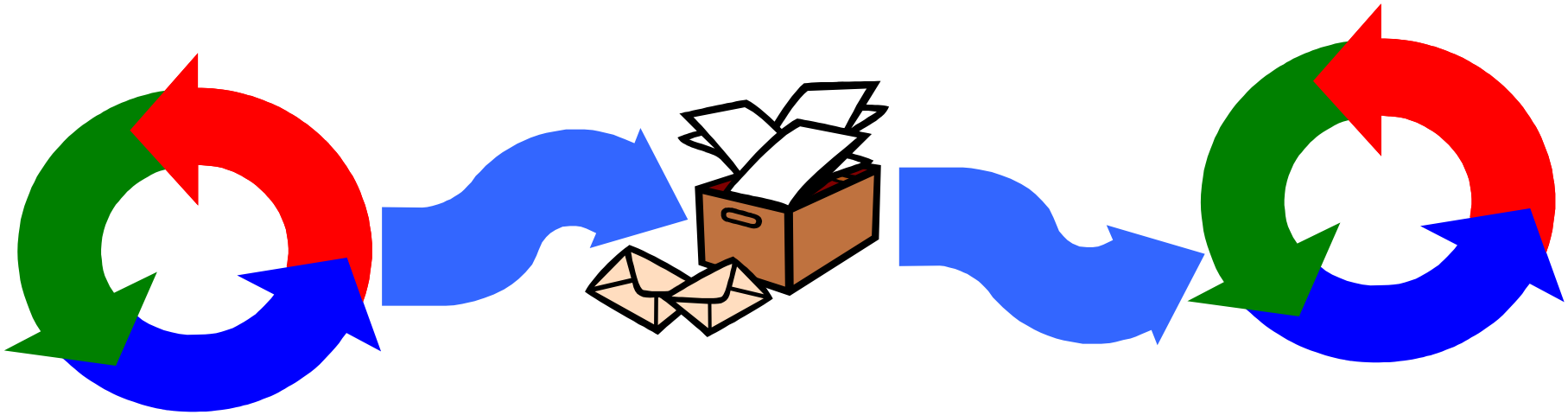
Systèmes Distribués : communication

Synchronisation par passage de messages :
application sur les « socket » de Java



Plan du cours

- **Introduction**
  - Définitions
  - Problématique
  - Architectures de distribution
- **Distribution intra-applications**
  - Notion de processus
  - Programmation multi-thread
- **Distribution inter-applications et inter-machines**
  - sockets
  - middlewares par appel de procédures distantes
  - middlewares par objets distribués (Java RMI, CORBA)
- **Conclusion**

Introduction : pourquoi la communication?

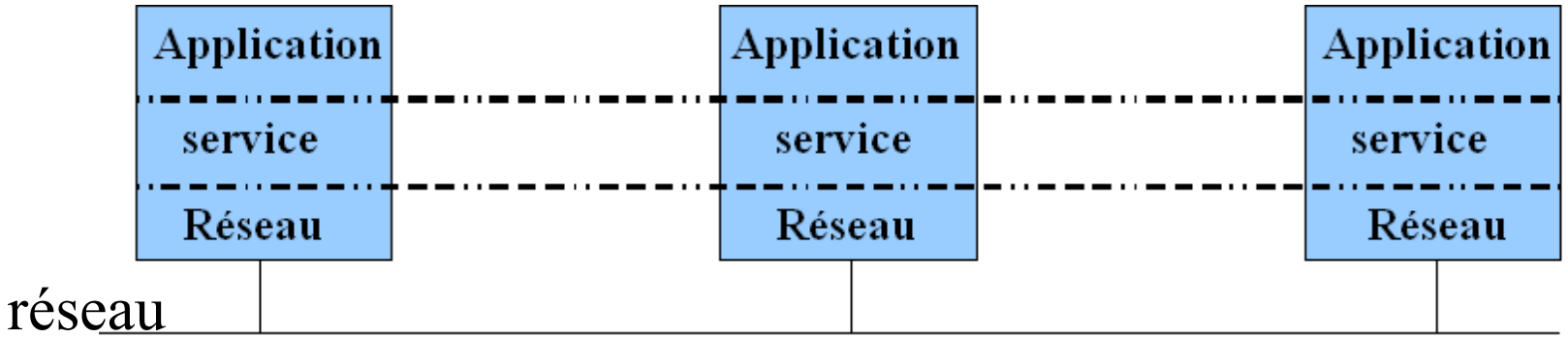
- **On a vu la synchronisation entre processus par partage de variables.**
  - La cohérence des états par exclusion mutuelle
  - Détection de blocage....
- **Mais quand les processus se trouvent sur deux sites distincts la synchronisation se fait par envoi et réception de messages.**
- **Les messages contiennent des valeurs qui influencent le déroulement de l'exécution du récepteur.**

Modèle de répartition

- **Le modèle de répartition considéré dans ce cours est composé par**
- **Un ensemble de sites**
  - Chaque site possède sa propre mémoire non accessible aux autres sites
  - Chaque site dispose d'un identifiant unique (IP ou adresse MAC)
- **Les lignes de communication**
  - Bi-point i.e reliant deux sites
  - Bi-directionnelle i.e l'échange est possible dans les deux directions
  - Chaque direction est appelée canal
  - On considère que le graphe résultant comme une clique i.e chaque deux sites peuvent physiquement échanger des données.

Les sites : modèle en couches

- **Nous nous intéressons à chaque site qu'en tant que composante d'une application répartie.**
- **Conçue selon un modèle en couches.**



- **Chaque couche fournit un ensemble de services aux couches supérieures.**
- **Ce qui a pour objectif de masquer les difficultés d'implémentation**

Les couches

- **Le réseau et la couche réseau**
  - Un canal de communication entre deux sites a les propriétés :
    - Les données ne sont pas altérées
    - Les messages ne sont pas perdus (pas toujours vrai on va la relaxer)
    - Le canal est FIFO les messages arrivent dans l'ordre de leurs envois.
  - Le réseau est considéré comme :
    - Asynchrone : le délai de transit est indéfini (le cas considéré ici)
    - Synchrone : le délai est borné et connu par le concepteur.
- **Couche services**
  - C'est une API qui offre un certain nombre de primitives sous forme d'API à la couche application.
  - Les primitives d'envoi et de réception de messages
  - Utilise les primitives de l'API et de la couche réseau pour offrir ses services
- **La chouche application**
  - => c'est la couche avec celle du service qui nous intéresse dans ce cours

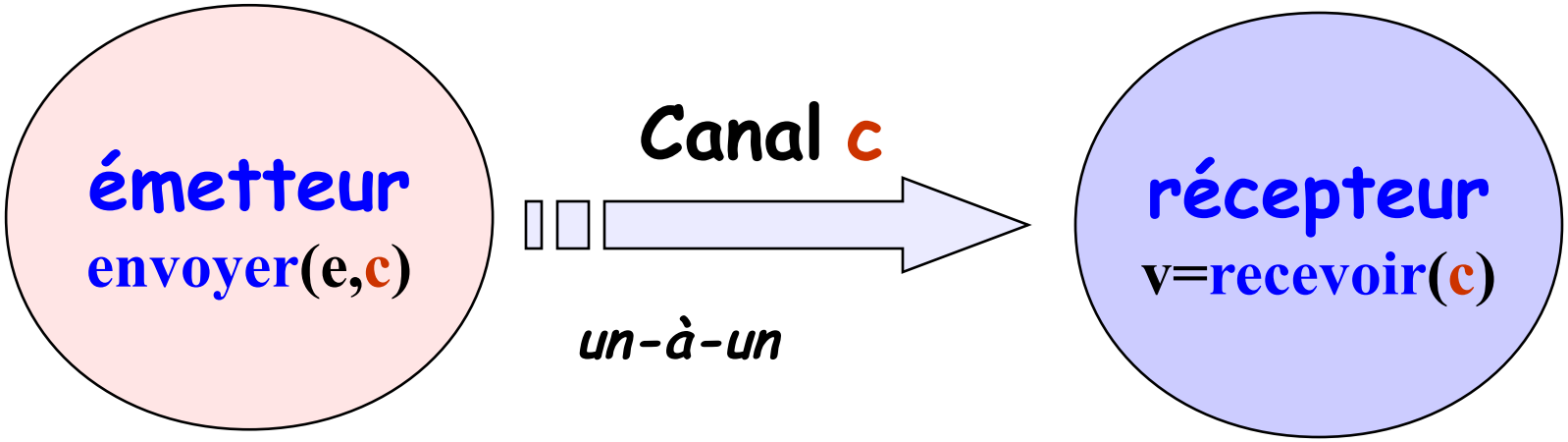
La communication

- **Plusieurs définitions mais on va garder une définition du point de vue de la couche application.**
- **Une communication est une suite de trois actions**
  - L'envoi
  - Le transport (ne nous intéresse pas)
  - La réception
- **On s'intéresse à la sémantique des deux actions de communication sur chaque site et sur l'apllication répartie**
- **Ici on va voir les modèles les plus connus et utilisés**
  - Communication synchrone
  - Communication asynchrone
  - Communication par rendez-vous
- **Attention à ne pas confondre avec le synchrone et asynchrone du réseau**

Communication synchrone

- **La communication est dite synchrone quand les actions d'envoi et de réception ne sont possibles que si :**
  - L'émetteur se trouve dans un état d'envoi.
  - Et le récepteur dans un état de reception.
- **La communication orale doit être synchrone car celui qui parle ne parle que si il sait que son interlocuteur est dans un état où il l'écoute.**
- **Une autre façon de modéliser la communication synchrone est de la considérer comme une émulation de l'opération d'affectation distribuée.**
- **Mettre une valeur locale dans une variable distante.**
- **=> D'où la modélisation en utilisant la notion de canal de communication.**

Communication synchrone – notion de canal



envoyer(e,c) - envoie la valeur de l'expression *e* sur le canal *c*. Le processus appelle l'opération envoyer et se **bloque** jusqu'à réception du message par le récepteur.

v = recevoir(*c*) - recevoir une valeur dans la variable *v* à partir du canal *c*. le processus qui appelle ***recevoir*** se **bloque** jusqu'à ce qu'une valeur soit envoyée sur le canal.

affectation distribuée $v = e$

Enrichissons notre langage avec les actions d'envoi et de réception

- **On va rajouter dans notre langage :**

EXPR ::= CONSTANTE | **CANAL** | VARIABLE
| EXPR+EXPR | EXPR*EXPR
| EXPR/EXPR | EXPR-EXPR
TEST ::= EXPR==EXPR | EXPR < EXPR | EXPR > EXPR | TEST & TEST | TEST OU TEST | ! TEST
INSTR ::= VARIABLE=EXPR | **CANAL!EXP** | **CANAL?VARIABLE**

BLOCK ::= |
INSTR;BLOCK
| if TEST
then BLOCK
else BLOCK
| while TEST
BLOCK ;

P1

```
int x=1;
c!x;
x+=1;
print(x);
```

x=

P2

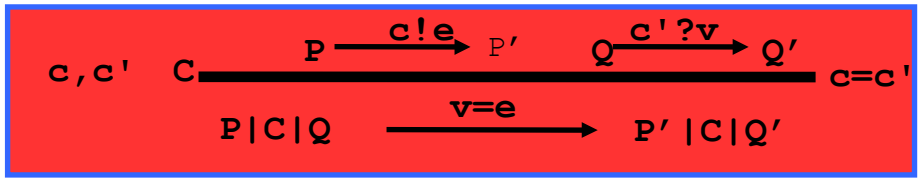
```
int y=0;
c?y;
print(y)
```

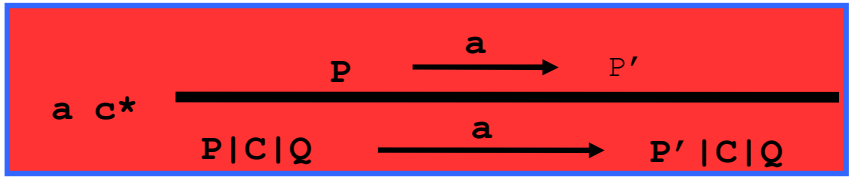
y=

```
Canal c;
new P1(c).start();
new P2(c).start();
```

La sémantique des actions de communication

- **Localement d'abord**
  - Envoi : $c!e;P \text{---}^{c!e}\text{---}> P$
  - Réception $c?v;P \text{--}^{c?v}\text{--}> P$
- **Sémantique dans le cas de la concurrence**
  - On introduit un opérateur de composition $P|C|Q$
    - P et Q deux processus de notre nouveau langage et C un ensemble de noms de canaux
  - Voici la sémantique opérationnelle de composition par des canaux



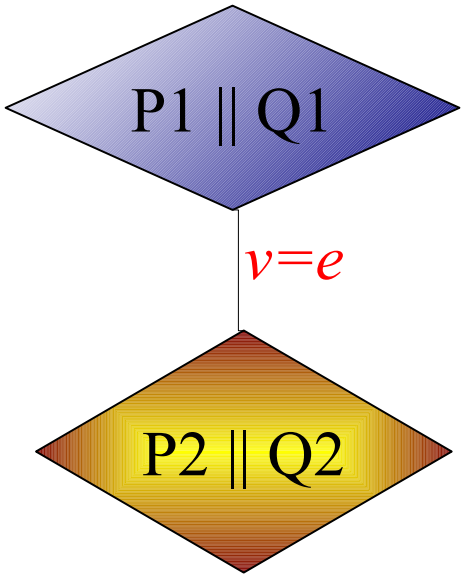


- **Par omission on conclue:**
  - Un processus qui est prêt a émettre/recevoir sur un canal est bloqué tant qu'aucun autre processus n'est capable de faire une action complémentaire sur ce même canal.
- **La synchronisation d'envoi/réception sur un canal est alors exécutée entre seulement deux processus à la fois (*un-à-un*).**
- **L'action synchrone résultante est équivalente à une affectation de la valeur envoyée vers la variable de réception.**

La communication synchrone synchronise les applications

- *ev* précède causalement *ev′ (ev --> ev′)* si [lamport 78]:
  - *ev* précède localement *ev′* (sur 1 site, dans 1 processus), ou
  - un message *m* tel que *ev = émission(m)*, *ev′ = réception(m)*, ou
  - *ev″* tel que (*ev --> ev″*) et (*ev″ --> ev′*)

P=P1;c!e;P2
Q=Q1;c?v;Q2



Émulation java d'une communication synchrone

- **Cas producteur consommateur:**
- **Canal : Mémoire tampon de taille 1**
- **Émetteur :**
  - Le producteur
- **Récepteur:**
  - Le consommateur
- **Mais il y a quelque chose qui change.**
  - Le producteur ne finit de produire que
    - S'il a fini de mettre l'objet dans le tampon
    - ET que le consommateur a consommé ce qui a été produit.
  - Le consommateur reste inchangé.

Émulation du canal

```
class Canal  {
 Object message = null;

   public synchronized void envoyer(Object v)
      throws InterruptedException {
   message = v;
   notifyAll();
     while (message != null) wait();
  }

   public synchronized Object recevoir()
      throws InterruptedException {

  while(message == null) wait();
  Object tmp = message; message = null;
   notifyAll();          //ou notify()
   return(tmp);
  }}
```

Émulation de l'émetteur

```
class Emetteur implements Runnable {
  private Canal canal;
  private SlotCanvas display;
  Emetteur(Canal c, SlotCanvas d)
    {chan=c; display=d;}

  public void run() {
    try {   int ei = 0;
      while(true) {
        display.enter(String.valueOf(ei));
        ThreadPanel.rotate(12);
        canal.envoyer(new Integer(ei));
        display.leave(String.valueOf(ei));
        ei=(ei+1)%10;
       ThreadPanel.rotate(348);
      }
    } catch (InterruptedException e){}
  }}
```

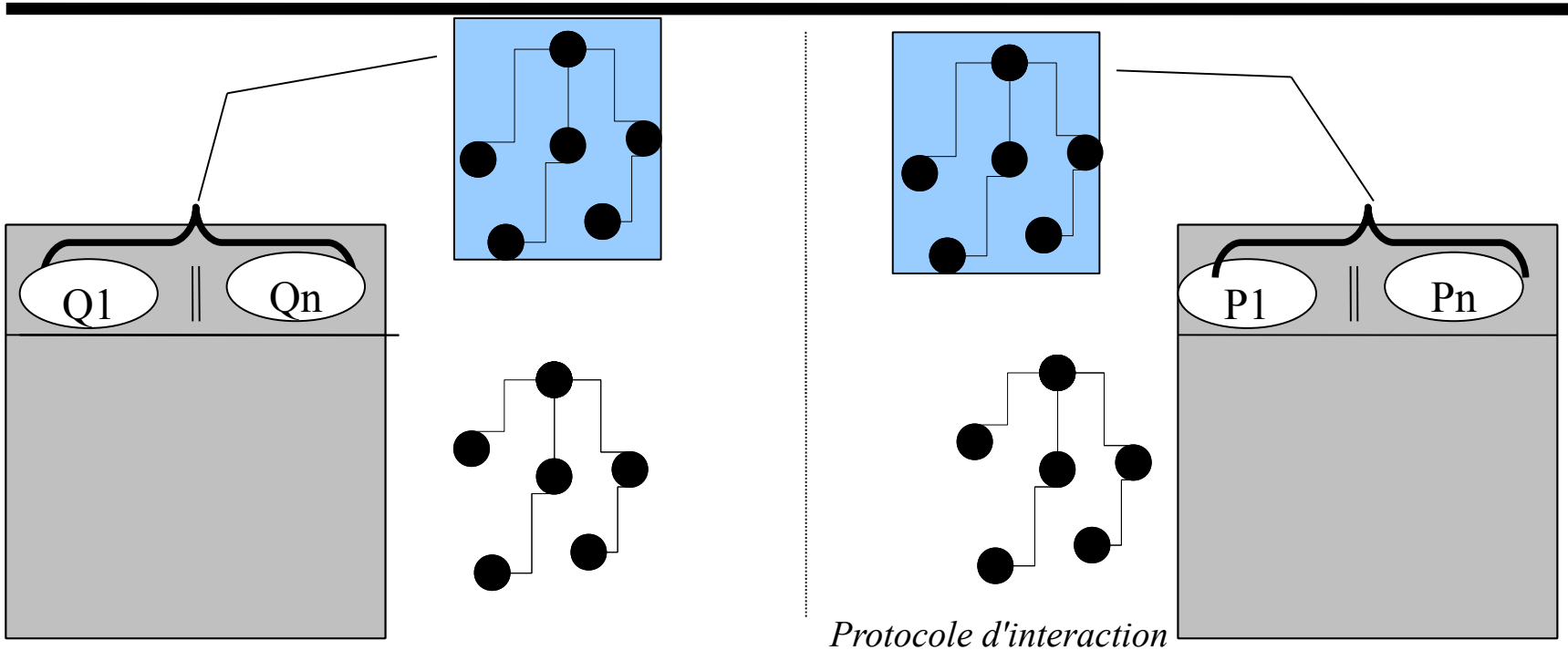
Émulation du récepteur

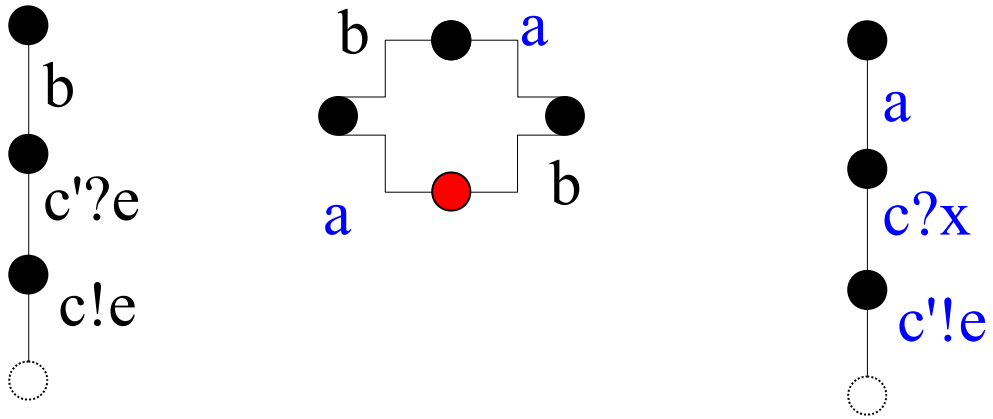
```
class Recepteur implements Runnable {
  private Canal canal;
  private SlotCanvas display;
  Receiver(Canal c, SlotCanvas d)
    {canal=c; display=d;}

  public void run() {
    try { Integer v=null;
      while(true) {
        ThreadPanel.rotate(180);
        if (v!=null) display.leave(v.toString());
        v = (Integer)canal.recevoir();
        display.enter(v.toString());
        ThreadPanel.rotate(180);
      }
    } catch (InterruptedException e){}
  }
}
```

Émulation java- communication synchrone

Caractérisation d'une application répartie
notion de deadlock





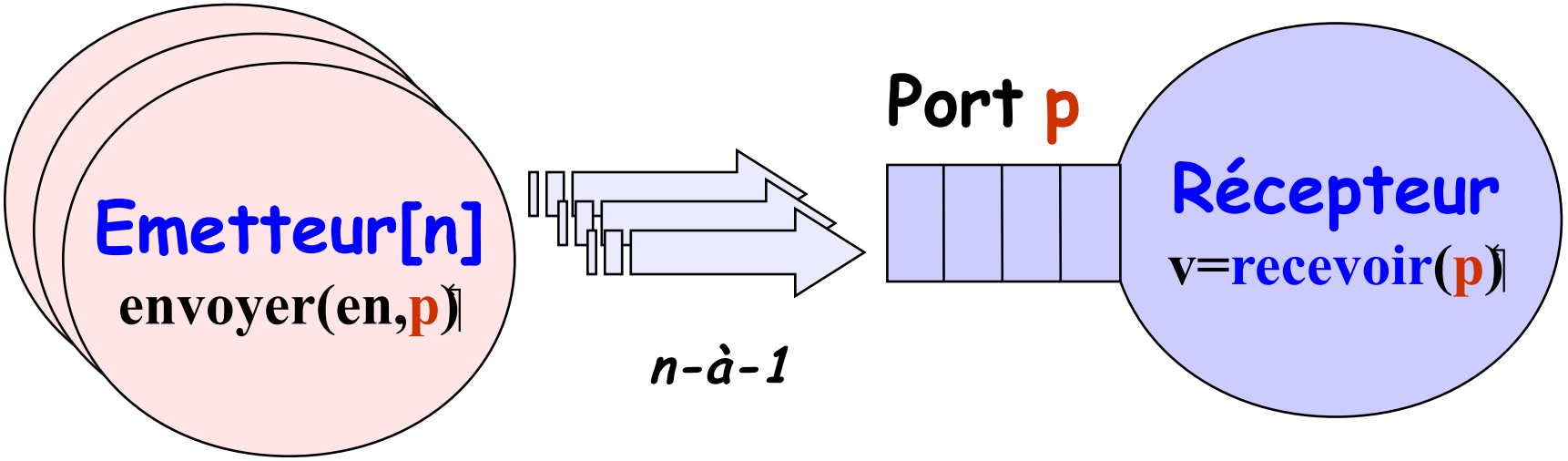
Avantage et inconvénient de la communication synchrone.

- **Communiquer est une action de synchronisation.**
- **Un moyen efficace pour le contrôle d'exécution des applications réparties**
  - => des applications simples
- **Contraignant quand la synchronisation n'est pas nécessaire mais juste l'échange de données.**
  - Beaucoup de synchronisation tue la concurrence (synonyme d'efficacité).
- **Une application mal conçue risque des blocages....**
- **Permet seulement une communication un-à-un.**
- **Solution:**
  - => Communication asynchrone

Communication asynchrone

- **C'est très contraignant : pour parler avec quelqu'un au téléphone, par exemple, il faut qu'il soit disponible pour écouter.**
- **Idée => utiliser les répondeurs**
  - Celui qui appelle peut déposer son message et continu à faire tout ce qu'il a à faire et qui ne dépend pas de la réponse.
  - Celui qui est appelé n'est pas obligé de définir son comportement en fonction de celui qui appelle. Il consulte son répondeur quand il a besoin de l'information.
- **Propriété recherchée:**
  - Action d'envoi n'est pas bloquante (peut être si le répondeur est plein)
  - L'action de réception est bloquante ssi il n'y a pas de message en attente.
  - Communication n-à-1
- **D'où l'utilisation de la notion de port**
  - Adresse d'écoute
  - Doté d'une mémoire tampon

Communication asynchrone

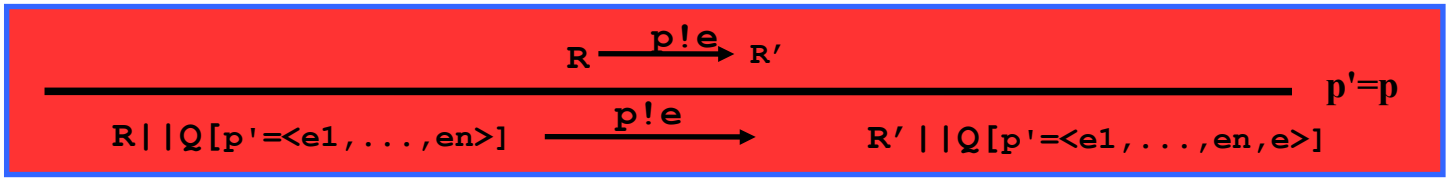


envoyer(e,p) - envoie la valeur de l'expression *e* au port *p*. le processus ne se bloque pas. Le message est stocké dans le tampon du port si celui-ci est non plein et que le récepteur n'est pas en attente.

***v* = recevoir(*p*)** - recoit une valeur du port *p* et la stocke dans une variable *v* . le processus se bloque si le tampon est vide.

Enrichissons notre langage et notre sémantique

- **Un processus peut donc avoir un ensemble de ports**
- **modélisé par une suite d'expressions p=<e1,...., en>.**
- **On garde la même syntaxe mais on change les canaux par des ports.**
- **Q [p_1=<....>,...p_n]**
- **Sémantique:**
  - *(p!e;Q)—$^{p!e}$-->Q*
  - *(p?v;Q)[p=<e1,....,en>]—$^{v=e1}$-->Q[p=<....,en>] si p*
- **Sémantique de la concurrence:**



Émulation java- communication asynchrone

- **Plusieurs producteurs - un consommateur**
- **Une zone tampon avec taille illimitée (ou limitée)**
- **Les émetteurs**
  - producteurs
- **Le port**
  - la zone tampon (taille n)
- **Le Récepteur**
  - consommateur + port.
- **Modifiez le code de canal pour qu'il devienne un port**

Émulation java- le port

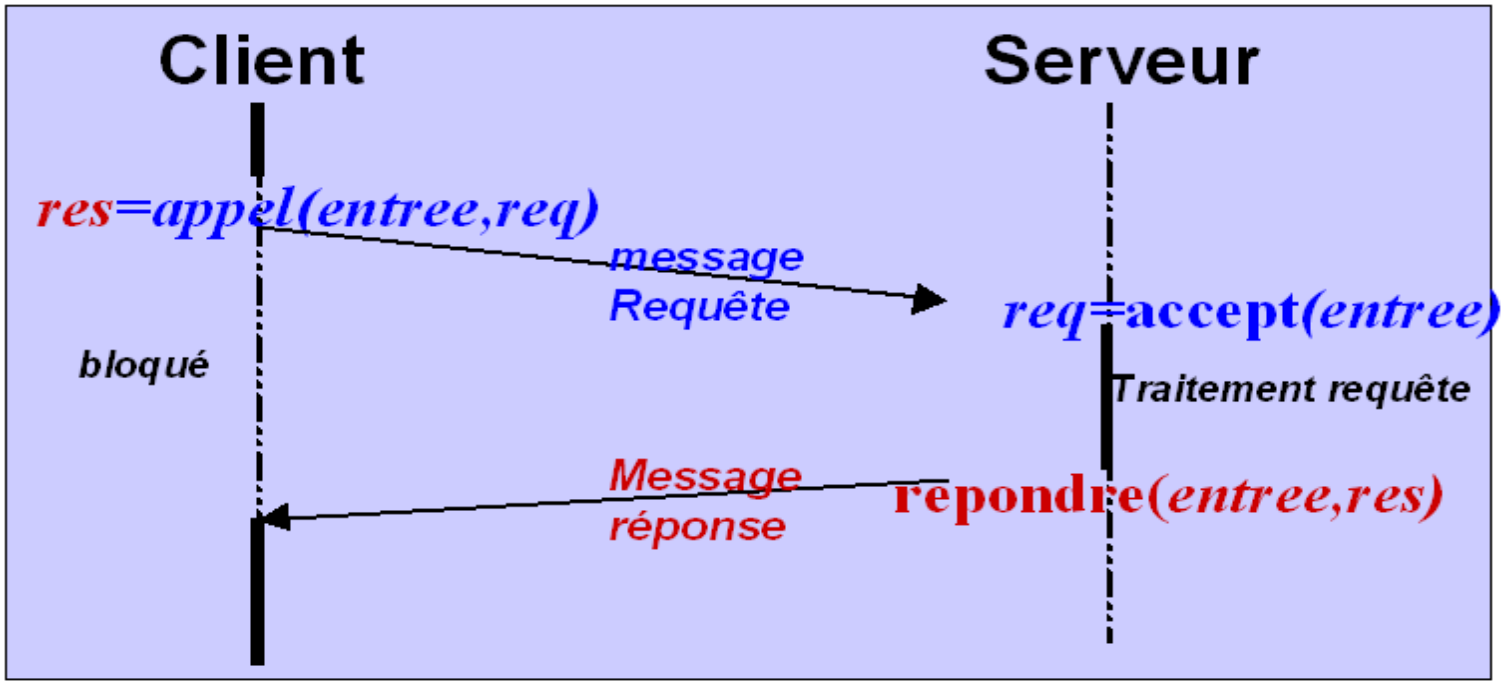
```
class Port {
 Vector queue = new Vector();
   public synchronized void envoyer(Object v){
  queue.addElement(v);
     notify();
  }

   public synchronized Object recevoir()
throws InterruptedException {
while(queue.size()==0) wait();
          Object tmp = queue.elementAt(0);
         queue.removeElementAt(0);
         return(tmp);
  }}
```

Émulation java- communication asynchrone

Communication par rendez-vous

- **La forme utilisée pour réaliser les systèmes *Requête-Réponse* pour supporter la communication *client-serveur***
- **C'est une forme qui mélange les deux premières formes**
  - Les clients envoient les demandes de *RDV* pour le traitement de requêtes.
  - Le serveur traite de manière asynchrone les requêtes. Une à la fois.
  - Les réponses sont envoyées au client sur un canal qui lui est spécifique



Communication par rendez-vous – point d'entrée

res=appel(e,req) - envoie la valeur ***req*** comme message de requête stocké dans le point d'entrée ***e***.

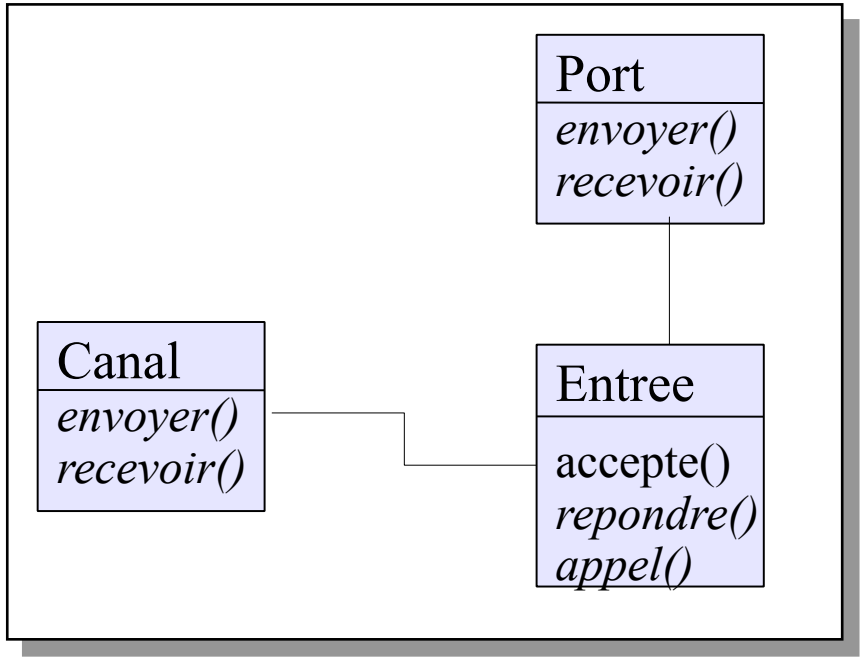
Le processus est bloqué jusqu'à l'arrivée de la réponse res.

req=accept(e) - Reçoit la valeur de la requête envoyée sur l'entrée ***e*** dans la variable. L'appel est bloquant quand l'entrée est vide.

répondre***(e,res)*** - envoit de ***res*** comme réponse à l'entrée ***e***.

Pour comprendre ce qu'est qu'un point d'entrée

Les entrées sont des ports, ils stockent les requêtes client pour un traitement asynchrone en plus d'autres fonctionnalités...



La méthode appel crée un canal et met le client en attente de la réponse sur ce canal. Le client à l'appel de appel se bloque tant qu'il n'a pas reçu ni la réponse ni un refus. La méthode appel envoit un message au port du serveur composé d'une référence au canal du client et la requête..

La méthode accepte récupère un message du tampon et récupère la requête ainsi que le canal du client. La méthode repondre envoie la réponse sur le canal du client.

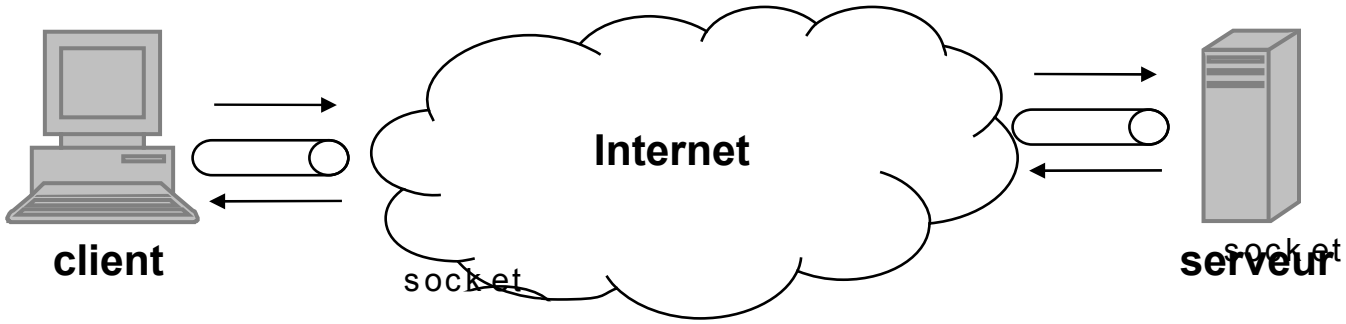
Émulation java – communication par RDV

```
public class Entree extends Port {
  private CallMsg cm;
  public Object appel(Object req) throws InterruptedException
{
    Canal clientCan = new Canal();
    envoyer(new CallMsg(req,clientCan));
    return clientChan.recevoir();
  }
  public Object accepte()throws InterruptedException {
    cm = (CallMsg) recevoir();
    return cm.req;
  }
  public void reponse(Object res) throws InterruptedException
{
    cm.repCan.envoyer(res);
  }
  private class CallMsg {
    Object  req; Canal repCan;
    CallMsg(Object m, Canal c)
      {req=m; repCan=c;}
  }
}
```

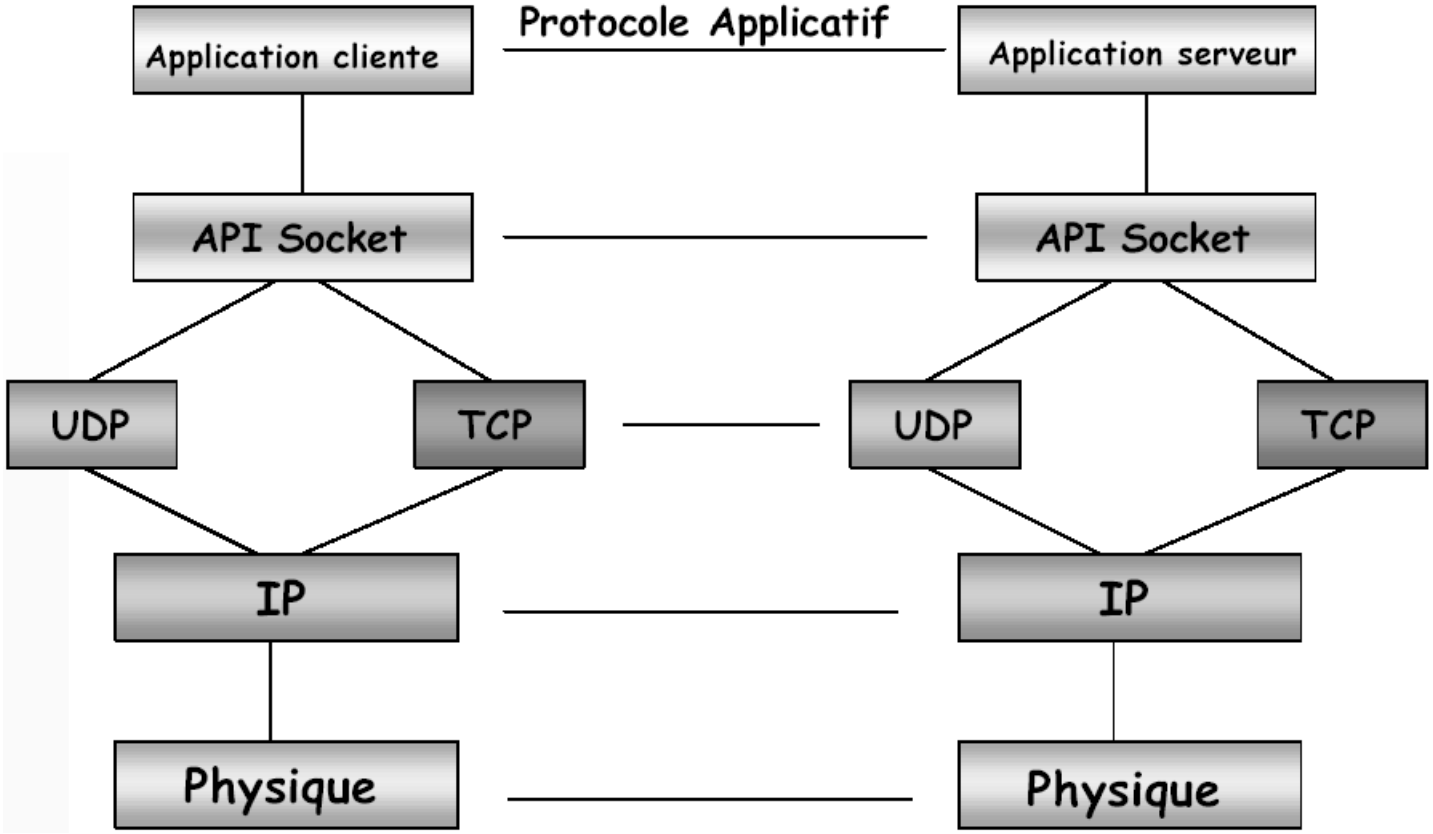
Émulation java – communication par RDV

Le modèle des sockets [N. Melab (LIFL)]

- **Interface (point de communication) client/serveur utilisée à l'origine dans le monde UNIX et TCP/IP**
  - étendue aux PCs (Winsock) et mainframes
  - primitives pour le support de communications reposant sur les protocoles (TCP/IP, UDP/IP)
  - les applications client/serveur ne voient les couches de communication qu'à travers l'API socket (abstraction)



Sockets et OSI [N. Melab (LIFL)]

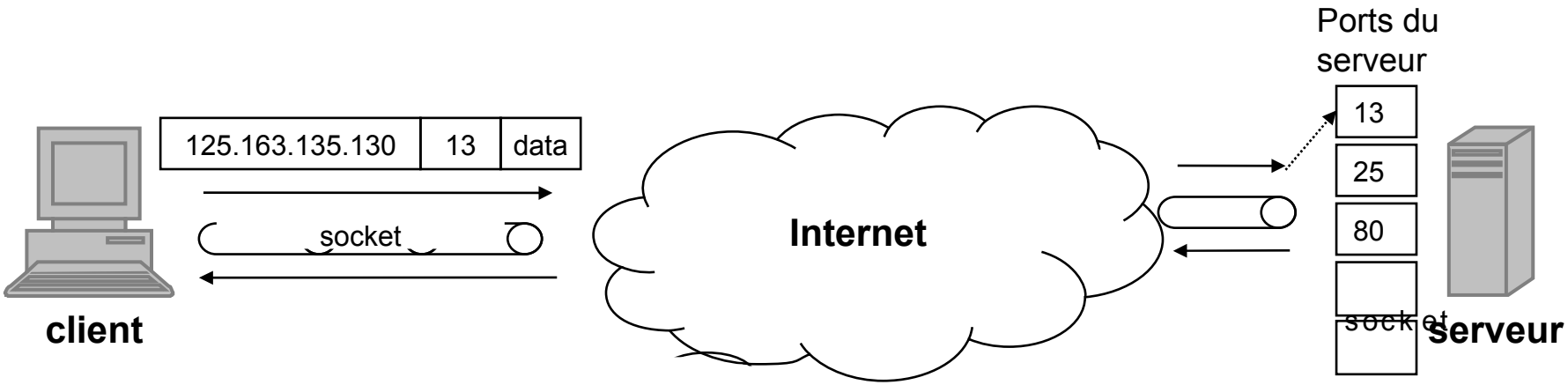


Protocoles TCP et UDP [N. Melab (LIFL)]

- **TCP**
  - Garantie d'arrivée dans l'ordre de paquets de données
  - Fiabilité de transmission
  - Lenteur des transmissions (http par exemple)
  - Vu comme un «service téléphonique»
- **UDP**
  - Arrivée dans le bon ordre non garantie
  - Non fiabilité des transmissions
  - Rapidité des transmissions
  - Applications audio/vidéo
  - Vu comme un «service postal»

Connexion réseau

- Adresse Internet de la machine
- Numéro du port



Notion de port [N. Melab (LIFL)]

- **Pourquoi les ports ?**
  - Sur une même machine, plusieurs services sont accessibles simultanément (web, email, etc.)
  - Points d'accès : ports logiques (65535)
  - Rien à voir avec les ports physiques (série et parallèle)
- **Désignation des ports**
  - Port : numéro allant de 1 à 65535
  - Les ports de 1 à 1023 sont réservés aux services courants finger, ftp, http (80), SMTP (25), etc.
  - Fichier d'assignation de ports : /etc/services

Adresses Internet [N. Melab (LIFL)]

- **Connexion réseau**
  - Adresse Internet de la machine
  - Numéro : 193.49.192.193
- **Désignation par des noms symboliques**
  - Association de noms symboliques aux adresses numériques
  - Domain Name Server (ou DNS)
  - Exemple : lil.univ-littoral.fr : 193.49.192.193

Client-serveur en mode connecté [M. Riveill (INPG)]

- **Le client**
  - ouvre une connexion avec le serveur avant de pouvoir lui adresser des appels, puis ferme la connexion à la fin de la suite d'opération
    - délimitation temporelle des échanges
    - maintien de l'état de connexion pour la gestion des paramètres de qualité de service
      - traitement des pannes, propriété d'ordre
  - orienté vers
    - traitement ordonné d'une suite d'appels
      - ordre local (requêtes d'un client traitées dans leur ordre d'émission), global ou causal
    - la gestion de données persistantes ou de protocole avec état

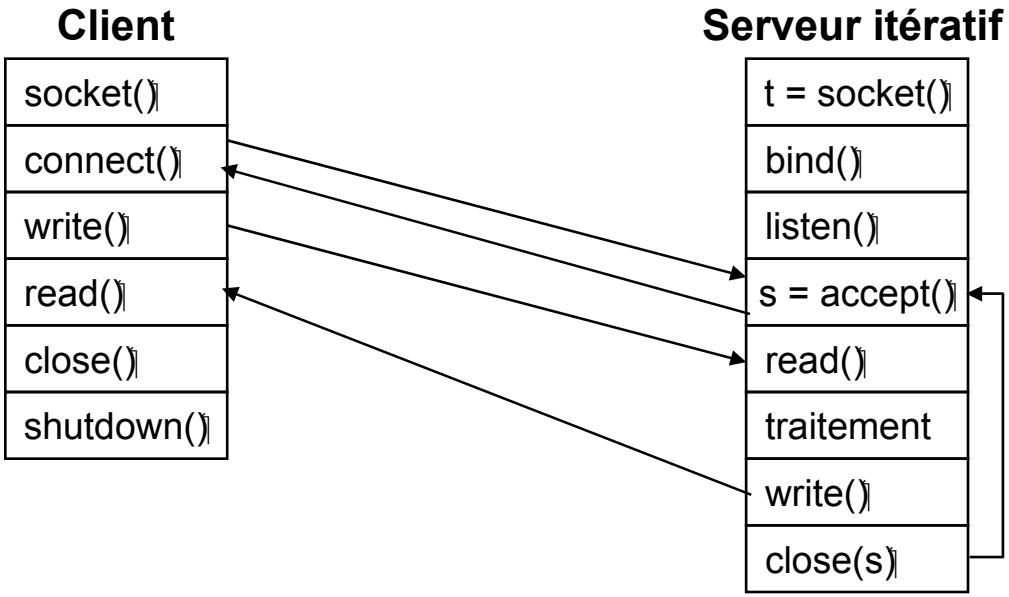
Mode connecté : caractéristiques [M. Riveill (INPG)]

- **Caractéristiques**
  - établissement préalable d'une connexion (circuit virtuel) : le client demande au serveur s'il accepte la connexion
  - fiabilité assurée par le protocole de transport utilisé : TCP
  - mode d'échange par flots d'octets : le récepteur n'a pas connaissance du découpage des données effectué par l'émetteur
  - possibilité d'émettre et de recevoir des caractères urgents (OOB : Out Of Band)
  - après initialisation, le serveur est "passif", il est activé lors de l'arrivée d'une demande de connexion d'un client
  - un serveur peut répondre aux demandes de services de plusieurs clients : les requêtes arrivées et non traitées sont stockées dans une file d'attente

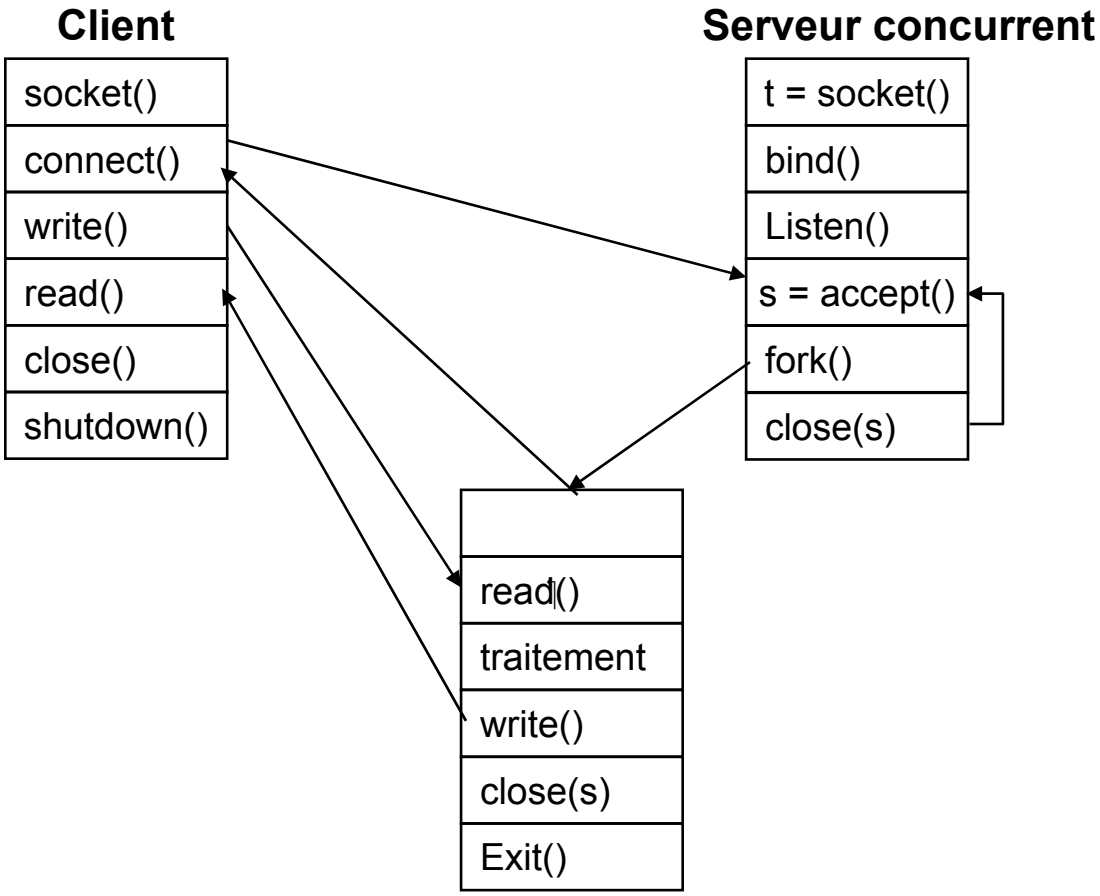
Caractéristiques du mode connecté [M. Riveill (INPG)]

- **Contrainte**
  - le client doit avoir accès à l'adresse du serveur (adresse IP et numéro de port)
- **Modes de gestion des requêtes**
  - itératif : le processus serveur traite les requêtes les unes après les autres
  - concurrent : par création de processus fils pour les échanges de chaque requête

Enchaînement des opérations (1) [M. Riveill (INPG)]



Enchaînement des opérations (2) [M. Riveill (INPG)]



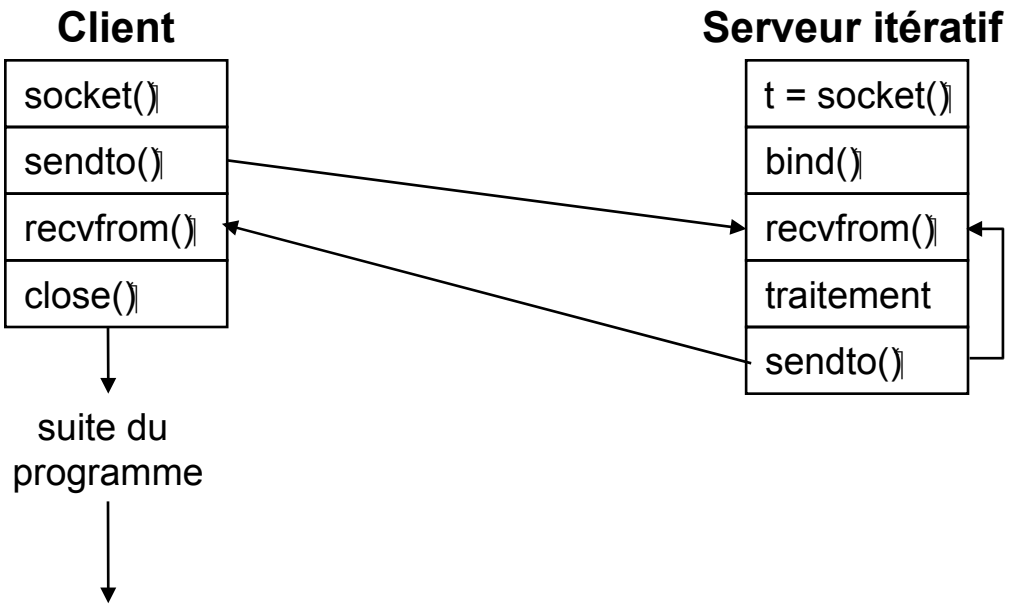
Client-serveur en mode non connecté [M. Riveill (INPG)]

- **le client peut envoyer des appels au serveur à n'importe quel moment**
  - mode assez léger orienté
    - traitement non ordonné des appels
    - absence de mémoire entre appels successifs (serveur sans données rémanentes et sans état)
  - exemple :
    - calcul de fonction numérique
    - DNS
    - NFS

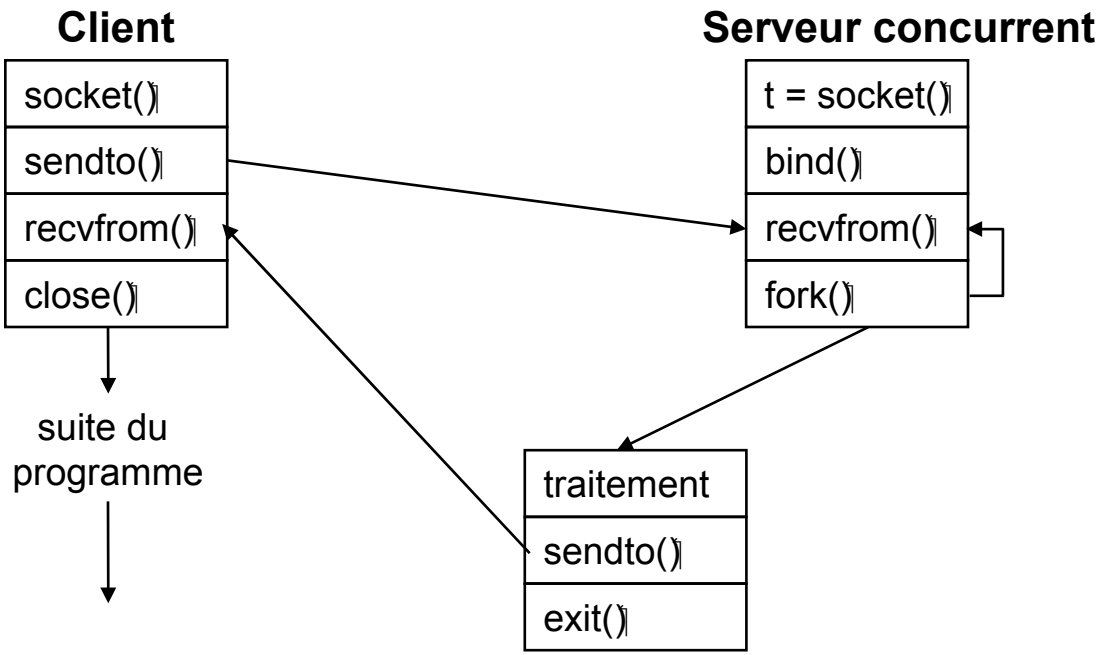
Mode non connecté : caractéristiques [M. Riveill (INPG)]

- **Caractéristiques**
  - pas d'établissement préalable d'une connexion
  - adapté aux applications pour lesquelles les réponses aux requêtes des clients sont courtes (un message)
  - protocole de transport utilisé : UDP
  - mode d'échange par messages : le récepteur reçoit les données suivant le même découpage que celui effectué par l'émetteur
- **Contraintes**
  - le client doit avoir accès à l'adresse du serveur (adresse IP et numéro de port)
  - pour répondre à chaque client, le serveur doit en récupérer l'adresse : il faut pour cela utiliser les primitives **`sendto`** et **`recvfrom`**
- **Mode de gestion des requêtes**
  - itératif : le processus serveur traite les requêtes les unes après les autres
  - concurrent : par création de processus fils pour les échanges de chaque requête

Enchaînement des opérations (1) [M. Riveill (INPG)]



Enchaînement des opérations (2) [M. Riveill (INPG)]



la communication Sous Java

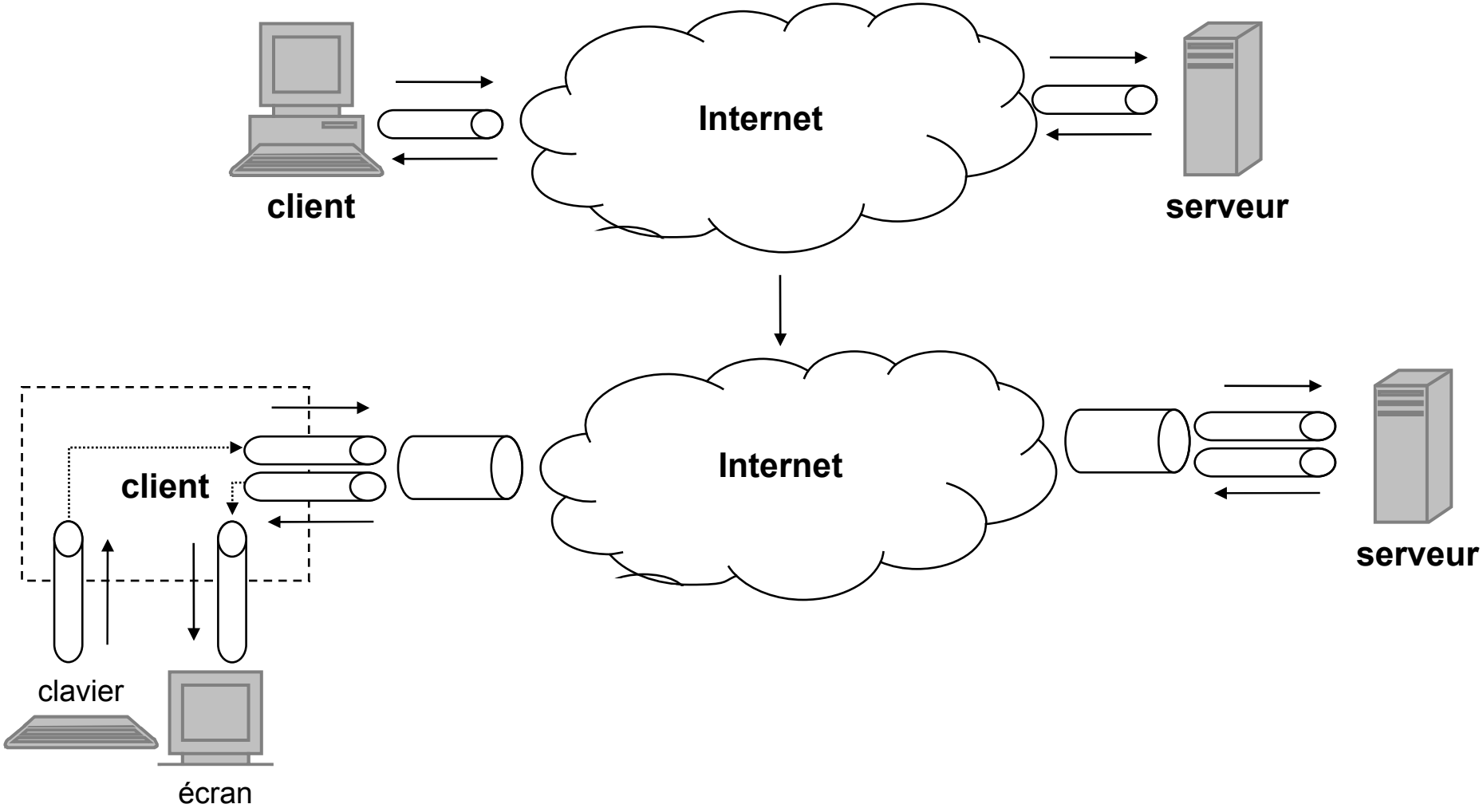
- **La classe Socket (et ServerSocket)**
  - Mode connecté
  - TCP.
  - Un mélange entre RDV pour obtenir un canal asynchrone (cf plus loin)
- **La classe DatagramSocket (et DatagramPacket)**
  - Mode non connecté
  - UDP.
  - Asynchrone

Mode connecté

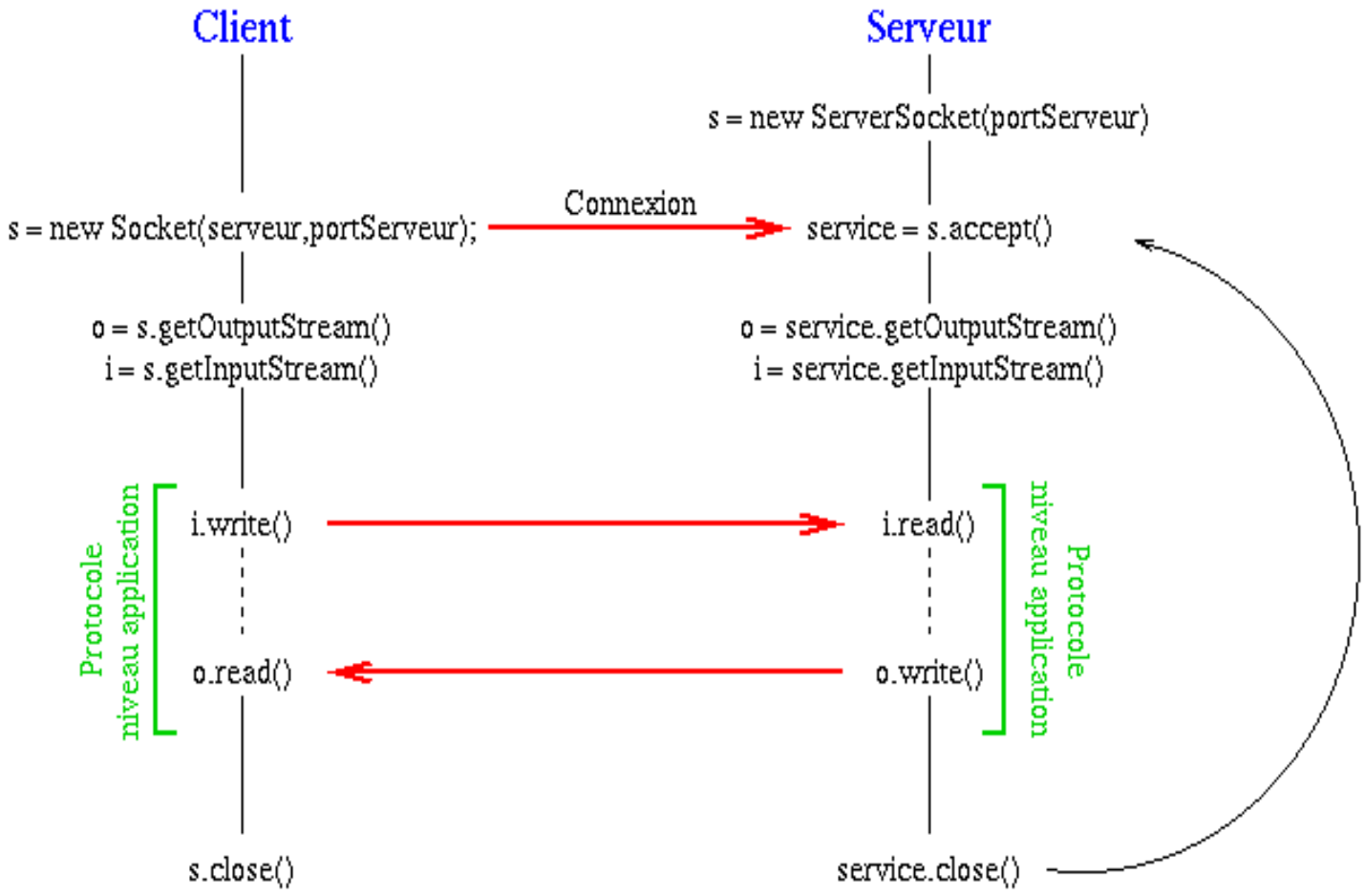
les sockets TCP sous Java

- **Deux classes interviennent:**
- **java.net.Socket**
  - Coté client (mais aussi coté serveur)
  - Elle permet une communication 1-1
  - C'est un couple de canaux (asynchrone)
- **java.net.ServerSocket**
  - Utilisé uniquement coté serveur
  - C'est un point d'entrée
  - Req : demande d'établissement d'une connexion
  - Rep : établissement d' une Socket entre le serveur et le client

Les sockets Java



Les sockets sous Java



Java.net.Socket (TCP)

- **`Socket(String host, int port)`**
  - crée une socket et la connecte à un port de l'ordinateur distant
- **`void close()`**
  - ferme la socket
- **`InputStream getInputStream()`**
  - récupère le flux de données pour lire sur la socket
- **`OutputStream getOutputStream()`**
  - récupère le flux de données pour écrire sur la socket
- **`void setSoTimeout(int timeout)`**
  - définit la valeur (en ms) de timeout en lecture sur cette socket
  - si la valeur de timeout est atteinte, une **`InterruptedIoException`** est déclenchée

Un premier client

- **Interrogation du service « date » d'un serveur**

```
$ telnet time-A.timefreq.bldrdoc.gov 13
$ 50692 05-01-10 10:27:15 50 0 0 50.0 UTC(NIST) *
```

- **Implantation Java**

```
import java.io.*;
import java.net.*;

public class SocketTest {
  public static void main(String[] args) {
    try {
      Socket s = new Socket("time-A.timefreq.bldrdoc.gov", 13);
      BufferedReader in = new BufferedReader
        (new InputStreamReader(s.getInputStream()));
      boolean more = true;
      while (more) {
        String line = in.readLine();
        if (line == null)
          more = false;
        else
          System.out.println(line);
      }
    }
    catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Attention la création d'un socket est une action bloquante

- **Problème**
  - si le serveur ne répond pas, le client est bloqué
- **Solution**
  - utilisation d'un timeout au bout duquel la socket sera fermée
  - quand la valeur de timeout est dépassée, toutes les opérations de lecture lancent une **`InterruptedIOException`**
- **Exemple**

```
Socket s = new Socket(...);
s. setSoTimeout(10000);
...
try {
  String line;
  while ((line = in.readLine()) != null) {
    traitement de la ligne
  }
}
catch (InterruptedIoException exception) {
  gestion du timeout
}
```

Les adresses Internet

- **La classe `java.net.InetAddress`**
  - l'objet **`InetAddress`** encapsule la séquence de 4 octets
  - méthodes qui offrent les services d'un DNS
- **Exemple**

```
import java.net.*;

public class InetAddressTest {
  public static void main(String[] args) {
    try {
      if (args.length > 0) {
        String host = args[0];
        InetAddress[] addresses = InetAddress.getAllByName(host);
        for (int i = 0; i < addresses.length; i++)
          System.out.println(addresses[i]);
      }
      else {
        InetAddress localHostAddress = InetAddress.getLocalHost();
        System.out.println(localHostAddress);
      }
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Java.net.InetAddress

- **`static InetAddress getByName(String host)`**
  - récupère l'addresse IP associée à un nom d'hôte
- **`static InetAddress[] getAllByName(String host)`**
  - récupère toutes les addresses IP associées à un nom d'hôte
- **`static InetAddress getLocalHost()`**
  - récupère l'addresse IP de l'ordinateur local
- **`byte[] getAddress()`**
  - renvoie un tableau d'octets contenant une addresse numérique
- **`String getHostAddress`**
  - renvoie une chaîne de caractères sous la forme de valeurs décimales séparées par des points
- **`String getHostName()`**
  - renvoie le nom de l'ordinateur

Java.net.ServerSocket

- **Elle sert à mettre en place un serveur**
- **Le relier à un port (logique de la machine)**
- **Permet de traiter les requêtes de demande de connexion:**
  - Les requêtes sont stockées dans le tampon du port et traitées une à une.
  - Le résultat de chaque traitement de requête est un objet Socket reliant le serveur et le client

Java.net.ServerSocket

- **`ServerSocket(int port) throws IOException`**
  - crée une socket serveur qui examine un port
- **`Socket accept() throws IOException`**
  - attend une connexion
  - bloque le thread courant jusqu'à une demande de connexion
  - renvoie un objet **`Socket`** pour communiquer avec le client
- **`void close() throws IOException`**
  - ferme la socket du serveur

Un premier serveur

- **Affichage en écho des messages reçus du client**
- **Implantation Java**

```
Bonjour, tapez OK pour sortir
Bonjour, comment allez-vous
Echo : Bonjour, comment allez-vous
Très bien et vous ?
Echo : Très bien et vous ?
OK
Echo : OK
```

```
import java.io.*;
import java.net.*;

public class EchoServer {
  public static void main(String[] args) {
    try {
      ServerSocket s = new ServerSocket(8189);
      Socket incoming = s.accept( );
      BufferedReader in = new BufferedReader
           (new InputStreamReader(incoming.getInputStream()));
      PrintWriter out = new PrintWriter
           (incoming.getOutputStream(), true /* autoFlush */);
      out.println( "Bonjour, tapez OK pour sortir" );
      boolean done = false;
      while (!done) {
        String line = in.readLine();
        if (line == null) done = true;
        else {
          out.println("Echo: " + line);
          if (line.trim().equals("OK"))
            done = true;
        }
      }
      incoming.close();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Un serveur multi-threadé

- **Problème**
  - impossible de servir plusieurs clients en même temps
- **Solution**
  - créer un nouveau thread à chaque nouvelle connexion
  - chaque thread est chargé de la gestion entre le serveur et un client particulier
  - cette gestion s'effectue dans la méthode **`run`** du thread
- **Implantation**

```
...
ServerSocket s = new ServerSocket(8189);

for (;;) {
  Socket incoming = s.accept( );
  Thread t = new ThreadedEchoHandler(incoming);
  t.start();
}
```

Mode non connecté

Les socket UDP sous Java

- **Deux classes interviennent :**
- **DatagramPacket**
  - Cette classe permet de créer des objets qui contiendront les données envoyées ou reçues ainsi que l'adresse de destination ou de provenance du datagramme.
  - Deux constructeurs disponibles
    (un pour le coté client et un pour le coté serveur).
- **DatagramSocket**
  - Cette classe permet de créer des sockets UDP qui permettent d'envoyer et de recevoir des datagrammes UDP.
  - La création d'une DatagramSocket est essentielle
    aussi bien coté client que coté serveur.

DatagramPacket

- **Constructeur pour recevoir les données (coté serveur) :**

```
DatagramPacket(byte buffer[], int taille)
```

  - **- buffer pour mettre les données**
  - **- taille maximale à lire (le reste est perdu)**

- **Constructeur pour envoyer les données**

```
DatagramPacket(byte buffer[], int taille,
        InetAddress adresse, int port)
```

  - **- buffer pour mettre les données**
  - **- taille maximale à envoyer (le reste est pas envoyé)**
  - **- plus l'adresse et le port du récepteur**

DatagramSocket (1)

- **Constructeur pour client**

```
public DatagramSocket ()
```

- on ne spécifie pas le port d'attachement

- notez bien que l'adresse de la destination ne figure pas dans la socket mais dans le DatagramPacket

- **Constructeur pour le serveur**

```
public DatagramSocket (int port)
```

- on spécifie le port

- il existe aussi un autre constructeur où on spécifie aussi l'adresse

DatagramSocket (2)

- **Envoi de message (asynchrone = non bloquant)**

```
public void send(DatagramPacket data) throws...
```

- envoi des données de data

au serveur dont l'adresse est spécifiée dans data

- **Réception (synchrone = bloquante)**

```
public synchronized void receive(DatagramPacket
                    data) throws...
```

- reçoit les données dans data

- il est possible de mettre une garde sur la réception (méthode SetTimeout(int x))

- après la réception data contient : les données, la taille réelle et l'adresse plus le port de l'émetteur.

Exemple Echo : le Serveur

```
import java.io.*;
import java.net.*;

class ServeurEcho
{
 final static int port = 8532;
 final static int taille = 1024;
 final static byte buffer[] = new byte[taille];

 public static void main(String argv[]) throws Exception
 {
 DatagramSocket socket = new DatagramSocket(port);
 while(true)
 {
 DatagramPacket data = new DatagramPacket(buffer,buffer.length);
 socket.receive(data);
 System.out.println(data.getAddress());
 socket.send(data);
 }
 }
}
```

Exemple Echo : le Client

```
import java.io.*;
import java.net.*;
public class ClientEcho
{
 final static int taille = 1024;
 final static byte buffer[] = new byte[taille];
 public static void main(String argv[]) throws Exception
 {
 InetAddress serveur = InetAddress.getByName(argv[0]);
 int length = argv[1].length();
 byte buffer[] = argv[1].getBytes();
 DatagramPacket dataSent = new DatagramPacket(buffer,length,serveur,ServeurEcho.port);
 DatagramSocket socket = new DatagramSocket();

 socket.send(dataSent);

 DatagramPacket dataRecieved = new DatagramPacket(new byte[length],length);
 socket.receive(dataRecieved);
 System.out.println("Data recieved : " + new String(dataRecieved.getData()));
 System.out.println("From : " + dataRecieved.getAddress() + ":" + dataRecieved.getPort());
 }
}
```

Rappels sur les flux Java (`java.io`)

- **Les entrées/sorties**
  - Communication entre le programme et le monde extérieur

- **Philosophie en Java**
  - les opérations E/S peuvent avoir des origines très diverses
    - Entrées : clavier, fichier, réseau, autre programme
    - Sorties : écran, fichier, réseau, autre programme
  - les flux ont une interface standard
    - les opérations effectuées sont indépendantes de la nature du périphérique concerné

Différents types de flux

- **Flux à accès séquentiel**

  les données sont traitées les unes après les autres dans un ordre qui ne peut pas être changé

  - majorité des flux Java
  - flux unidirectionnels (lecture OU écriture)
  - diverses catégories en fonction de la nature des données, du sens de transfert, du type de source ou de destination

- **Flux à accès indexé**
  - permet d'accéder à un fichier en choisissant directement la position (méthode **`seek`**) à laquelle lire ou écrire
  - flux bidirectionnel = un seul flux permet à la fois la lecture et l'écriture
  - une seule classe = **`RandomAccessFile`**

Construction des noms de flux Java (1)

PrŽfixe du nom de flux	Suffixe du nom de flux
type de la source ou de la destination (suivant le sens du flux)	nature du flux
nature du traitement	nature du flux

	Flux dÕentrŽe (lecture)	Flux de sortie (Žcriture)
Flux de caractüres	Reader	Writer
Flux dÕoctets	InputStream	OutputStream

Traitement	PrŽfixe du nom du flux
tampon	Buffered
concatŽnation de flux dÕentrŽe	Sequence
conversion de donnŽes	Data
numŽrotation des lignes de texte	LineNumber
lecture avec retour arriüre	PushBack
impression	Print
sŽrialisation et dŽsŽrialisation dÕobjets	Object
conversion dÕoctets en caractüres (et rŽciproquement)	InputStream ou OutputStream

Source ou destination	PrŽfixe du nom du flux
tableau de caractüres	CharArray
flux dÕoctets	InputStream ou OutputStream
chaŒne de caractüres	String
programme	Pipe
fichier	File
tableau dÕoctets	ByteArray
objet	Object

Construction des noms de flux Java (2)

- Liste des flux séquentiels de **`java.io`** (les classes en italique sont abstraites)

Reader	*Writer*	*InputStream*	*OutputStream*
BufferedReader	BufferedWriter	BufferedInputStream	BufferedOutputStream
CharArrayReader	CharArrayWriter	ByteArrayInputStream	ByteArrayOutputStream
FileReader	FileWriter	DataInputStream	DataOutputStream
FilterReader	*FilterWriter*	FileInputStream	FileOutputStream
InputStreamReader	InputStreamWriter	*FilterInputStream*	*FilterOutputStream*
LineNumberReader		ObjectInputStream	ObjectOutputStream
PipedReader	PipedWriter	PipedInputStream	PipedOutputStream
	PrintWriter		PrintStream
PushBackReader		PushbackInputStream	
StringReader	StringWriter	SequenceInputStream	

Opérations courantes sur des flux de caractères

Signature de la méthode	Classes concernées	Donnée traitée	Type et valeur retournés	Exceptions levées
`read()`	`Reader`	caractère	int : • le caractère lu • -1, si fin de flux	IOException si erreur de lecture
`write(int)`	`Writer`	caractère extrait des 16 bits de poids faible de l'argument		IOException si erreur d'écriture
`read(char[])`	`Reader`	tableau de caractères	int : • le nombre de caractères lus • -1, si fin de flux	IOException si erreur de lecture
`write(char[])`	`Writer`	tableau de caractères donné en argument		IOException si erreur d'écriture
`readLine()`	`BufferedReader`	chaîne de caractères	String : • la chaîne lue sans les caractères de terminaison • null, si fin de flux	IOException si erreur de lecture
`write(String)`	`Writer`	chaîne de caractères donnée en argument		IOException si erreur d'écriture
`print(arg)` ou `println(arg)`	`PrintWriter`	arg : donnée de type simple, chaîne de caractères ou objet		aucune, CheckError() renvoie true en cas d'erreur

Opérations courantes sur des flux d'octets

Signature de la méthode	Classes concernées	Donnée traitée	Type et valeur retournés	Exceptions levées
read()	**classes dérivées de InputStream**	octet extrait des 8 bits de poids faibles de la valeur retournée	int : • l'octet lu • -1, si fin de flux	IOException si erreur de lecture
write(int)	**classes derives de OutputStream**	octet extrait des 8 bits de poids faibles de l'argument		IOException si erreur d'écriture
read(byte[])	**classes dérivées de InputStream**	tableau d'octets	int : • nombre d'octets lus • -1, si fin de flux	IOException si erreur de lecture
write(byte[])	**classes dérivées de OutputStream**	tableau d'octets		IOException si erreur d'écriture
readDouble()	**DataInputStream**	double	double : valeur lue	EOFException si fin de flux IOException si erreur de lecture
writeDouble(double)	**DataOutputStream**	double		IOException si erreur d'écriture
read*TypeSimple*()	**DataInputStream**	*TypeSimple* (pour Boolean, Char, Double, Float, Int, Long, Short)	*TypeSimple* : valeur lue	EOFException si fin de flux IOException si erreur de lecture
***writeTypeSimple*()**	**DataOutputStream**	*TypeSimple* (pour Boolean, Char, Double, Float, Int, Long, Short)		IOException si erreur d'écriture

Lecture/écriture de caractères dans un fichier

```
// création d'un flux de lecture de caractères dans un fichier
BufferedReader fichEntree = new BufferedReader(new FileReader("fichEntree.txt"));

// utilisation d'un flux de lecture de caractères dans un fichier
void loadFile(BufferedReader entree) throws Exception {
  int valeurEntiere = Integer.valueOf(entree.readLine()).intValue();
  double valeurDouble = Double.valueOf(entree.readLine()).doubleValue() ;
  boolean valeurBooleenne = Boolean.valueOf(entree.readLine()).booleanValue() ;
  byte valeurByte = Byte.valueOf(entree.readLine()).byteValue() ;
  entree.close() ;
}

// création d'un flux d'écriture de caractères dans un fichier
PrintWriter fichSortie = new PrintWriter(new FileWriter("fichSortie.txt"));

void toFile(PrintWriter sortie) throws Exception {
  sortie.println(valeurEntiere);
  sortie.println(valeurDouble);
  sortie.println(valeurBooleenne);
  sortie.println(valeurByte);
  sortie.close();
}
```

Lecture/écriture d'octets dans un fichier

```
// création d'un flux de lecture d'octets dans un fichier
DataInputStream fichEntree = new DataInputStream(new FileInputStream("fichEntree.bin"));

void loadFile(DataInputStream entree) throws Exception {
  int valeurEntiere = entree.readInt();
  double valeurDouble = entree.readDouble();
  boolean valeurBooleenne = entree.readBoolean();
  byte valeurByte = entree.readByte();
  entree.close() ;
}

// création d'un flux d'écriture d'octets dans un fichier
DataOutputStream fichSortie = new DataOutputStream(new
    FileOutputStream("fichSortie.bin"));

void toFile(DataOutputStream sortie) throws Exception {
  sortie.writeInt(valeurEntiere);
  sortie.writeDouble(valeurDouble);
  sortie.writeBoolean(valeurBooleenne);
  sortie.writeByte(valeurByte);
  sortie.close();
}
```

Retour aux sockets

Quelques références

- **Ce cours a été réalisé en se basant sur:**
- **Cours Guillaume Hutzler pour la partie sur les Socket**
  http://www.lami.univ-evry.fr/%7Ehutzler/Cours/CPAR/Sockets.pdf

- **Une interprétation libre du chapitre 10 du livre :**

  Concurrency: State Models & Java Programs de *Jeff Magee & Jeff Kramer*

  les 10 premiers chapitres sont dispo en ligne ainsi que les slides qui vont avec (En Anglais ;-).

  – *http://www-dse.doc.ic.ac.uk/concurrency/*