- Préfixe représenter des flux ou bien des filtres:
  - **Flux**: le préfixe contient la source ou la destination du flux
  - **Filtres**: le préfixe contient le type de traitement qu'il effectue
  - Préfixe Buffered: Améliorer les performances avec la mise en tampon des données (lues ou écrites)

```
BufferedReader fichier =
        new BufferedReader(new FileReader("source.txt"));
```

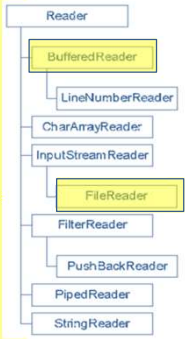


Flux de caractères (Classe Reader)

Méthodes	Rôles
boolean markSupported()	indique si le flux supporte la possibilité de marquer des positions
boolean ready()	indique si le flux est prêt à être lu
close()	ferme le flux et libère les ressources qui lui étaient associées
int read()	renvoie le caractère lu ou -1 si la fin du flux est atteinte.
int read(char[])	lire plusieurs caractères et les mettre dans un tableau de caractères
int read(char[], int, int)	lire plusieurs caractères. Elle attend en paramètre : un tableau de caractères qui contiendra les caractères lus, l'indice du premier élément du tableau qui
long skip(long)	saute autant de caractères dans le flux que la valeur fournie en paramètre. Elle renvoie le nombre de caractères sautés.
mark()	permet de marquer une position dans le flux
reset()	retourne dans le flux à la dernière position marquée

Exemple flux de caractères (Lecture)

```
public class TestBufferedReader {
    private String source;
    public TestBufferedReader(String source) {
        this.source = source;
        lecture();
    }
    public static void main(String args[]) {
        new TestBufferedReader("source.txt");
    }
    private void lecture() {
        try {
            String ligne ;
            BufferedReader fichier = new BufferedReader(new FileReader(source));
        while ((ligne = fichier.readLine()) != null) {
            System.out.println(ligne);
        }
        fichier.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```



Flux de caractères (Classe Writer)

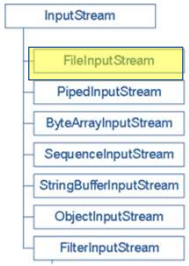
Méthodes	Rôles
close()	Ferme le flux et libère les ressources qui lui étaient associées
write(int)	Ecrire le caractère en paramètre dans le flux.
write(char[])	Ecrire le tableau de caractères en paramètre dans le flux.
write(char[], int, int)	Ecrire plusieurs caractères. Elle attend en paramètres : un tableau de caractères, l'indice du premier caractère et le nombre de caractères à écrire.
write(String)	Ecrire la chaîne de caractères en paramètre dans le flux
write(String, int, int)	Ecrire une portion d'une chaîne de caractères. Elle attend en paramètre : une chaîne de caractères, l'indice du premier caractère et le nombre de caractères à écrire.

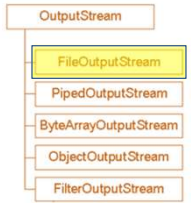
Exemple flux de caractères (Ecriture)

```
public class TestPrintWriter {
    private String destination;
    public TestPrintWriter(String destination) {
        this.destination = destination;
        ecriture();
    }
    public static void main(String args[]) {
        new TestPrintWriter("dest.txt");
    }
    private void ecriture() {
        try {
            String ligne ;
            String data = "systemes distribués";
            PrintWriter fichier = new PrintWriter(new FileWriter(destination));
            fichier.println("bonjour tout le monde");
            fichier.println("Nous sommes le "+ new Date());
            fichier.println("intitulé du module est: " + data);
            fichier.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

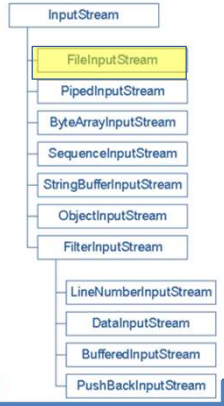
Flux d'octets

- Les classes qui gèrent les flux d'octets héritent d'une des deux classes abstraites InputStream ou OutputStream.
- Il existe de nombreuses sous-classes pour traiter les flux d'octets



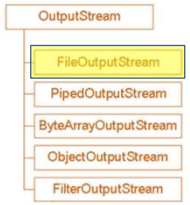


Flux d'octets (écriture sur un fichier)



Méthodes	Rôles
close()	Ferme le flux et libère les ressources qui lui étaient associées
write(int)	Ecrire l'octet en paramètre dans le flux.
write(byte[])	Ecrire le tableau d'octets à écrire en paramètre dans le flux.
write(byte[], int, int)	Ecrire plusieurs octets. En paramètres : un tableau d'octets contenant les octets à écrire, l'indice du premier et le nombre d'octets à écrire.

flux d'octets (Lecture sur un fichier)



Méthodes	Rôles
close()	Ferme le flux et libère les ressources qui lui étaient associées
int read()	envoie la valeur de l'octet lu ou -1 si la fin du flux est atteinte.
int read(byte[], int, int)	lit plusieurs octets. En paramètre : un tableau d'octets qui contiendra les octets lus, l'indice qui recevra le premier octet et le nombre d'octets à lire
int available()	retourne une estimation du nombre d'octets qu'il est encore possible de lire dans le flux

Exemple flux d'octets (L/E sur fichier)

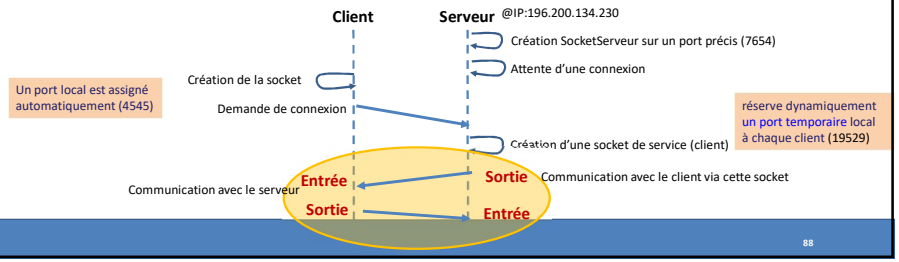
```
public class FileUtilitiesFichier {
    public static void copierfichierOctets(String source, String dest)throws IOException{
        FileInputStream in = null;
        FileOutputStream out = null;
        try {
            in = new FileInputStream(source);
            out = new FileOutputStream(dest);
            int c; //byte buffer[] = new byte[1024];
            while ((c = in.read()) != -1) //(c = fis.read(buffer)) != -1
            out.write(c); //out.write(buffer, 0, c);
        }
        finally {
            if (in != null) in.close();
            if (out != null) out.close();
        }
    }

    public static void main(String[] args)  {
        try {
            copierfichierOctets("source.txt", "copie.txt");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```


Sockets TCP

Principe de fonctionnement

❖**TCP fonctionne en mode connecté (point à point):**

1. Le Serveur enregistre son Service sur un port et se met sous écoute à l'aide **d'une Socket Serveur**
2. Le client crée une socket (port dynamique) puis demande d'ouvrir une connexion avec le serveur (@IP:port) sur sa **socket Serveur** d'écoute
3. Du coté du serveur, le service d'attente de connexion retourne une socket de service (Par client)
4. le client et le serveur communiquent en envoyant et recevant des données via leurs sockets **(I/O)**



Sockets TCP en Java

❖Classes du package java.net utilisées pour communication via TCP

Socket
Socket(String host, int port)
▪ InputStream **getInputStream()**
▪ OutputStream **getOutputStream()**
Retourne le flux d'entrée/sortie attaché à cette socket
▪ void **close()**
Ferme la socket client

ServerSocket
ServerSocket(int port)
• Socket **accept()**
renvoie la socket et établit une connexion (C/S).
• void **close()**
Ferme la socket serveur

InetAdress
InetAddress **getByName**(String host) **Static**
InetAddress **getLocalHost**()
▪ String **getHostName()**
renvoie le nom d'hôte de l'adresse IP.
▪ String **getHostAddress()**
renvoie l'adresse IP du hôte.

```
ServerSocket ss=new ServerSocket(7654);
Socket s=ss.accept();
```


InetAdress Exemple

```
public class InetAdressDemo{
	public static void main(String[] args){
		InetAddress adr;
		try {
			adr = InetAddress.getByName("ensias.um5.ac.ma");
			System.out.println("Host Name: "+adr.getHostName());
			System.out.println("IP Address: "+adr.getHostAddress());
		} catch (UnknownHostException e) {
		e.printStackTrace();
	 }
}
```

Problems @ Javadoc Declaration Search Console

<terminated> InetAdressDemo [Java Application] C:\Program Files\Java\jre1.

```
Host Name: ensias.um5.ac.ma
IP Address: 196.200.134.28
```


Interfaces de connexions réseau

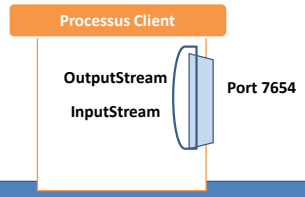
❖**NetworkInterface** classe qui encapsule les interfaces de connexions aux réseaux (cartes) et qui permet d'obtenir la liste des interfaces de connexions de la machine.

❖Chaque interface (carte réseau) peut disposer de plusieurs adresses

```
public class TestNetworkInterface {
	public static void main(String[] args) {
	try {
			TestNetworkInterface.getLocalNetworkInterface();
	} catch (Exception e) {e.printStackTrace();}
	}
	private static void getLocalNetworkInterface() throws Exception {
	    Enumeration interfaces = NetworkInterface.getNetworkInterfaces();
	    while (interfaces.hasMoreElements()) {
			NetworkInterface ni;
			Enumeration<InetAddress> adresses;
			ni = (NetworkInterface) interfaces.nextElement();
			System.out.println("Network interface : ");
			System.out.println(" nom court = " + ni.getName());
			System.out.println(" désignation = " + ni.getDisplayName());
			adresses = ni.getInetAddresses();
			while (adresses.hasMoreElements()) {
				InetAddress ia = (InetAddress) adresses.nextElement();
				System.out.println(" adresse I.P. = " + ia);
			}
		}
	}
}
```

Socket Côté client

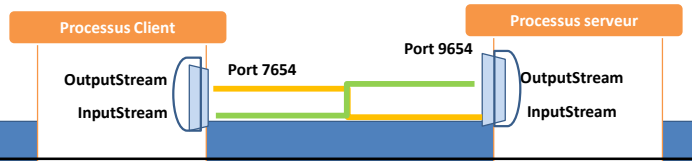
1. **Créer une socket**
   ```
   Socket client = new Socket(server, port_id);
   ```
2. **Créer les flux d'E/S pour la communication**
   ```
   Scanner in = new Scanner(client.getInputStream());
   PrintWriter out = new PrintWriter(client.getOutputStream(),true);
   ```
3. **Communiquer avec le serveur**
   ```
   out.println("Message envoyé au serveur… \n"); // envoyer message
   in.nextLine() // recevoir les données du serveur
   ```
4. **Fermer la socket**
   ```
   client.close();
   ```



Socket Côté Serveur

1. **Créer une socket serveur // port d'écoute port_id**
   ```
   ServerSocket server = new ServerSocket(port_id);
   ```
2. **Répéter**
   1. Attendre une connexion avec un client
      ```
      Socket s_com_client = Server.accept();
      ```
   1. **Traitement la connexion**
      ```
      // faire appel au traitement
      traitement(s_com_client)
      ```
4. **Fermer la socket serveur**
   ```
   Server.close();
   ```

```
Public void traitement( socket s){
  BufferedReader in = new BufferedReader(
        new InputStreamReader(s.getInputStream()));
  PrintWriter out = new
PrintWriter(s.getOutputStream(),true);
  //Communiquer avec le client
  out.println("Message envoyé au client… \n");
  in.readLine() // recevoir les données du serveur
  //Fermer la socket de traitement avec le client
  s.close();
}
```



Exercice

❖Ecrire un programme (client/Serveur) qui permet d'envoyer la date à un client qui demande de communiquer avec le serveur.

Servir plusieurs clients

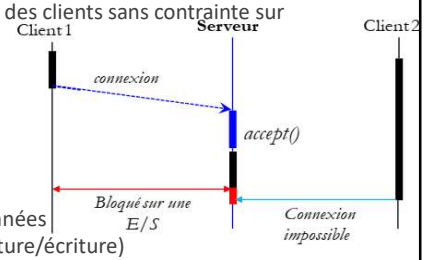
❖Application client/serveur classique
- Un serveur et Plusieurs clients (indépendants les un des autres et en parallèle)

❖Le serveur doit pouvoir répondre aux requêtes des clients sans contrainte sur l'ordre d'arrivée des requêtes

Problèmes

❖Le serveur est **bloqué** jusqu'à ce qu'un client se connecte au serveur :
Socket s = listener.accept();

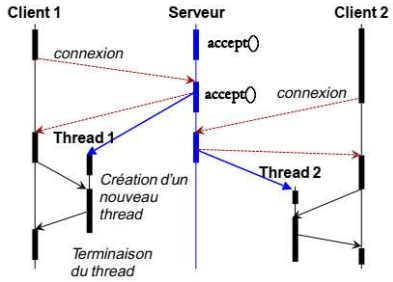
❖Le serveur et le client seront **bloqués** si les données à partir de la socket ne sont pas disponibles (lecture/écriture)
String line = in.readLine();



Servir plusieurs Clients

Solution (Multithreading)

- Un processus en attente de connexion sur le port d'écoute
- Nouvelle connexion : un nouveau processus à créer pour gérer la communication avec le nouveau client



Socket Côté Serveur (MultiThread)

1. **Créer une socket serveur // port d'écoute port_id**

```
ServerSocket server = new ServerSocket(port_id);
```

2. **Répéter**

i. Attendre une connexion avec un client

```
Socket s_com_client = Server.accept();
```

ii. **Traitement sur la connexion**

```
// créer un thread pour le nouveau client
ClientJob clientHandler= new ClientHandler(s_com_client);
clientThread.start();
```

4. **Fermer la socket serveur**

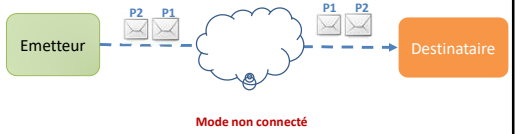
```
Server.close();
```

```
public class ClientJob extends Thread {
  private Socket socket;
  public ClientJob(Socket socket) {
        this.socket = socket;
  }
  Public void traitement( socket s){
    Scanner in = new Scanner(s.getInputStream());
   PrintWriter out;
   out = new PrintWriter(s.getOutputStream(),true);
   //Communiquer avec le client
   out.println("Message envoyé au client… \n");
   in.nextLine() // recevoir les données du serveur
   //Fermer la socket de traitement avec le client
   s.close();
  }
   Public void run( ){
         traitement(socket);
   }
}
```


Socket UDP

Socket UDP: Principe

- UDP est un protocole basé sur IP qui permet une connexion de type point à point ou de type multipoint.
- **Mode datagramme (non connecté)** : Envois de paquets de données (datagrammes)
- **Pas de fiabilité** ou de gestion de la communication:
  - Perte de paquet (sans que l'émetteur en soit informé)
  - L'ordre des paquets n'est pas garantie (routage réseau)
  - Paquets détruits si le destinataire n'est pas sur écoute (sans informer l'émetteur)
- Caractéristiques des primitives de communication:
  - Émission de paquets est **non bloquante**
  - Réception de paquets est **bloquante**

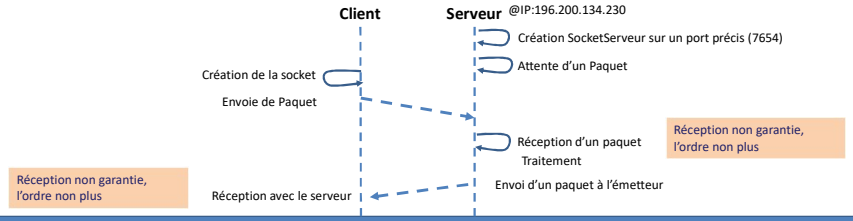


Mode non connecté

Principe de fonctionnement

❖**UDP fonctionne en mode non connecté :**

1. Le Serveur crée une **socket** sur un port précis
2. Le serveur se met en attente de réception d'un **paquet**
3. Le client crée une **socket** (port dynamique) pour accéder à la couche UDP
4. Le client envoie un **paquet** via sa socket en précisant l'adresse du destinataire (@IP:port du serveur)
   - L'adresse du client (@IP et port) est précisée dans le paquet, le serveur peut alors lui répondre



Socket UDP en Java

❖Java intègre nativement les fonctionnalités de communication réseau :**Package java.net**

❖Classes utilisées pour communication via UDP

- **InetAddress** : codage des adresses IP
- **DatagramSocket** : socket mode non connecté (UDP)
- **DatagramPacket** : paquet de données envoyé via une socket sans connexion (UDP)

DatagramPacket	DatagramSocket	InetAdress
DatagramPacket(byte[] buf, int length, InetAddress address, int port) **DatagramPacket**(byte[] buf, int length)	**DatagramSocket**(int port) **DatagramSocket**()	InetAddress **getByName**(String host) InetAddress **getLocalHost**() **Static**
▪InetAddress **getAddress()** **Si paquet à envoyer** : adresse du destinataire **Si paquet reçu** : adresse de l'émetteur ▪int **getPort()** Retourne le port enregistré dans le paquet(Emetteur ou destinataire) **LES SETTERS AUSSI!!!** ▪byte[] **getData()** Données contenues dans le paquet ▪Int **getLength()** **retourne la** longueur des données (envoyées/reçues)	• void **send(DatagramPacket** p) Envoie le paquet au destinataire spécifié dedans (@IP/port) . • void **receive(DatagramPacket p)** Reçoit un paquet. Les données reçues seront placées dans P (data, adresse émetteur) • int **getLocalPort()** Retourne le port local de lié à la socket • void **close()** Ferme la socket	▪ String **getHostName()** renvoie le nom d'hôte de l'adresse IP. ▪ String **getHostAddress()** renvoie l'adresse IP du hôte.

Socket Côté client

1. **Encapsuler l'adresse du serveur dans un objet InetAddress**

   InetAddress adr = InetAddress.getByName("196.200.134.230");
2. **Spécifier les données à envoyer**

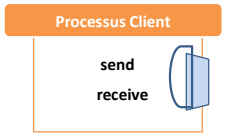
   byte[] sendData = (new String("message envoyé au serveur")).getBytes();
3. **Créer un paquet avec les données et en précisant l'adresse du serveur**

   DatagramPacket packet = new DatagramPacket(sendData, sendData.length, adr, 7676);
4. **Créer d'une socket sans la lier à un port particulier (allocation dynamique)**

   socket = new DatagramSocket();
5. **Envoyer le paquet via la socket**

   socket.send(packet);
6. **Fermer la socket**

   socket.close()



Socket Côté Serveur

1. **Créer la socket serveur liée au port 7676**

   DatagramSocket socket = new DatagramSocket(7676);
2. **Créer le tableau qui va recevoir les données**

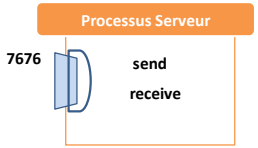
   byte[] data = new byte[50];
3. **Créer un paquet en utilisant le tableau d'octets (encore vide)**

   DatagramPacket packet = new DatagramPacket(data, data.length);
4. **Attendre la réception d'un paquet. Le paquet reçu est placé dans l'objet *packet* et les données dans *data***

   socket.receive(packet);
5. **Récupérer (et afficher) les données**

   String s = new String(packet.getData(), 0, packet.getLength());

   System.out.println(" Message reçu du client: "+s);



Socket côté serveur (suite)

- La communication se fait souvent dans les 2 sens
  - Le serveur connaît la localisation du client à travers son paquet qu'il a reçu.
- Réponse au client, coté serveur

```
System.out.println(" paquet reçu de : "+ packet.getAddress()+":"+ packet.getPort());
```

- Mettre la donnée à envoyer dans le paquet (contient déjà le couple @IP/port du client)

```
String reponse = "bien recu";
packet.setData(reponse.getBytes());
packet.setLength(reponse.length());
```

- Envoyer le paquet au client

```
socket.send(packet);
```


Résumé

- Socket TCP
- Socket UDP
- ☺ Simple à programmer
- ☹ Pas fiable
- ☹ Ne permet d'envoyer que des tableaux de byte
- ☹ les Langages de programmation de haut niveau manipulent des objets

Multicast UDP/IP

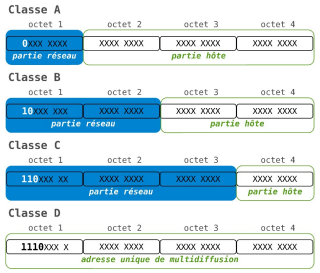
Introduction

- Communiquer des applications 1 à 1 via des sockets UDP ou TCP
- **Comment réaliser une communication 1 à plusieurs?**
- UDP offre un autre mode de communication : multicast
  - Plusieurs récepteurs pour une seule émission d'un paquet
- **Broadcast, multicast**
  - Broadcast (diffusion) : envoi de données à tous les éléments d'un réseau
  - Multicast : envoi de données à un sous-groupe de tous les éléments d'un réseau
- **Multicast IP**
  - Envoi d'un datagramme sur une adresse IP particulière
  - Plusieurs éléments lisent à cette adresse IP

Multicast

❖**Adresse IP multicast**
- Classe d'adresse IP entre 224.0.0.0 et 239.255.255.255
  - Classe D
  - Adresses entre 225.0.0.0 et 238.255.255.255 sont utilisables par un programme quelconque
  - Les autres sont réservées
- Une adresse IP multicast n'identifie pas une machine sur un réseau mais un groupe multicast

❖**Socket UDP multicast**
- Avant envoi de paquet : on doit rejoindre un groupe
  - Identifié par un couple : @IP multicast/numéro port
  - Un paquet envoyé par un membre du groupe est reçu par tous les membres de ce groupe



Classe	Bits de départ	Début	Fin
Classe A	0	0.0.0.0	126.255.255.255[3] (127 est réservé)
Classe B	10	128.0.0.0	191.255.255.255
Classe C	110	192.0.0.0	223.255.255.255
Classe D (multicast)	1110	224.0.0.0	239.255.255.255

Multicast

❖**Utilités du multicast UDP/IP**
- Évite d'avoir à créer X connexions et/ou d'envoyer X fois la même donnée à X machines différentes
- Utilisé pour diffuser des informations
- Diffusion de flux vidéos à plusieurs récepteurs
  - Chaîne de télévision, diffusion d'une conférence
  - Le même flux est envoyé à tous au même moment
- Pour récupérer des informations sur le réseau
  - 224.0.0.12 : pour localiser un serveur DHCP

❖**Limites**
- Non fiable et non connecté comme UDP

Multicast UDP en Java

❖Classe java.net.MulticastSocket

❖Hérite de la classe DatagramSocket
- Constructeurs : identiques à ceux de DatagramSocket
- public MulticastSocket() Crée une nouvelle socket en la liant à un port quelconque libre
- public MulticastSocket(int port) Crée une nouvelle socket en la liant au port précisé
  - c'est le port qui identifie le groupe de multicast

❖Gestion des groupes
- public void joinGroup(InetAddress mcastaddr) : Rejoint le groupe dont l'adresse IP multicast est passée en paramètre
- public void leaveGroup(InetAddress mcastaddr): Quitte un groupe de multicast

Multicast UDP (exemple)

❖**adresse IP multicast du groupe**
```
InetAddress group = InetAddress.getByName("228.5.6.7");
```
❖ **socket UDP multicast pour communiquer avec groupe 228.5.6.7:4000**
```
MulticastSocket socket = new MulticastSocket(4000);
```
❖ **données à envoyer**
```
byte[] data = (new String("youpi")).getBytes();
```
❖ **paquet à envoyer (en précisant le couple @IP/port du groupe)**
```
DatagramPacket packet = new DatagramPacket(data, data.length, group, 4000);
```
❖ **on joint le groupe**
```
socket.joinGroup(group);
```
❖ **on envoie le paquet**
```
socket.send(packet);
```
❖ **attend un paquet en réponse**
```
socket.receive(packet);
```
❖ **traite le résultat et quitte le groupe**
```
socket.leaveGroup(group);
```