

# ***Systemes distribués***

## ***Sockets TCP/UDP et leur mise en œuvre en Java***

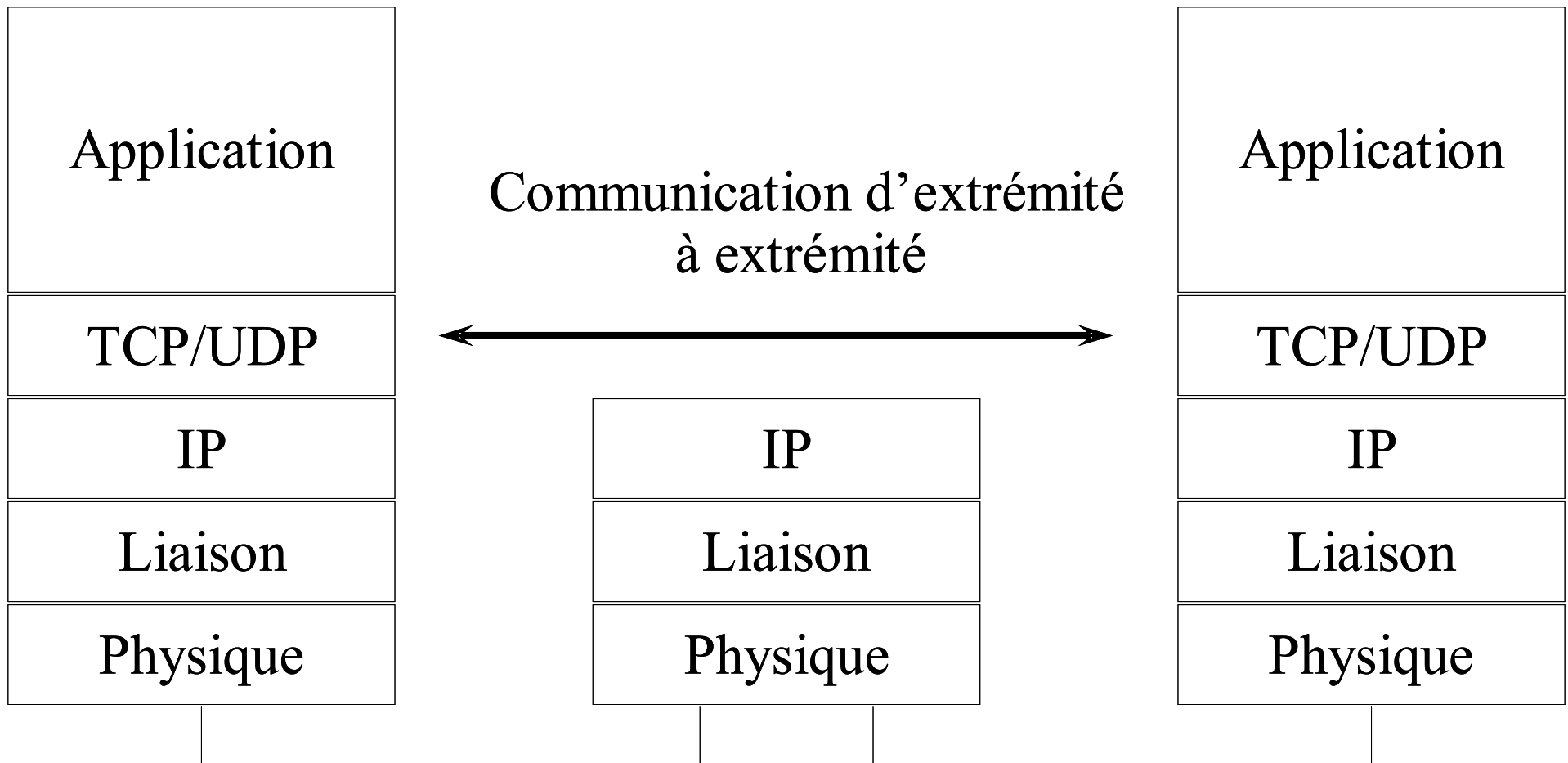
Eric Cariou

*Université de Pau et des Pays de l'Adour  
Département Informatique*

Eric.Cariou@univ-pau.fr

# Rappel sur les réseaux

- ◆ TCP ou UDP
  - ◆ Communication entre systèmes aux extrémités
  - ◆ Pas de visibilité des systèmes intermédiaires



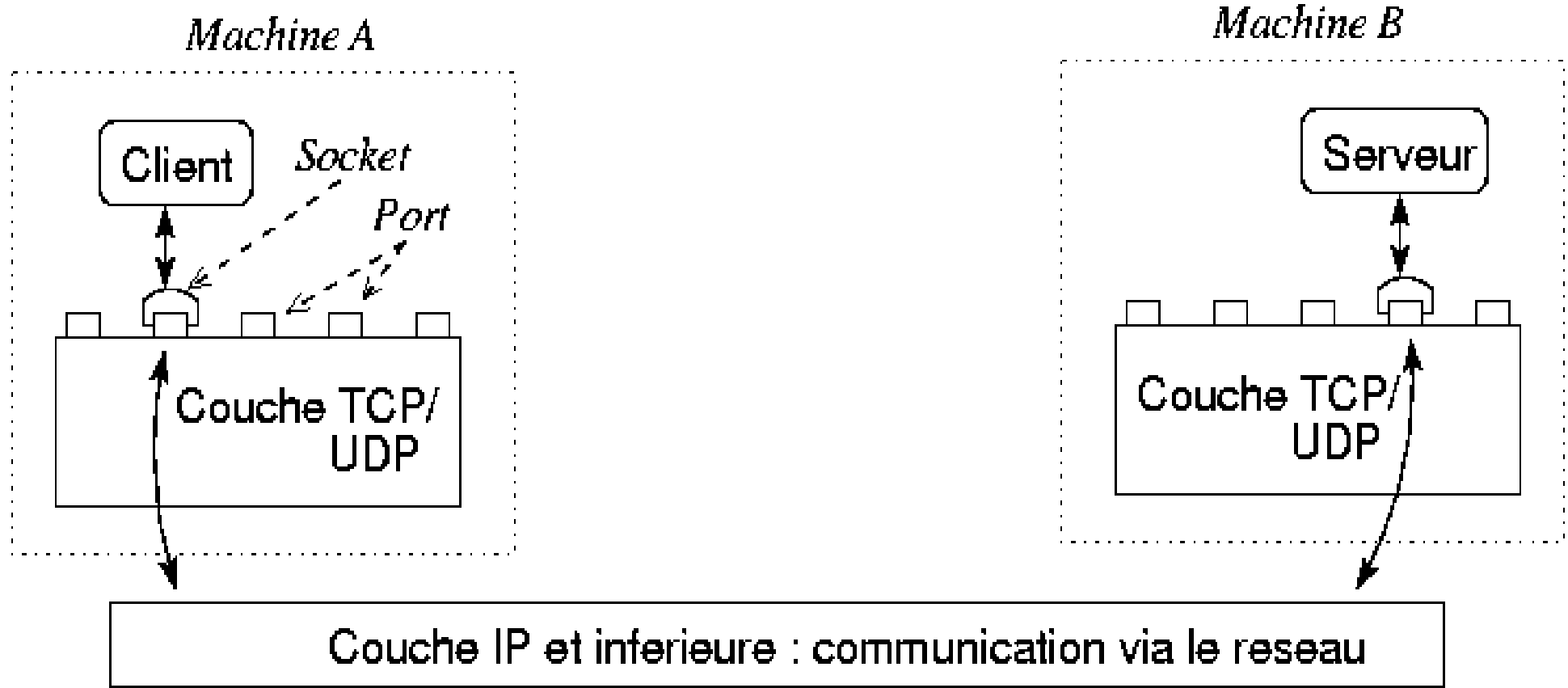
# Adressage

- ◆ Adressage pour communication entre applications
  - ◆ Adresse « réseau » application = couple de 2 informations
    - ◆ Adresse IP et numéro de port
  - ◆ Couche réseau : adresse IP
    - ◆ Ex : 192.129.12.34
  - ◆ Couche transport : numéro de port TCP ou UDP
    - ◆ Ce numéro est en entier d'une valeur quelconque
      - ◆ Ports < 1024 : réservés pour les applications ou protocoles systèmes
      - ◆ Exemple : 80 = HTTP, 21 = FTP, ...
    - ◆ Sur un port : réception ou envoi de données
  - ◆ Adresse notée : *@IP:port* ou *nomMachine:port*
    - ◆ 192.129.12.34:80 : accès au serveur Web tournant sur la machine d'adresse IP 192.129.12.34

# Sockets

- ◆ Socket : prise
  - ◆ Associée, liée à un port
    - ◆ C'est donc un point d'accès aux couches réseaux
  - ◆ Services d'émission et de réception de données sur la socket via le port
  - ◆ En mode connecté (TCP)
    - ◆ Connexion = tuyau entre 2 applications distantes
    - ◆ Une socket est un des deux bouts du tuyau
    - ◆ Chaque application a une socket locale pour gérer la communication à distance
  - ◆ Une socket peut-être liée
    - ◆ Sur un port précis à la demande du programme
    - ◆ Sur un port quelconque libre déterminé par le système

# Sockets



- ◆ Une socket est
  - ◆ Un point d'accès aux couches réseau TCP/UDP
  - ◆ Liée localement à un port
    - ◆ Adressage de l'application sur le réseau : son couple @IP:port
- ◆ Elle permet la communication avec un port distant sur une machine distante : c'est-à-dire avec une application distante

# *Client/serveur avec sockets*

- ◆ Il y a toujours différenciation entre une partie client et une partie serveur
  - ◆ Deux rôles distincts au niveau de la communication via TCP/UDP
  - ◆ Mais possibilité que les éléments communiquant jouent un autre rôle ou les 2 en même temps
- ◆ Différenciation pour plusieurs raisons
  - ◆ Identification : on doit connaître précisément la localisation d'un des 2 éléments communicants
    - ◆ Le coté serveur communique via une socket liée à un port précis : port d'écoute
  - ◆ Dissymétrie de la communication/connexion
    - ◆ Le client initie la connexion ou la communication

# ***Sockets UDP***

# *Sockets UDP : principe*

- ◆ Mode datagramme
  - ◆ Envois de paquets de données (datagrammes)
  - ◆ Pas de connexion entre parties client et serveur
  - ◆ Pas de fiabilité ou de gestion de la communication
    - ◆ Un paquet peut ne pas arrivé (perdu par le réseau)
    - ◆ Un paquet P2 envoyé après un paquet P1 peut arriver avant ce paquet P1 (selon la gestion des routes dans le réseau)
- ◆ Principe de communication
  - ◆ La partie serveur crée une socket et la lie à un port UDP particulier
  - ◆ La partie client crée une socket pour accéder à la couche UDP et la lie sur un port quelconque



# *Sockets UDP : principe*

- ◆ Principe de communication (suite)
  - ◆ Le serveur se met en attente de réception de paquet sur sa socket
  - ◆ Le client envoie un paquet via sa socket en précisant l'adresse du destinataire
    - ◆ Couple @IP/port
    - ◆ Destinataire = partie serveur
      - ◆ @IP de la machine sur laquelle tourne la partie serveur et numéro de port sur lequel est liée la socket de la partie serveur
  - ◆ Il est reçu par le serveur (sauf pb réseau)
  - ◆ Si le client envoie un paquet avant que le serveur ne soit prêt à recevoir : le paquet est perdu

# *Sockets UDP en Java*

- ◆ Java intègre nativement les fonctionnalités de communication réseau au dessus de TCP-UDP/IP
  - ◆ 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)

# Sockets UDP en Java

## ◆ Classe `InetAddress`

### ◆ Constructeurs

- ◆ Pas de constructeurs, on passe par des méthodes statiques pour créer un objet

### ◆ Méthodes

- ◆ `public static InetAddress getByName(String host) throws UnknownHostException`
  - ◆ Crée un objet `InetAddress` identifiant une machine dont le nom est passé en paramètre
  - ◆ L'exception est levée si le service de nom (DNS...) du système ne trouve pas de machine du nom passé en paramètre sur le réseau
  - ◆ Si précise une adresse IP sous forme de chaîne ("192.12.23.24") au lieu de son nom, le service de nom n'est pas utilisé
  - ◆ Une autre méthode permet de préciser l'adresse IP sous forme d'un tableau de 4 octets

# Sockets UDP en Java

## ◆ Classe `InetAddress`

### ◆ Méthodes (suite)

- ◆ `public static InetAddress getLocalHost()`  
`throws UnknownHostException`

- ◆ Retourne l'adresse IP de la machine sur laquelle tourne le programme, c'est-à-dire l'adresse IP locale

- ◆ `public String getHostName()`

- ◆ Retourne le nom de la machine dont l'adresse est codée par l'objet `InetAddress`

# Sockets UDP en Java

- ◆ **Classe** DatagramPacket
  - ◆ Structure des données en mode datagramme
  - ◆ Constructeurs
    - ◆ `public DatagramPacket(byte[] buf, int length)`
      - ◆ Création d'un paquet pour recevoir des données (sous forme d'un tableau d'octets)
      - ◆ Les données reçues seront placées dans `buf`
      - ◆ `length` précise la taille max de données à lire
        - ◆ Ne pas préciser une taille plus grande que celle du tableau
        - ◆ En général, `length = taille de buf`
    - ◆ Variante du constructeur : avec un offset pour ne pas commencer au début du tableau

# Sockets UDP en Java

## ◆ Classe DatagramPacket

### ◆ Constructeurs (suite)

- ◆ `public DatagramPacket(byte[] buf, int length, InetAddress address, int port)`
  - ◆ Création d'un paquet pour envoyer des données (sous forme d'un tableau d'octets)
  - ◆ `buf` : contient les données à envoyer
  - ◆ `length` : longueur des données à envoyer
    - ◆ Ne pas préciser une taille supérieure à celle de `buf`
  - ◆ `address` : adresse IP de la machine destinataire des données
  - ◆ `port` : numéro de port distant (sur la machine destinataire) où envoyer les données

# Sockets UDP en Java

## ◆ Classe DatagramPacket

### ◆ Méthodes « get »

◆ `InetAddress getAddress()`

◆ Si paquet à envoyer : adresse de la machine destinataire

◆ Si paquet reçu : adresse de la machine qui a envoyé le paquet

◆ `int getPort()`

◆ Si paquet à envoyer : port destinataire sur la machine distante

◆ Si paquet reçu : port utilisé par le programme distant pour envoyer le paquet

◆ `byte[] getData`

◆ Données contenues dans le paquet

◆ `int getLength()`

◆ Si paquet à envoyer : longueur des données à envoyer

◆ Si paquet reçu : longueur des données reçues

# *Sockets UDP en Java*

## ◆ Classe DatagramPacket

### ◆ Méthodes « set »

- ◆ `void setAddress(InetAddress adr)`

- ◆ Positionne l'adresse IP de la machine destinataire du paquet

- ◆ `void setPort(int port)`

- ◆ Positionne le port destinataire du paquet pour la machine distante

- ◆ `void setData(byte[] data)`

- ◆ Positionne les données à envoyer

- ◆ `int setLength(int length)`

- ◆ Positionne la longueur des données à envoyer



# *Sockets UDP en Java*

- ◆ Classe `DatagramPacket`, complément sur les tailles des données envoyées
- ◆ Java n'impose aucune limite en taille pour les tableaux d'octets circulant dans les paquets UDP, mais
  - ◆ Pour tenir dans un seul datagramme IP, le datagramme UDP ne doit pas contenir plus de 65467 octets de données
    - ◆ Un datagramme UDP est rarement envoyé via plusieurs datagrammes IP
  - ◆ Mais en pratique : il est conseillé de ne pas dépasser 8176 octets
    - ◆ Car la plupart des systèmes limitent à 8 Ko la taille des datagrammes UDP
  - ◆ Pour être certain de ne pas perdre de données : 512 octets max
  - ◆ Si datagramme UDP trop grand : les données sont tronquées
- ◆ Si tableau d'octets en réception est plus petit que les données envoyées
  - ◆ Les données reçues sont généralement tronquées

# Sockets UDP en Java

- ◆ **Classe** `DatagramSocket`
  - ◆ Socket en mode datagramme
  - ◆ Constructeurs
    - ◆ `public DatagramSocket() throws SocketException`
      - ◆ Crée une nouvelle socket en la liant à un port quelconque libre
      - ◆ Exception levée en cas de problème (a priori il doit pas y en avoir)
    - ◆ `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

# Sockets UDP en Java

## ◆ Classe DatagramSocket

### ◆ Méthodes d'émission/réception de paquet

◆ `public void send(DatagramPacket p)`  
`throws IOException`

◆ Envoie le paquet passé en paramètre. Le destinataire est identifié par le couple @IP/port précisé dans le paquet

◆ Exception levée en cas de problème d'entrée/sortie

◆ `public void receive(DatagramPacket p)`  
`throws IOException`

◆ Reçoit un paquet de données

◆ Bloquant tant qu'un paquet n'est pas reçu

◆ Quand paquet arrive, les attributs de `p` sont modifiés

◆ Les données reçues sont copiées dans le tableau passé en paramètre lors de la création de `p` et sa longueur est positionnée avec la taille des données reçues

◆ Les attributs d'@IP et de port de `p` contiennent l'@IP et le port de la socket distante qui a émis le paquet

# Sockets UDP en Java

- ◆ **Classe** DatagramSocket
  - ◆ **Autres méthodes**
    - ◆ `public void close()`
      - ◆ Ferme la socket et libère le port à laquelle elle était liée
    - ◆ `public int getLocalPort()`
      - ◆ Retourne le port local sur lequel est liée la socket
  - ◆ **Possibilité de créer un canal (mais toujours en mode non connecté)**
    - ◆ Pour restreindre la communication avec un seul destinataire distant
    - ◆ Car par défaut peut recevoir sur la socket des paquets venant de n'importe où

# Sockets UDP en Java

- ◆ Classe `DatagramSocket`
- ◆ Réception de données : via méthode `receive`
  - ◆ Méthode bloquante sans contrainte de temps : peut rester en attente indéfiniment si aucun paquet n'est jamais reçu
- ◆ Possibilité de préciser un délai maximum d'attente
  - ◆ `public void setSoTimeout(int timeout) throws SocketException`
  - ◆ L'appel de la méthode `receive` sera bloquante pendant au plus `timeout` millisecondes
  - ◆ Une méthode `receive` se terminera alors de 2 façons
    - ◆ Elle retourne normalement si un paquet est reçu en moins du temps positionné par l'appel de `setSoTimeout`
    - ◆ L'exception `SocketTimeoutException` est levée pour indiquer que le délai s'est écoulé avant qu'un paquet ne soit reçu
      - ◆ `SocketTimeoutException` est une sous-classe de `IOException`

# Sockets UDP Java – exemple coté client

```
◆ InetAddress adr;  
DatagramPacket packet;  
DatagramSocket socket;  
  
// adr contient l'@IP de la partie serveur  
adr = InetAddress.getByName("scinfr222");  
  
// données à envoyer : chaîne de caractères  
byte[] data = (new String("youpi")).getBytes();  
  
// création du paquet avec les données et en précisant l'adresse du serveur  
// (@IP et port sur lequel il écoute : 7777)  
packet = new DatagramPacket(data, data.length, adr, 7777);  
  
// création d'une socket, sans la lier à un port particulier  
socket = new DatagramSocket();  
  
// envoi du paquet via la socket  
socket.send(packet);
```

# Sockets UDP Java – exemple coté serveur

```
◆ DatagramSocket socket;  
  DatagramPacket packet;
```

```
// création d'une socket liée au port 7777
```

```
DatagramSocket socket = new DatagramSocket(7777);
```

```
// tableau de 15 octets qui contiendra les données reçues
```

```
byte[] data = new byte[15];
```

```
// création d'un paquet en utilisant le tableau d'octets
```

```
packet = new DatagramPacket(data, data.length);
```

```
// attente de la réception d'un paquet. Le paquet reçu est placé dans
```

```
// packet et ses données dans data.
```

```
socket.receive(packet);
```

```
// récupération et affichage des données (une chaîne de caractères)
```

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

```
System.out.println(" reçu : "+chaine);
```

# *Sockets UDP en Java – exemple suite*

- ◆ La communication se fait souvent dans les 2 sens
  - ◆ Le serveur doit donc connaître la localisation du client
  - ◆ Elle est précisée dans le paquet qu'il reçoit du client
- ◆ Réponse au client, coté serveur

```
◆ System.out.println(" ca vient de :  
  "+packet.getAddress()+" : "+ packet.getPort());  
  
// on met une nouvelle donnée dans le paquet  
// (qui contient donc le couple @IP/port de la socket coté client)  
packet.setData((new String("bien reçu")).getBytes());  
  
// on envoie le paquet au client  
socket.send(packet);
```



# *Sockets UDP en Java – exemple suite*

## ◆ Réception réponse du serveur, coté client

```
// attente paquet envoyé sur la socket du client  
socket.receive(packet);
```

```
// récupération et affichage de la donnée contenue dans le paquet  
String chaine = new String(packet.getData(), 0,  
                             packet.getLength());  
System.out.println(" reçu du serveur : "+chaine);
```

# *Critique sockets UDP*

- ◆ Avantages
  - ◆ Simple à programmer (et à appréhender)
- ◆ Inconvénients
  - ◆ Pas fiable
  - ◆ Ne permet d'envoyer que des tableaux de byte

# *Structure des données échangées*

- ◆ Format des données à transmettre
  - ◆ Très limité a priori : tableaux de byte
    - ◆ Et attention à la taille réservée : si le récepteur réserve un tableau trop petit par rapport à celui envoyé, une partie des données est perdue
  - ◆ Doit donc pouvoir convertir
    - ◆ Un objet quelconque en `byte[]` pour l'envoyer
    - ◆ Un `byte[]` en un objet d'un certain type après réception
- ◆ Deux solutions
  - ◆ Créer les méthodes qui font cela : lourd et dommage de faire des tâches de si « bas-niveau » avec un langage évolué comme Java
  - ◆ Utiliser les flux Java pour conversion automatique

# *Conversion Object <-> byte[]*

- ◆ Pour émettre et recevoir n'importe quel objet via des sockets UDP

- ◆ En écriture : conversion de Object en byte[]

```
ByteArrayOutputStream byteStream =  
    new ByteArrayOutputStream( );  
ObjectOutputStream objectStream =  
    new ObjectOutputStream(byteStream) ;  
objectStream.writeObject(object) ;  
byte[] byteArray = byteStream.toByteArray( ) ;
```

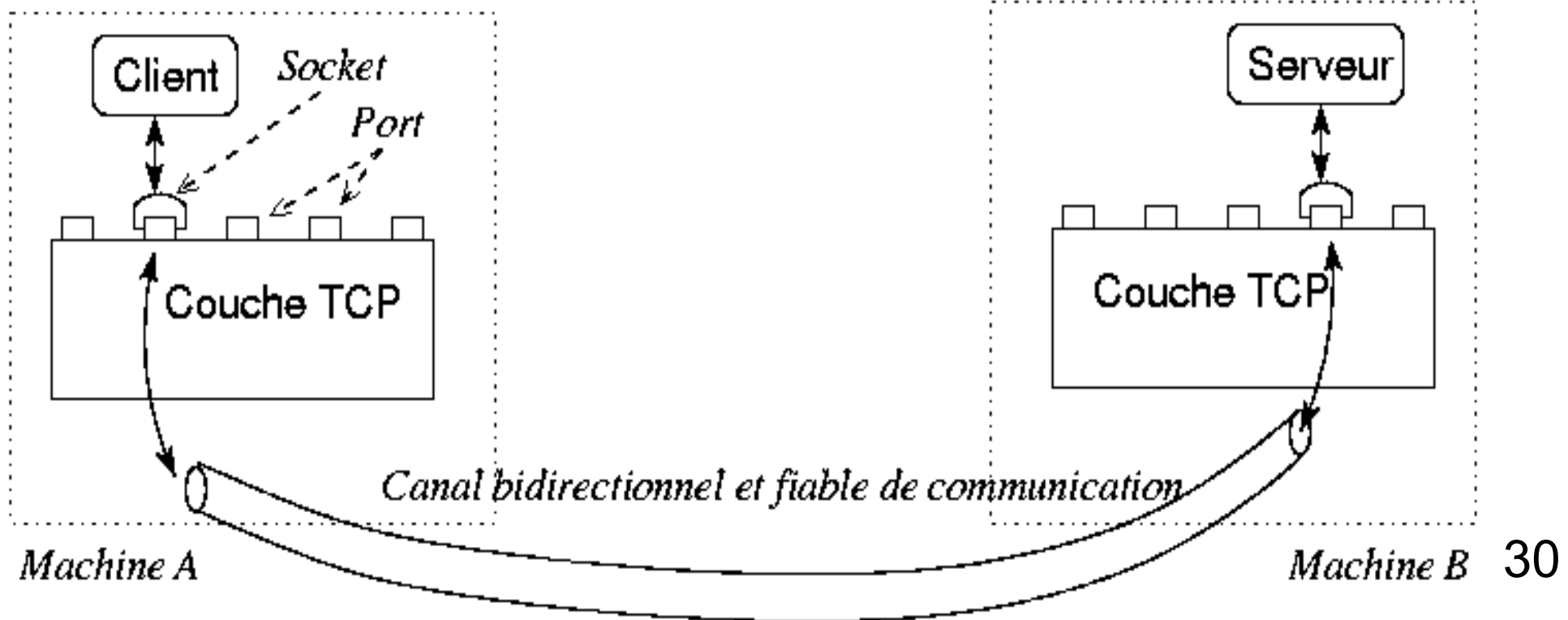
- ◆ En lecture : conversion de byte[] en Object

```
ByteArrayInputStream byteStream =  
    new ByteArrayInputStream(byteArray) ;  
ObjectInputStream objectStream =  
    new ObjectInputStream(byteStream) ;  
object = objectStream.readObject( ) ;
```

# ***Sockets TCP***

# Sockets TCP : principe

- ◆ Fonctionnement en mode connecté
  - ◆ Données envoyées dans un « tuyau » et non pas par paquet
    - ◆ Flux de données
    - ◆ Correspond aux flux Java dans la mise en oeuvre Java des sockets TCP
  - ◆ Fiable : la couche TCP assure que
    - ◆ Les données envoyées sont toutes reçues par la machine destinataire
    - ◆ Les données sont reçues dans l'ordre où elles ont été envoyées



# *Sockets TCP : principe*

- ◆ Principe de communication
  - ◆ Le serveur lie une socket d'écoute sur un certain port bien précis et appelle un service d'attente de connexion de la part d'un client
  - ◆ Le client appelle un service pour ouvrir une connexion avec le serveur
    - ◆ Il récupère une socket (associée à un port quelconque par le système)
  - ◆ Du côté du serveur, le service d'attente de connexion retourne une socket de service (associée à un port quelconque)
    - ◆ C'est la socket qui permet de dialoguer avec ce client
  - ◆ Comme avec sockets UDP : le client et le serveur communiquent en envoyant et recevant des données via leur socket

# *Sockets TCP en Java*

- ◆ Classes du package `java.net` utilisées pour communication via TCP
  - ◆ `InetAddress` : codage des adresses IP
    - ◆ Même classe que celle décrite dans la partie UDP et usage identique
  - ◆ `Socket` : socket mode connecté
  - ◆ `ServerSocket` : socket d'attente de connexion du côté server



# Sockets TCP en Java

## ◆ Classe Socket

### ◆ Socket mode connecté

### ◆ Constructeurs

- ◆ `public Socket(InetAddress address, int port)`  
`throws IOException`
  - ◆ Crée une socket locale et la connecte à un port distant d'une machine distante identifié par le couple `address/port`
- ◆ `public Socket(String address, int port)`  
`throws IOException, UnknownHostException`
  - ◆ Idem mais avec nom de la machine au lieu de son adresse IP codée
  - ◆ Lève l'exception `UnknownHostException` si le service de nom ne parvient pas à identifier la machine
- ◆ Variante de ces 2 constructeurs pour préciser en plus un port local sur lequel sera liée la socket créée

# *Sockets TCP en Java*

## ◆ Classe Socket

### ◆ Méthodes d'émission/réception de données

- ◆ Contrairement aux sockets UDP, les sockets TCP n'offre pas directement de services pour émettre/recevoir des données

- ◆ On récupère les flux d'entrée/sorties associés à la socket

- ◆ `OutputStream getOutputStream()`

- ◆ Retourne le flux de sortie permettant d'envoyer des données via la socket

- ◆ `InputStream getInputStream()`

- ◆ Retourne le flux d'entrée permettant de recevoir des données via la socket

### ◆ Fermeture d'une socket

- ◆ `public close()`

- ◆ Ferme la socket et rompt la connexion avec la machine distante

# Sockets TCP en Java

## ◆ Classe Socket

### ◆ Méthodes « get »

◆ `int getPort()`

◆ Renvoie le port distant avec lequel est connecté la socket

◆ `InetAddress getAddress()`

◆ Renvoie l'adresse IP de la machine distante

◆ `int getLocalPort()`

◆ Renvoie le port local sur lequel est liée la socket

◆ `public void setSoTimeout(int timeout)  
throws SocketException`

◆ Positionne l'attente maximale en réception de données sur le flux d'entrée de la socket

◆ Si temps dépassé lors d'une lecture : exception `SocketTimeoutException` est levée

◆ Par défaut : temps infini en lecture sur le flux

# Sockets TCP en Java

## ◆ Classe ServerSocket

- ◆ Socket d'attente de connexion, coté serveur uniquement

## ◆ Constructeurs

- ◆ `public ServerSocket(int port) throws IOException`
  - ◆ Crée une socket d'écoute (d'attente de connexion de la part de clients)
  - ◆ La socket est liée au port dont le numéro est passé en paramètre
    - ◆ L'exception est levée notamment si ce port est déjà lié à une socket

## ◆ Méthodes

- ◆ `Socket accept() throws IOException`
  - ◆ Attente de connexion d'un client distant
  - ◆ Quand connexion est faite, retourne une socket permettant de communiquer avec le client : socket de service
- ◆ `void setSoTimeout(int timeout) throws SocketException`
  - ◆ Positionne le temps maximum d'attente de connexion sur un accept
    - ◆ Si temps écoulé, l'accept lève l'exception `SocketTimeoutException`
  - ◆ Par défaut, attente infinie sur l'accept

# *Sockets TCP Java – exemple coté client*

- ◆ Même exemple qu'avec UDP
  - ◆ Connexion d'un client à un serveur
  - ◆ Envoi d'une chaîne par le client et réponse sous forme d'une chaîne par le serveur
- ◆ Coté client

```
// adresse IP du serveur
InetAddress adr = InetAddress.getByName("scinfr222");

// ouverture de connexion avec le serveur sur le port 7777
Socket socket = new Socket(adr, 7777);
```

# *Sockets TCP Java – exemple coté client*

## ◆ Coté client (suite)

```
// construction de flux objets à partir des flux de la socket
ObjectOutputStream output =
    new ObjectOutputStream(socket.getOutputStream());
ObjectInputStream input =
    new ObjectInputStream(socket.getInputStream());

// écriture d'une chaîne dans le flux de sortie : c'est-à-dire envoi de
// données au serveur
output.writeObject(new String("youpi"));

// attente de réception de données venant du serveur (avec le readObject)
// on sait qu'on attend une chaîne, on peut donc faire un cast directement
String chaine = (String)input.readObject();
System.out.println(" reçu du serveur : "+chaine);
```

# *Sockets TCP Java – exemple coté serveur*

```
◆ // serveur positionne sa socket d'écoute sur le port local 7777
  ServerSocket serverSocket = new ServerSocket(7777);

// se met en attente de connexion de la part d'un client distant
Socket socket = serverSocket.accept();

// connexion acceptée : récupère les flux objets pour communiquer
// avec le client qui vient de se connecter
ObjectOutputStream output =
    new ObjectOutputStream(socket.getOutputStream());
ObjectInputStream input =
    new ObjectInputStream(socket.getInputStream());

// attente les données venant du client
String chaine = (String)input.readObject();
System.out.println(" reçu : "+chaine);
```

# Sockets TCP Java – exemple coté serveur

## ◆ Coté serveur (suite)

```
// affiche les coordonnées du client qui vient de se connecter
System.out.println(" ca vient de : "
    +socket.getInetAddress()+" : "+socket.getPort());

// envoi d'une réponse au client
output.writeObject(new String("bien reçu")) ;
```

## ◆ Quand manipule des flux d'objets

- ◆ Souvent utile de vérifier le type de l'objet reçu

- ◆ Utilise instanceof

### ◆ Exemple

- ◆ 

```
String chaine;
Personne pers;
Object obj = input.readObject();
if (obj instanceof String) chaine = (String)obj;
if (obj instanceof Personne) pers = (Personne)obj;
```



# Sockets TCP

## ◆ Critique sockets TCP

### ◆ Avantages

- ◆ Niveau d'abstraction plus élevé qu'avec UDP
  - ◆ Mode connecté avec phase de connexion explicite
  - ◆ Flux d'entrée/sortie
- ◆ Fiable

### ◆ Inconvénients

- ◆ Plus difficile de gérer plusieurs clients en même temps
  - ◆ Nécessite du parallélisme avec des threads (voir suite cours)
  - ◆ Mais oblige une bonne structuration coté serveur

# *Sockets UDP ou TCP ?*

- ◆ Choix entre UDP et TCP
  - ◆ A priori simple
    - ◆ TCP est fiable et mieux structuré
  - ◆ Mais intérêt tout de même pour UDP dans certains cas
    - ◆ Si la fiabilité n'est pas essentielle
    - ◆ Si la connexion entre les 2 applications n'est pas utile
    - ◆ Exemple
      - ◆ Un thermomètre envoie toutes les 5 secondes la température de l'air ambiant à un afficheur distant
        - ◆ Pas grave de perdre une mesure de temps en temps
        - ◆ Pas grave d'envoyer les mesures même si l'afficheur est absent

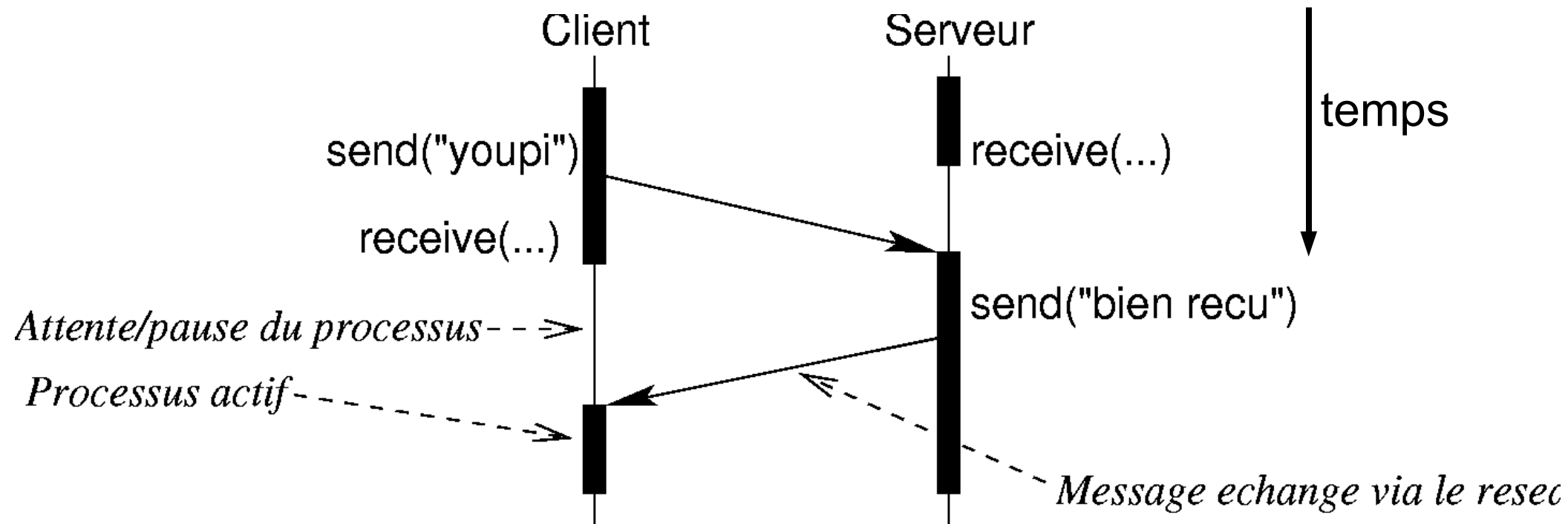
# *Sockets UDP ou TCP ?*

- ◆ Exemple de protocole utilisant UDP : NFS
  - ◆ Network File System (NFS)
    - ◆ Accès à un système de fichiers distant
  - ◆ A priori TCP mieux adapté car besoin de fiabilité lors des transferts des fichiers, mais
    - ◆ NFS est généralement utilisé au sein d'un réseau local
      - ◆ Peu de pertes de paquets
    - ◆ UDP est plus basique et donc plus rapide
      - ◆ TCP gère un protocole assurant la fiabilité impliquant de nombreux échanges supplémentaires entre les applications (envoi d'acquittement...)
      - ◆ Peu de perte de paquet en UDP en local : peu directement gérer la fiabilité au niveau NFS ou applicatif et c'est moins coûteux en temps
  - ◆ Dans ce contexte, il n'est pas pénalisant d'utiliser UDP au lieu de TCP pour NFS
    - ◆ NFS fonctionne sur ces 2 couches

# ***Serveur en gestion multi-clients***

# Concurrence

- ◆ Par principe, les éléments distants communiquants sont actifs en parallèle
- ◆ Plusieurs processus concurrents
- ◆ Avec processus en pause lors d'attente de messages
- ◆ Exemple de flux d'exécution pour notre exemple de client/serveur précédent



# *Sockets TCP – gestion plusieurs clients*

- ◆ Particularité coté serveur en TCP
  - ◆ Une socket d'écoute sert à attendre les connexions des clients
  - ◆ A la connexion d'un client, une socket de service est initialisée pour communiquer avec ce client
- ◆ Communication avec plusieurs clients pour le serveur
  - ◆ Envoi de données à un client
    - ◆ UDP : on précise l'adresse du client dans le paquet à envoyer
    - ◆ TCP : utilise la socket correspondant au client
  - ◆ Réception de données venant d'un client quelconque
    - ◆ UDP : se met en attente d'un paquet et regarde de qui il vient
    - ◆ TCP : doit se mettre en attente de données sur toutes les sockets actives

# ***Sockets TCP – gestion plusieurs clients***

- ◆ Communication avec plusieurs clients (suite)
  - ◆ Contrainte
    - ◆ Lecture sur une socket : opération bloquante
      - ◆ Tant que des données ne sont pas reçues
    - ◆ Attente de connexion : opération bloquante
      - ◆ Jusqu'à la prochaine connexion d'un client distant
  - ◆ Avec un seul flot d'exécution (processus/thread)
    - ◆ Si ne sait pas quel est l'ordonnancement des arrivées des données des clients ou de leur connexion au serveur
      - ◆ Impossible à gérer
  - ◆ Donc nécessité de plusieurs processus ou threads
    - ◆ Un processus en attente de connexion sur le port d'écoute
    - ◆ Nouvelle connexion : un nouveau processus est créé pour gérer la communication avec le nouveau client

# Sockets TCP – gestion plusieurs clients

- ◆ Boucle de fonctionnement général d'un serveur pour gérer plusieurs clients

- ◆ while(true)  
    socketClient = acceptConnection()  
    newThread(socketClient)

- ◆ Exemple avec 2 clients ->

