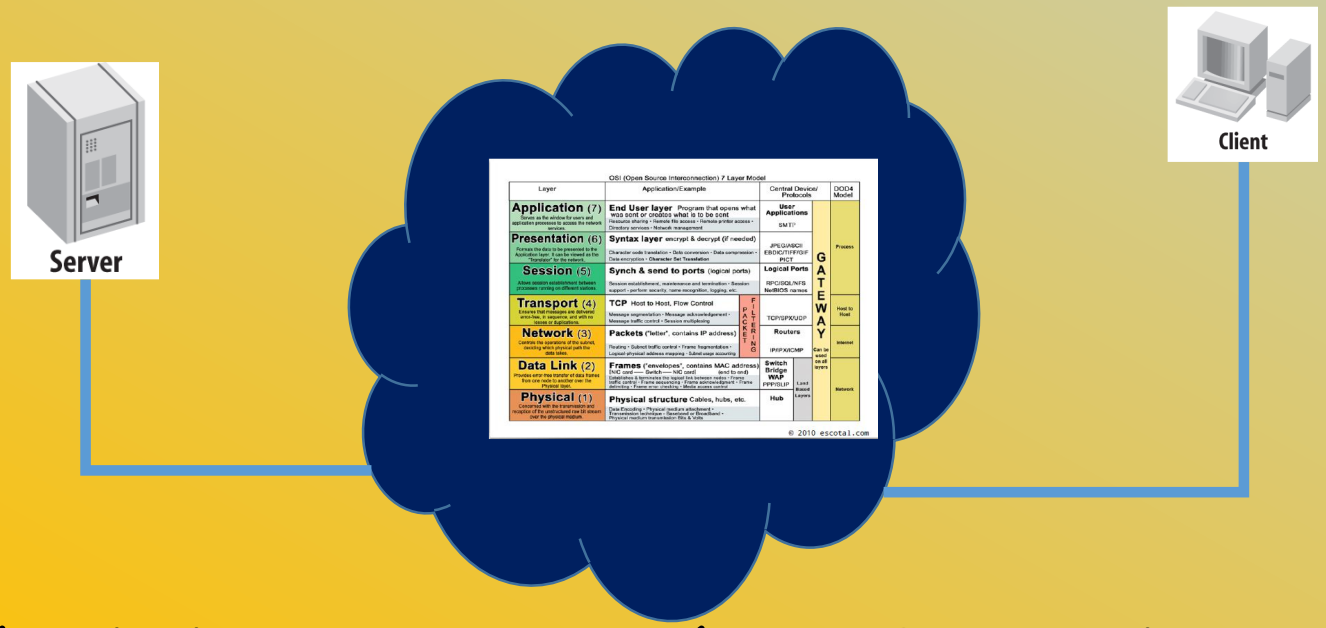
Introduction

- Dans ce cours on s'intéresse à l'écriture d'applications qui interagissent (communiquent) à travers un réseau.
- Nous aurons besoin de quelques notions de base sur les réseaux informatiques.
- Mais, ce n'est pas les détails de fonctionnement des réseaux qui nous intéressent.

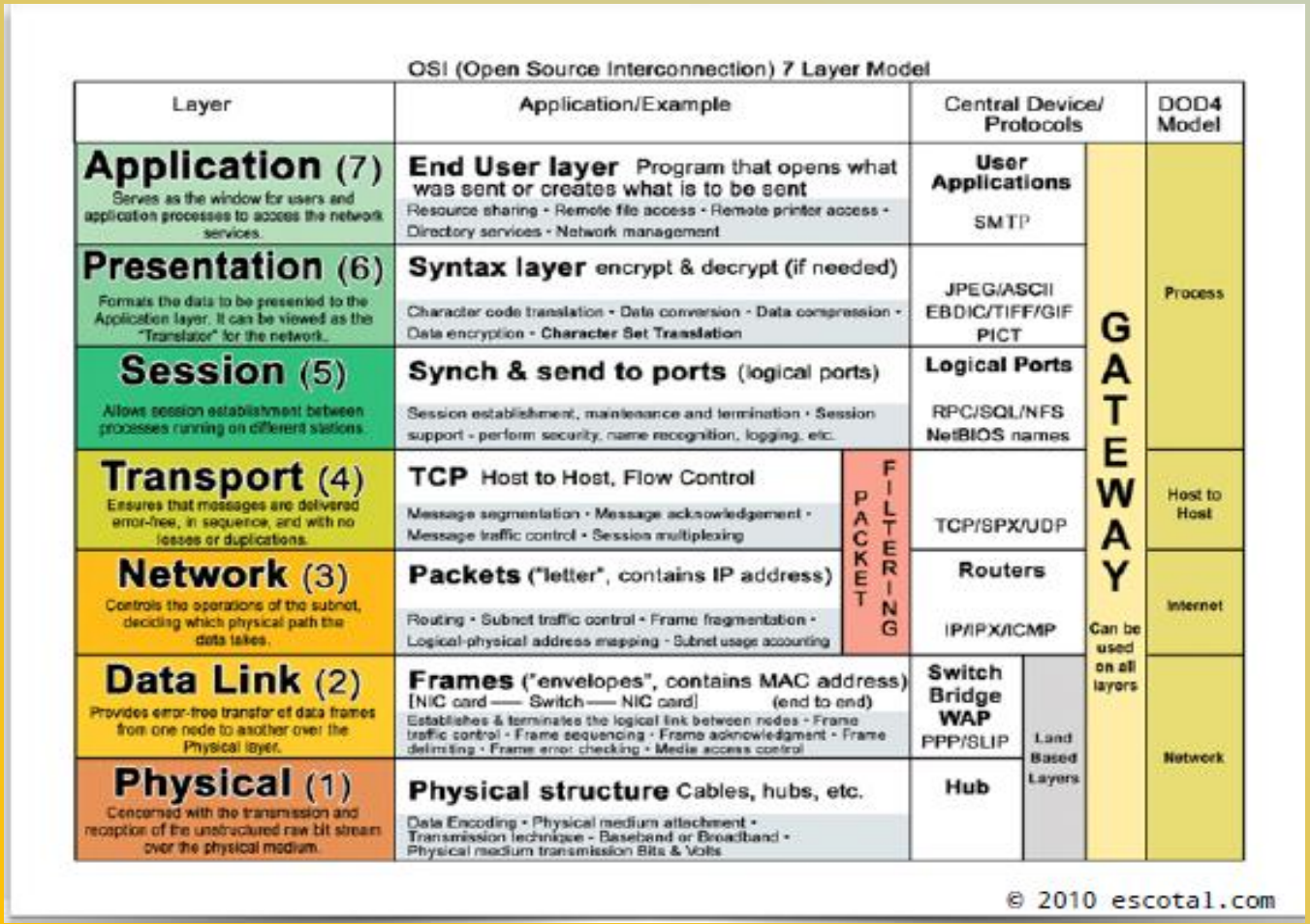
Une application voit le réseau comme un service, un moyen de communication, ou un espace d'une certaine nature.

- L'application ne s'intéresse pas aux détails de l'architecture du réseaux ou aux divers protocoles qui assurent son fonctionnement.



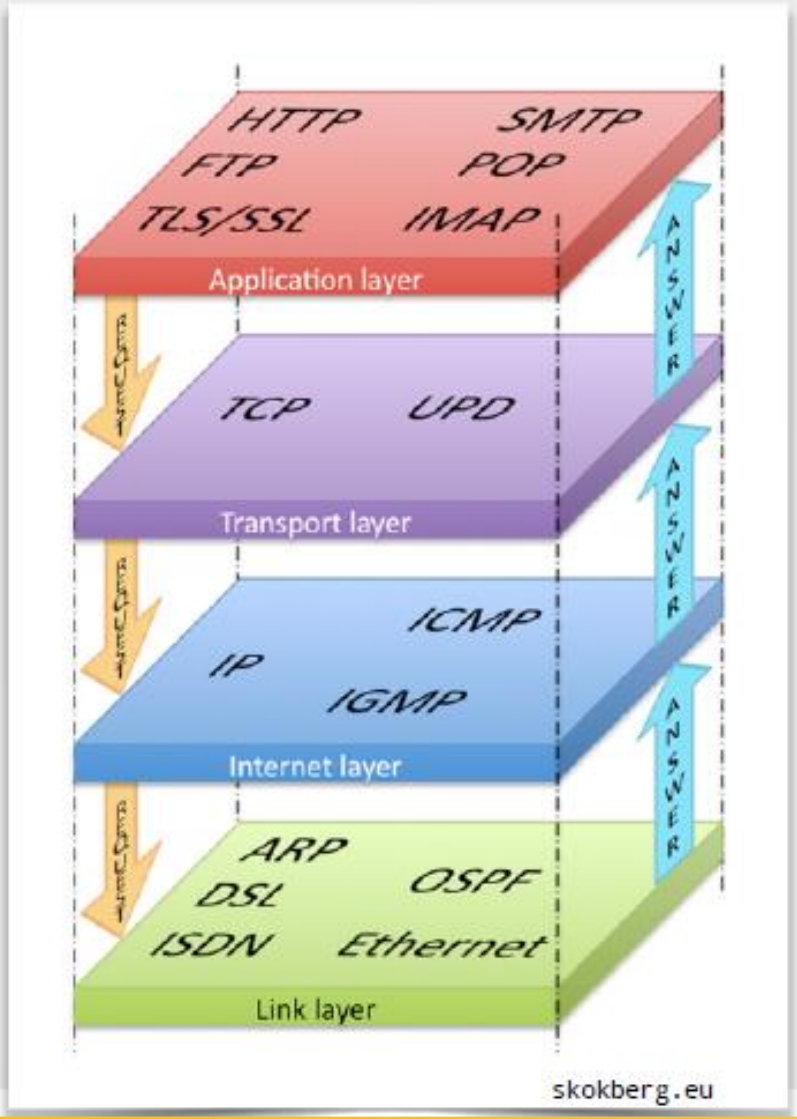
- Ces détails lui sont masqués du fait de la structure en couches de l'architecture des réseaux.

Modèle de référence OSI/ISO

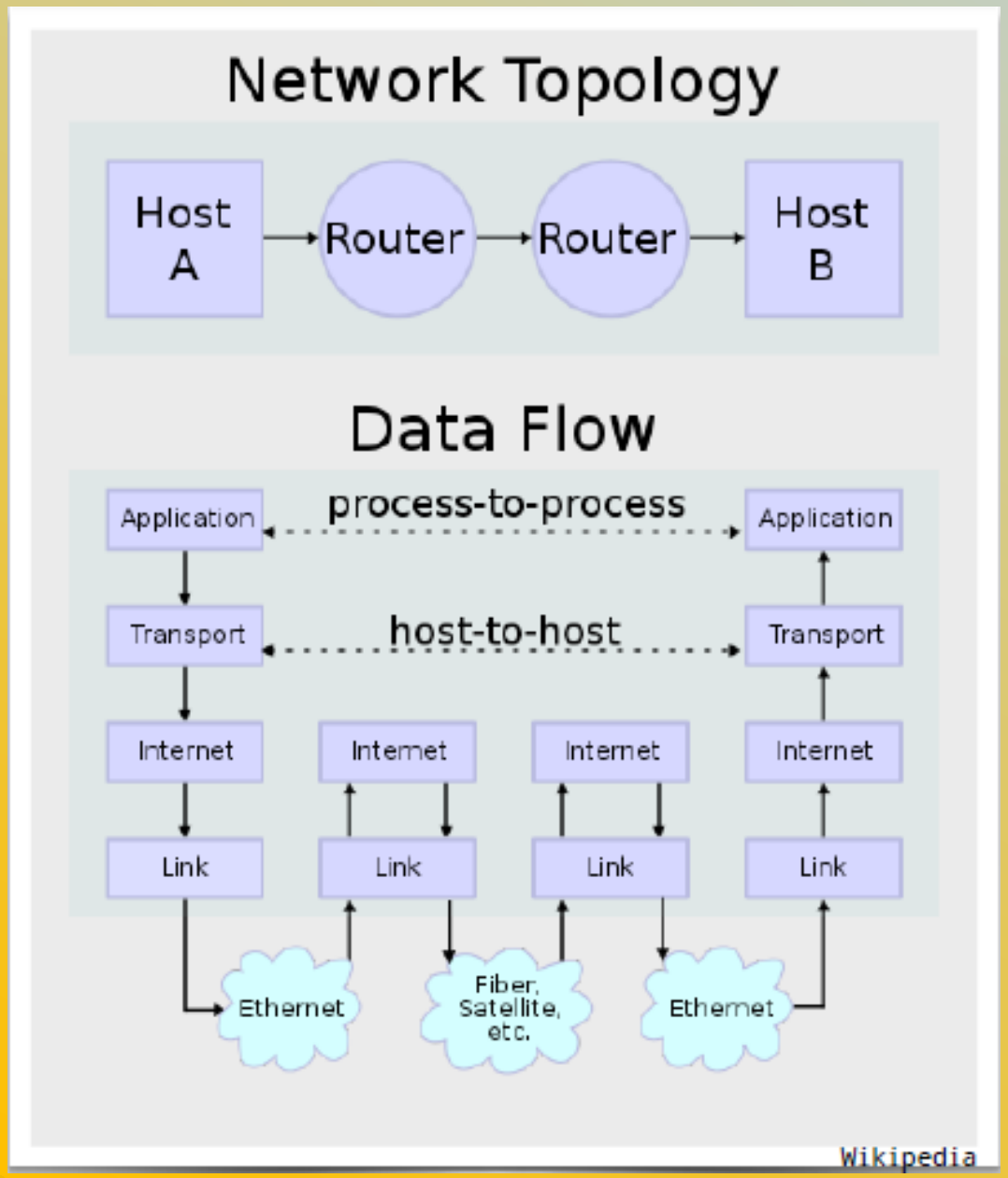


Couche	Protocoles
Application	FTP, HTTP, SMTP, POP, SSH, TELNET, IMAP…
Présentation	SSL, WEP, WPA, Kerberos
Session	Ports
Transport	TCP, UDP, SPX
Réseau	IPv4, IPv6, ARP, IPX
Liaison	802.11, WiFi, ATM, Ethernet, ISDN
Physique	Fibre, Câble, Radio…

- Le modèle Internet est à 4 couches



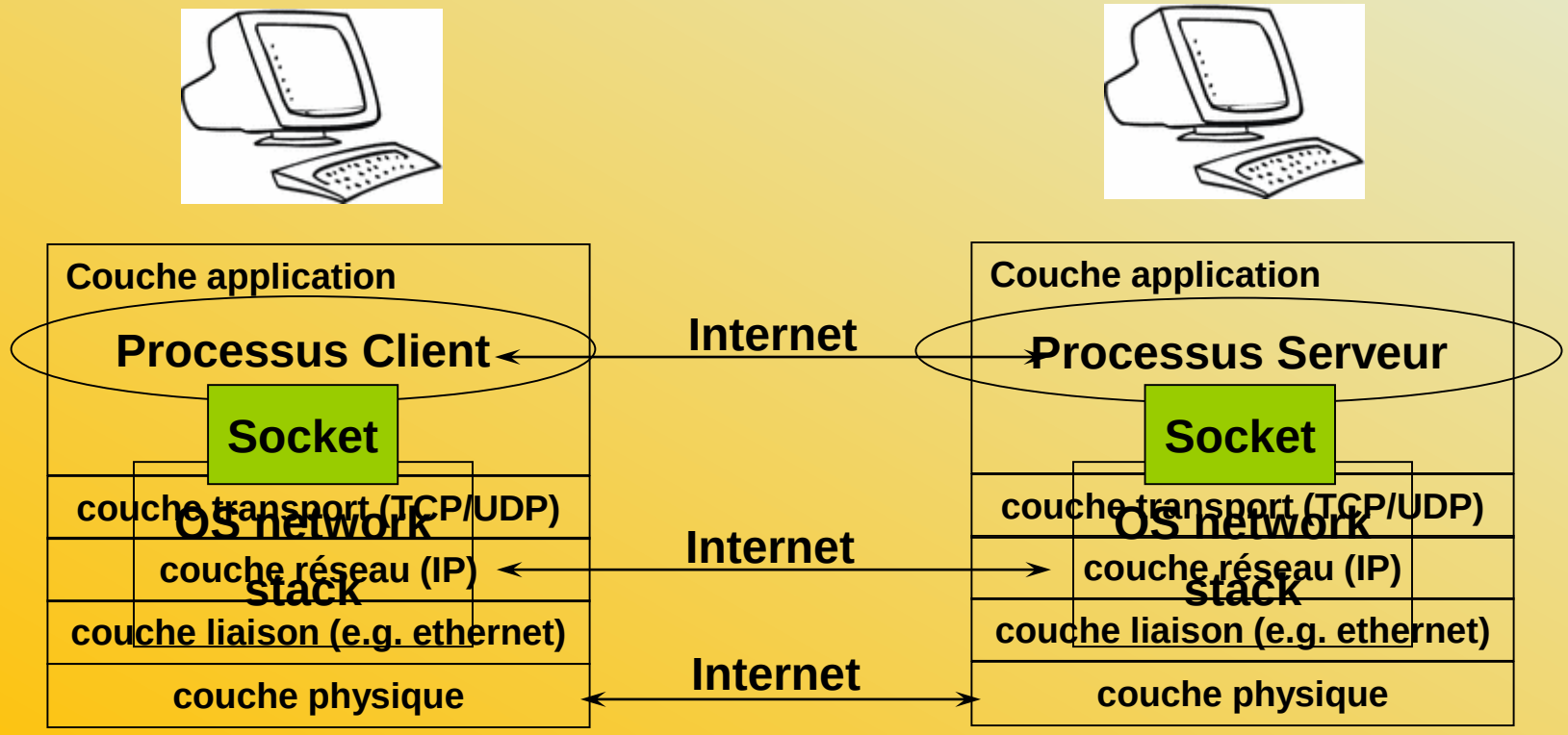
Couche	Protocoles
Application	FTP, HTTP, IMAP, POP…
Transport	TCP, UDP…
Internet	IPv4, IPv6, IPsec, ICMP…
Liaison	ARP, PPP, DSL, Ethernet…


Network Topology
Host A
Router
Router
Host B
Data Flow
Application
process-to-process
Application
Transport
host-to-host
Transport
Internet
Internet
Internet
Internet
Link
Link
Link
Link
Ethernet
Fiber, Satellite, etc.
Ethernet
Wikipedia

Communications Client/Serveur

- Client demande (*requête*) / serveur fournit (*résponse*)
- Typiquement: un seul serveur / multiple clients
- Le serveur n'a pas besoin de connaître quoi que ce soit sur le client (même pas qu'il existe)
- Le client doit toujours connaître quelque chose sur le serveur (au moins où il est localisé)

Communication inter-processus (IPC)



L' interface que l'OS fournit à son sous-système réseau

Communication inter-processus (IPC)

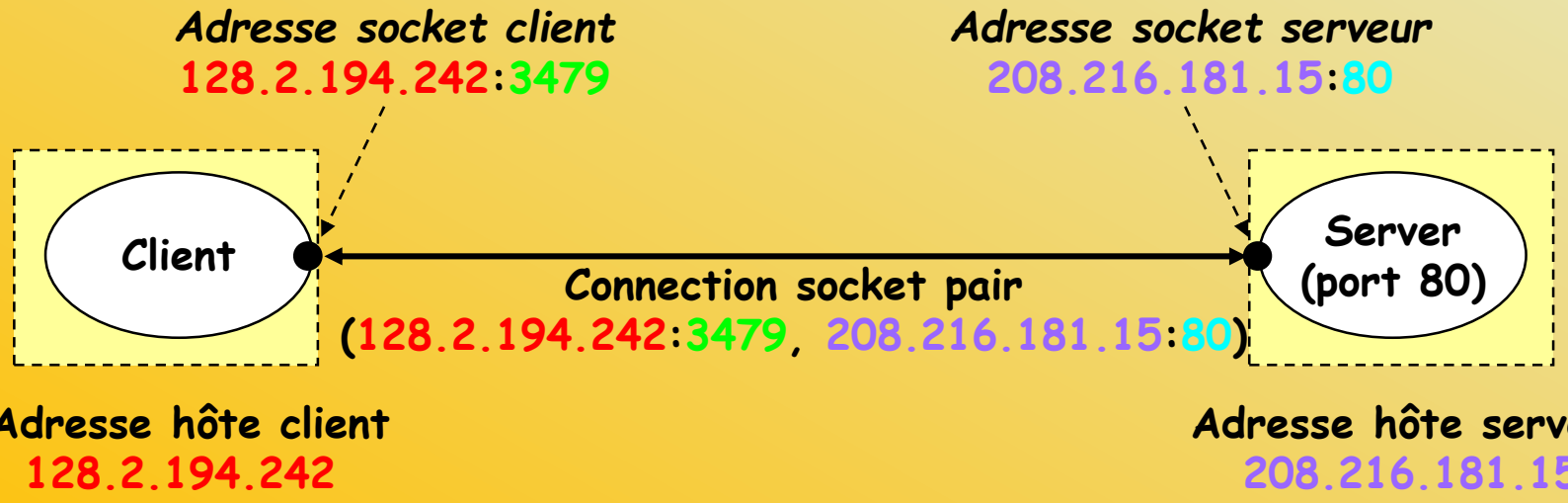
- La couche transport fournit un ou plusieurs services de communication entre applications (ex TCP ou UDP)
- Pour communiquer ces applications doivent se connaître
- Une application fournit un service particulier sur une machine donnée
- Le service est identifié par un port
- La machine est identifiée par une adresse

Communication inter-processus (IPC)

- Un couple (adresse, port) est un point de communication
- Une communication s'effectue entre au moins deux points de communication(l'émetteur et le -ou les- récepteurs).

Connections Internet (TCP/IP)

- Adresse de la machine sur le réseau
  - Par l'adresse IP
- Adresse du processus (ou service sur la machine)
  - Par le numéro de "port" – un nombre sur 16bits
- La pair *adresse IP + port* – constitue une "adresse-*socket*"



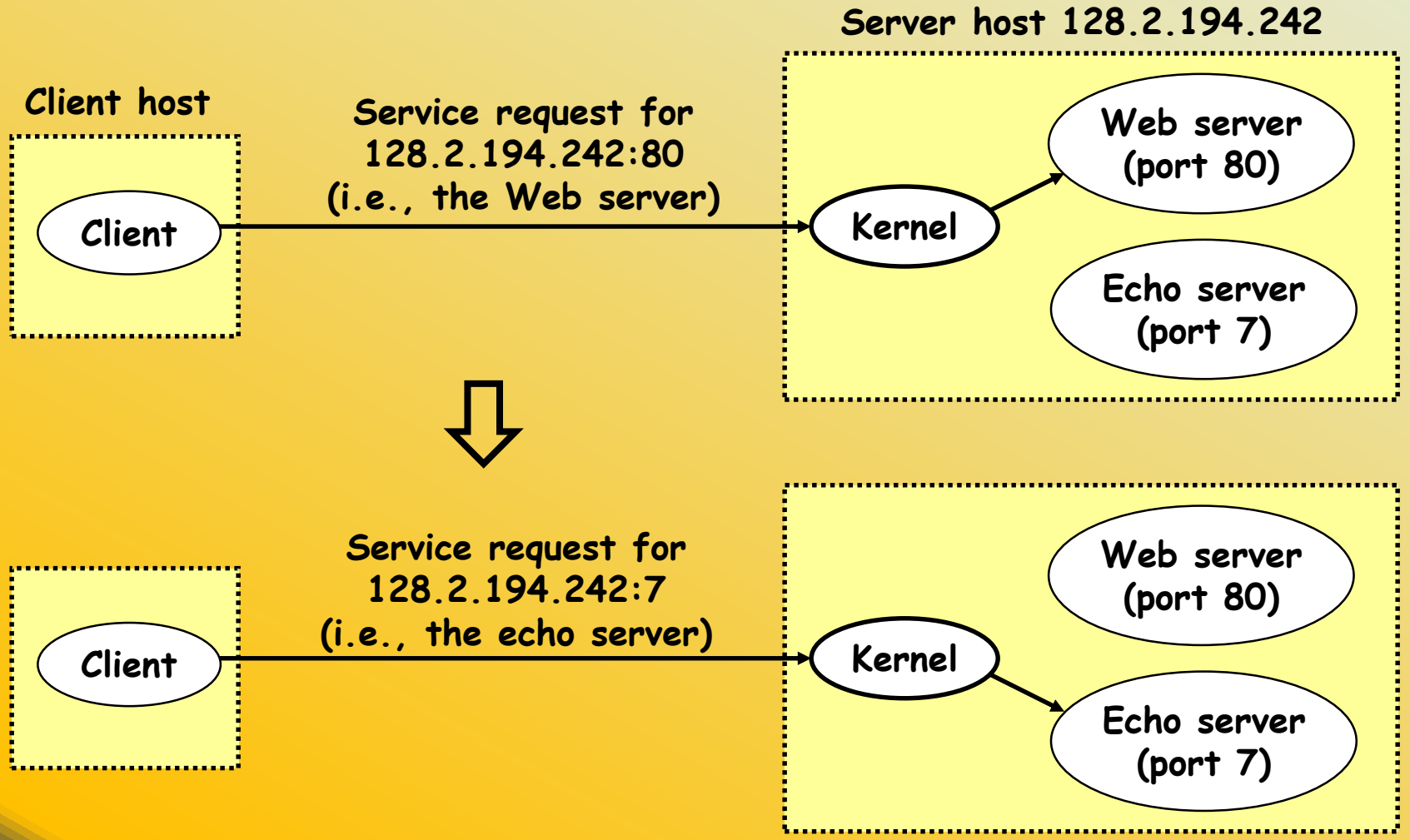
Note: 3479 est un port éphémère alloué par le noyau

Note: 80 est un port bien connu associé avec des serveurs Web

Clients

- Examples de programmes clients
  - Web browsers, `ftp`, `telnet`, `ssh`
- Comment un client trouve-t-il le serveur?
  - L'adresse IP dans l'adresse socket du serveur identifie le hôte
  - Le (bien-connu) port dans l'adresse socket du serveur identifie le service, et ainsi identifie implicitement le processus serveur qui rend ce service.
  - Examples ports bien-connus (entre 1 et 1023)
    - Port 7: Echo server
    - Port 23: Telnet server
    - Port 25: Mail server
    - Port 80: Web server

Utilisation des ports pour identifier les services



Serveurs

- Les Serveurs sont des processus (deamons).
  - Crées au moment du boot (typiquement)
  - S'exécutent continuellement jusqu'à ce que la machine soit éteinte.
- Chaque serveur se met en attente de l'arrivée de requêtes sur un port bien-connu associé à un service particulier
  - Port 7: echo server
  - Port 23: telnet server
  - Port 25: mail server
  - Port 80: HTTP server
- Les autres applications doivent choisir un port entre 1024 and 65535

I - Les Streams

I.1 - Les Streams

- Une remarque frappante que l'on peut faire lorsqu'on examine des applications réseaux fonctionnelles est le peu de code dédié aux aspects réseaux eux-mêmes.
- La part d'un programme dédiée aux aspects réseaux est presque toujours la plus courte et la plus simple.
- Pour les applications Java, par exemple, il est facile d'envoyer et de recevoir des données à travers Internet.

- Une grande partie de ce qu'un programme réseau fait sont des entées et des sorties : déplaçant des octets d'un système à un autre.
- Dans une large mesure, lire des données envoyées par un serveur n'est pas du tout différent de la lecture à partir d'un fichier.
- Envoyer un texte à un client n'est pas tellemment différent de l'écriture dans un fichier.

- Les E/S dans Java sont basées sur la notion de Streams

- Un stream est une séquence ordonnée de bytes.
- On peut penser à un stream comme une structure de donnée qui permet de lire ou écrire de manière séquentielle sans possibilité de retours arrière.
- Les streams sont des abstractions qui permettent l'accès à des ressources externes sans beaucoup se soucier de la ressource spécifique.
- L'utilisation de la bibliothèque des streams est un processus à deux étapes: i) création de l'objet stream ii) lecture ou écriture des données.

- Les streams d'entrée lisent des données; les streams de sortie écrivent des données.

- Différentes classes de stream, comme java.io.FileInputStream and sun.net.TelnetOutputStream lisent et écrivent des sources de données particulières.

- Cependant, tous les streams de sortie ont les même méthodes de base pour écrire des données et tous les streams d'entrée ont les même méthodes de base pour lire des données.

- Les streams sont synchrones; c.à.d, lorsqu'un programme (en réalité un thread) demande à un stream de lire ou écire un bout de données, il attend que les donnée soient lues ou écrites avant de faire quoi que ce soit d'autre.

- Java offer aussi les E/S non blocantes utilsant des cannaux et des tampons.

- Les E/S non blocantes sont un peu plus compliquées, mais peuvent être beaucoup plus rapides dans certaines applications à volume élevé, comme les serveurs web.

I.2- Les Filters Streams

- Les Filter streams peuvent être chainés soit à un input stream soit à un output stream.

- Les Filters peuvent modifier les données lorsqu'ells sont lues ou écrites, par exemple, en les cryptant ou compressant, ou bien ils peuvent simplement fournir des méthodes additionnelles pour convertir les données en d'autre formats.

- Par exemple, la classe java.io.DataOutputStream fournit une méthode qui convertis un int en quatre bytes et écrit ces bytes dans l'output stream sous-jacent.

I.3 - Les readers et writers

- Les readers et writers peuvent être chainés aux input et output streams pour permettre aux programmes de lire ou écrire du texte (c.à.d. cacarctères) plutôt que des bytes.

- Proprement utilisés, les readers et writers peuvent prendre en charge une variété de codage de caractères, incluant des ensembles de caractères mulyibyte tells que UTF-8 ou SJIS.

I.4 - Output Streams

- La classe de sortie de base Java est java.io.OutputStream:

public abstract class OutputStream

- Cette classe fournit les méthodes fondamentales nécessaires pour écrire des données. Cellec-çi sont :
  - public abstract void **write(int b)** throws IOException
  - public void **write(byte[] data)** throws IOException
  - public void **write(byte[] data, int offset, int length)** throws IOException
  - public void **flush()** throws IOException
  - public void **close()** throws IOException

- Les sous-classes de OutputStream utilisent ces méthodes pour écrire les données sur des medias particuliers.

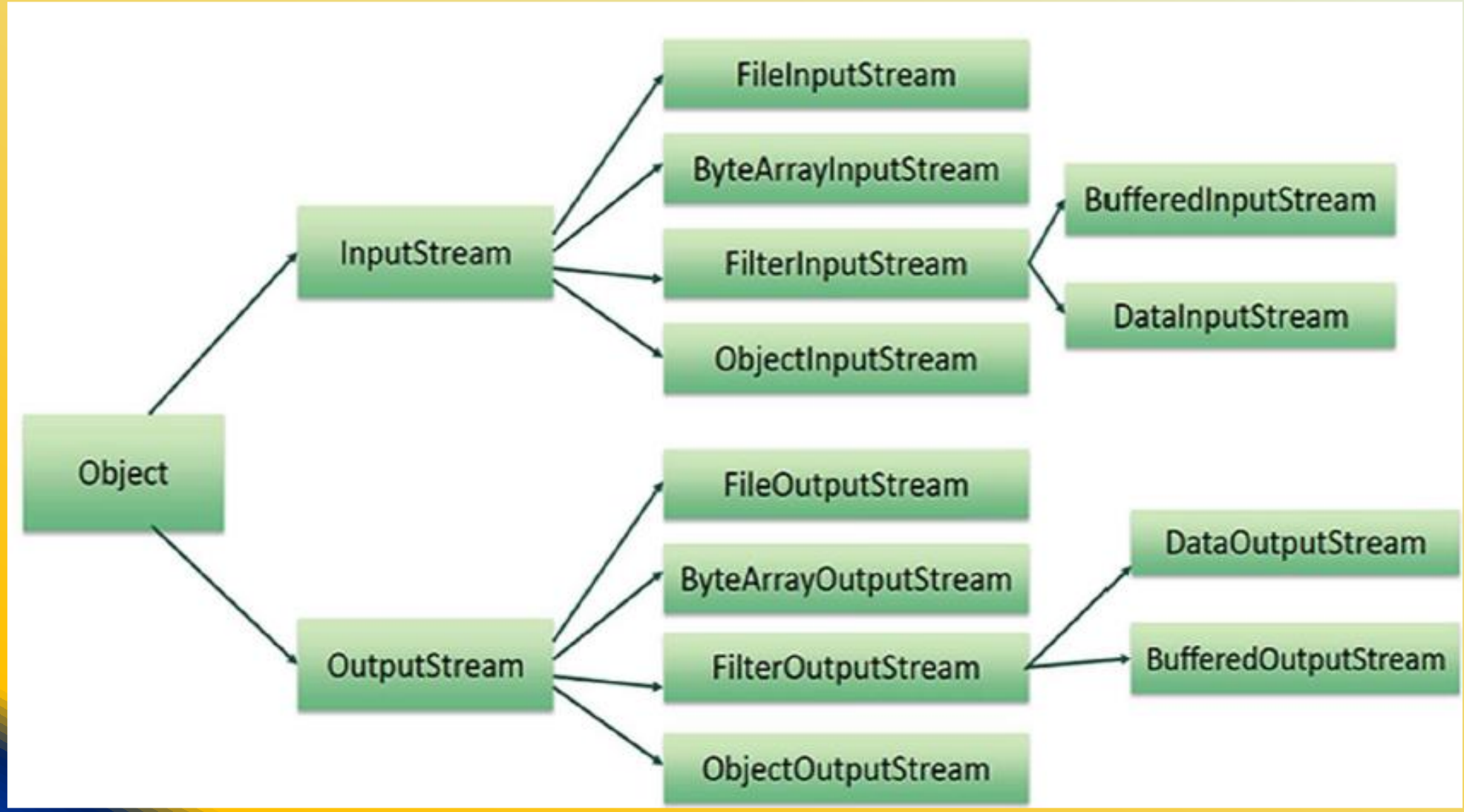
I.5 - Input Streams

- La classe de sortie de base Java est java.io.IntputStream:

public abstract class IntputStream

- Cette classe fournit les méthodes fondamentales nécessaires pour lire des bytes brutes. Cellec-çi sont :
  - public abstract **int read()** throws IOException
  - public **int read(byte[] input)** throws IOException
  - public **int read(byte[] input, int offset, int length)** throws IOException
  - public **long skip(long n)** throws IOException
  - public **int available()** throws IOException
  - public void **close()** throws IOException
- Les sous-classes de IntputStream utilisent ces méthodes pour lire les données à partir de medias particuliers.

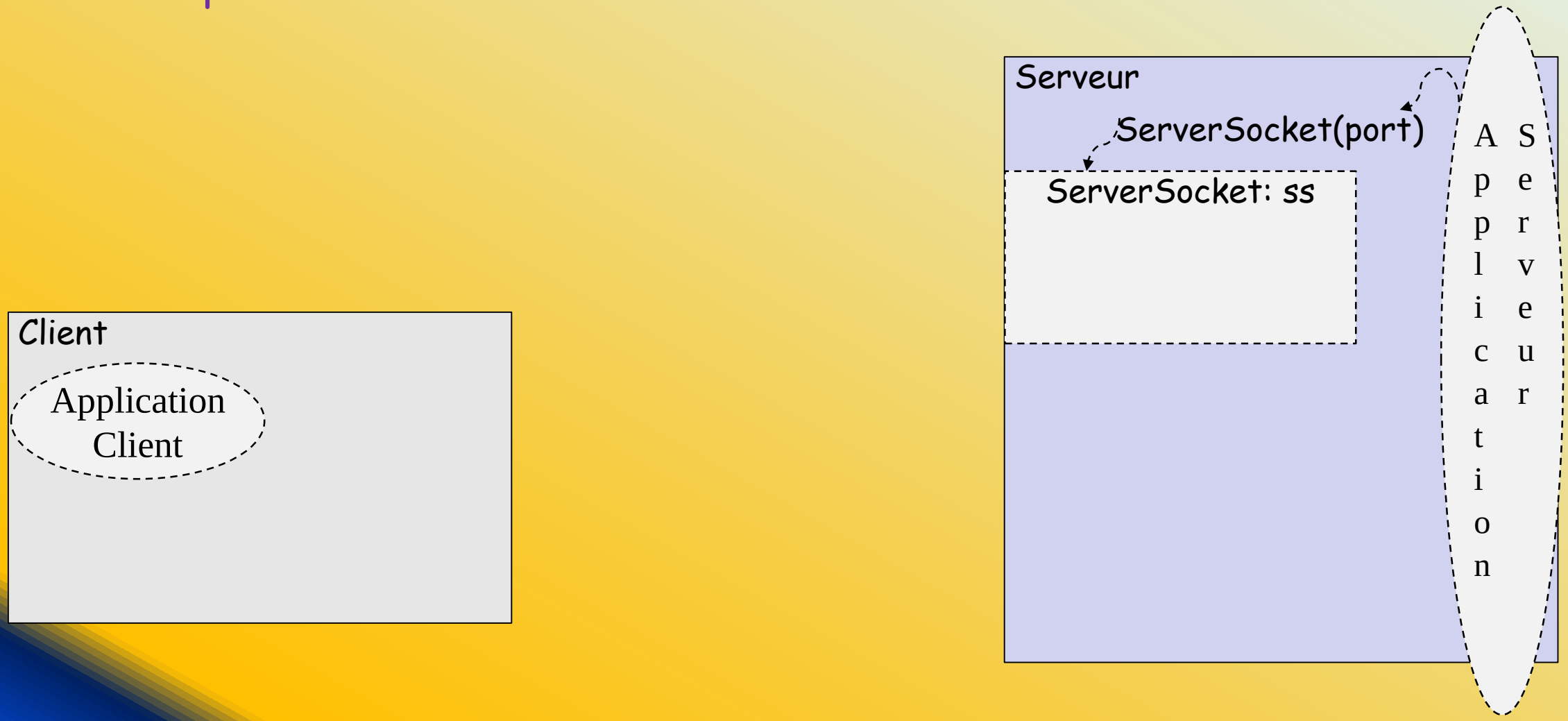
I.5 - Hierarchie des Streams



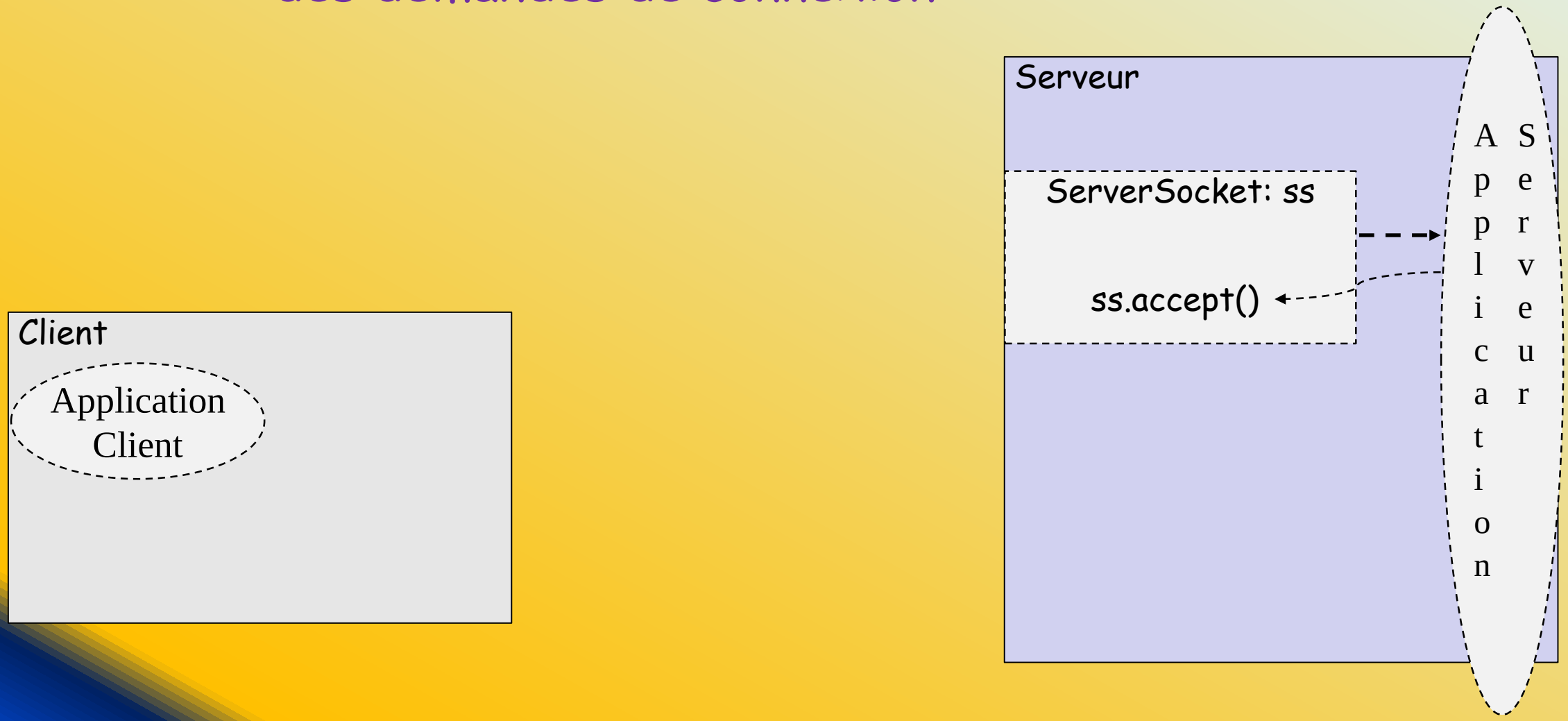
II-Communications basées sur les Sockets TCP

- Les sockets sont les terminaisons des connections entre machines hôtes et peuvent être utilisées pour envoyer et recevoir des données.
- Il y a deux types de sockets: *socket serveur* et socket client.
- *Socket serveur pour* recevoir les reqûetes des clients.
- Socket *client* peut être utilisé pour l'envoi et la réception des données.

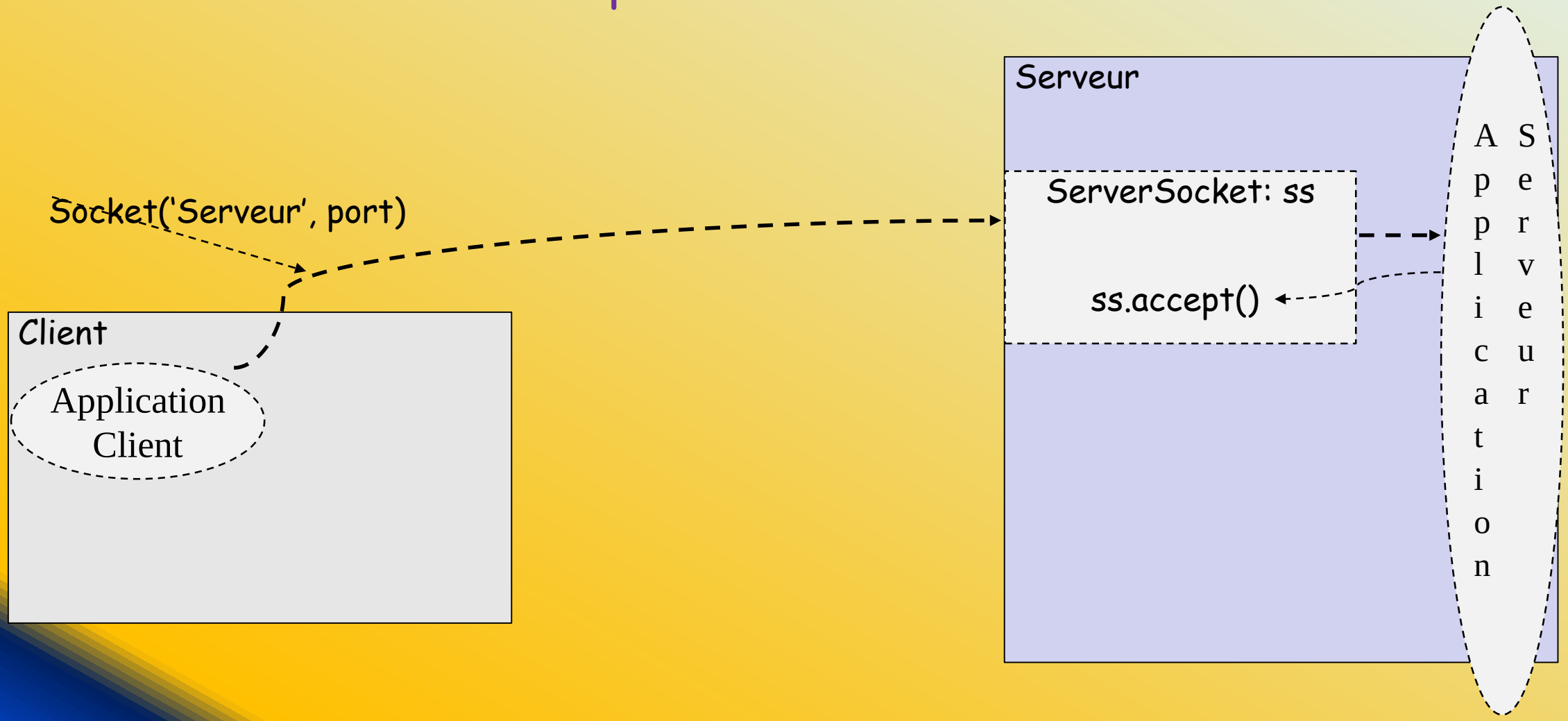
Le serveur commence par créer une ServerSocket pour recevoir les demandes de connexion



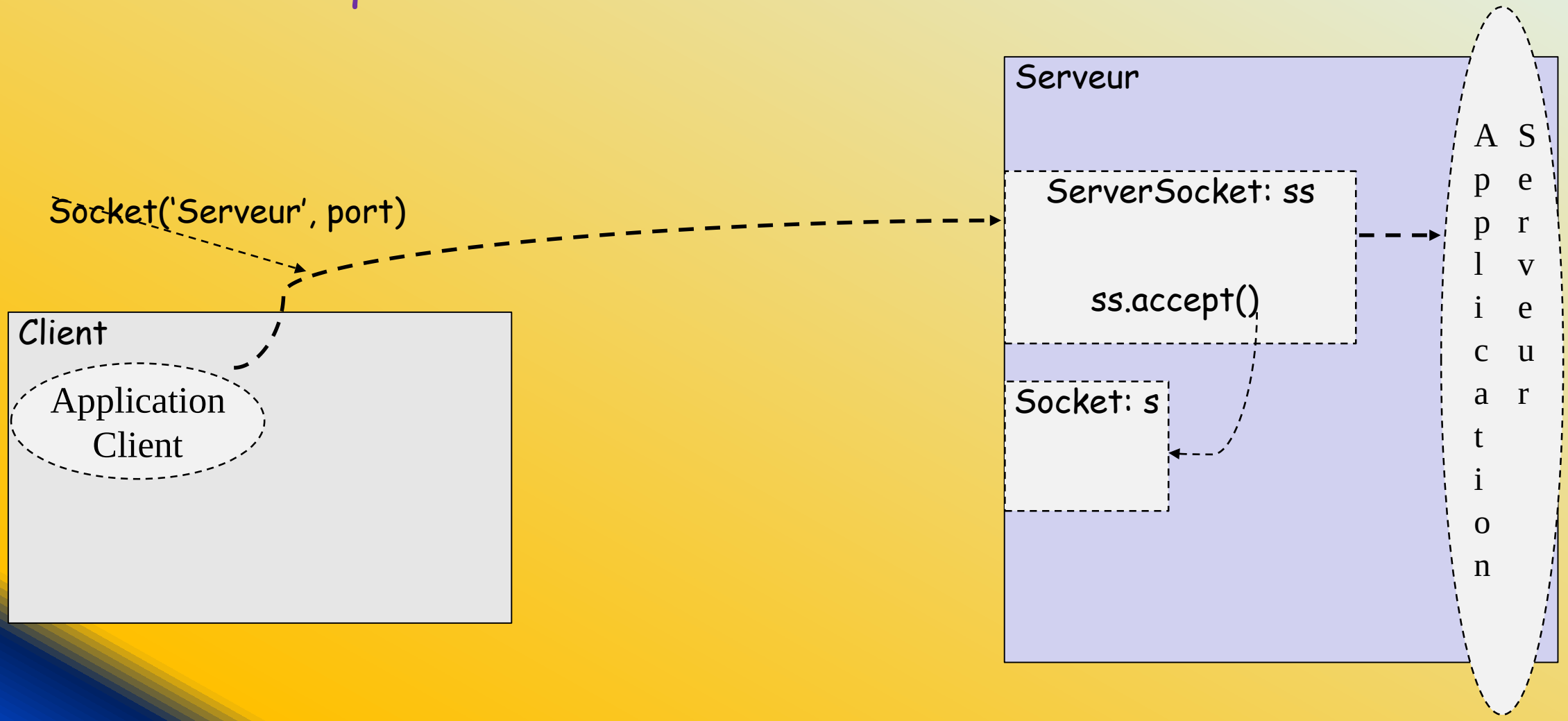
Le serveur se met en attente des demandes de connexion



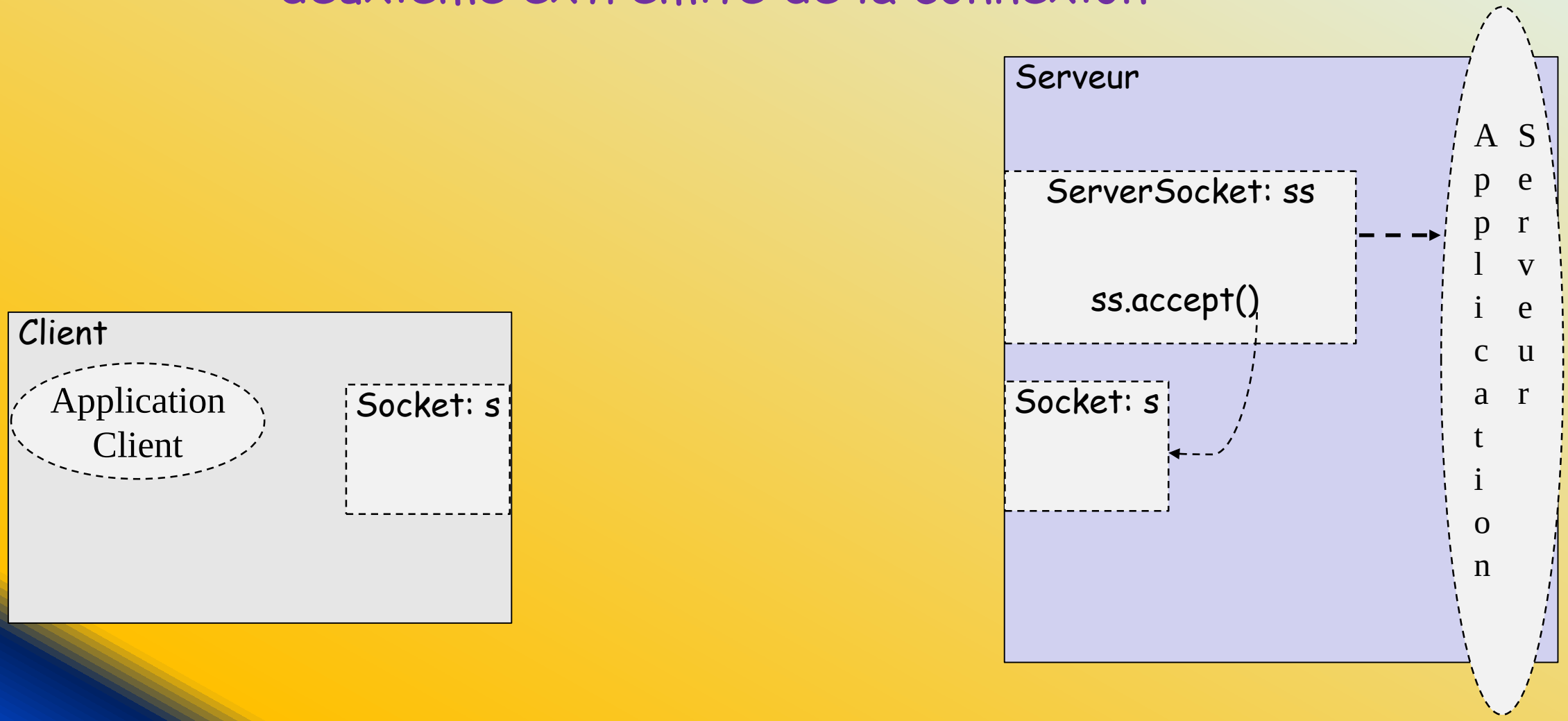
Le client demande une connexion à travers le port associé au service



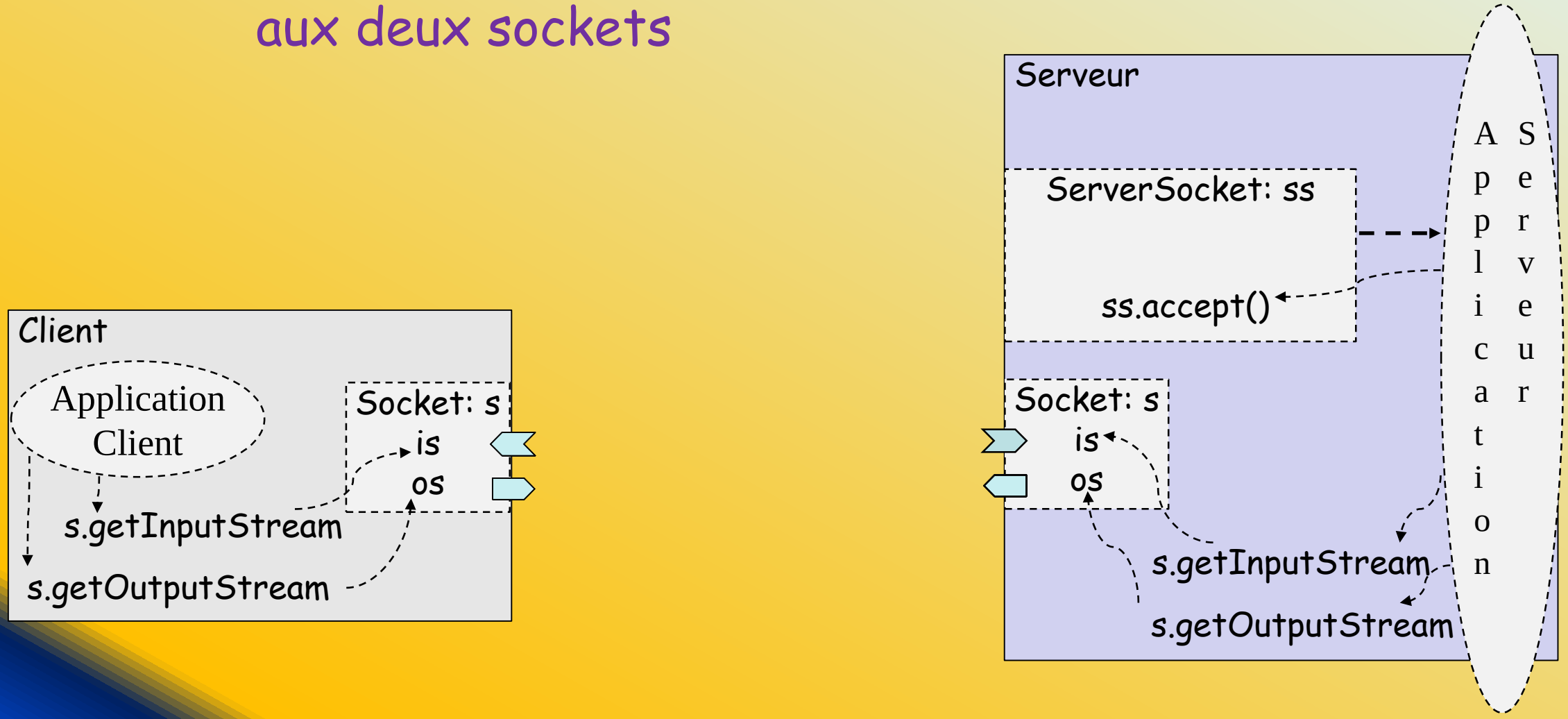
Le serveur crée une nouvelle socket qui sera l'une des extrémités de la connexion



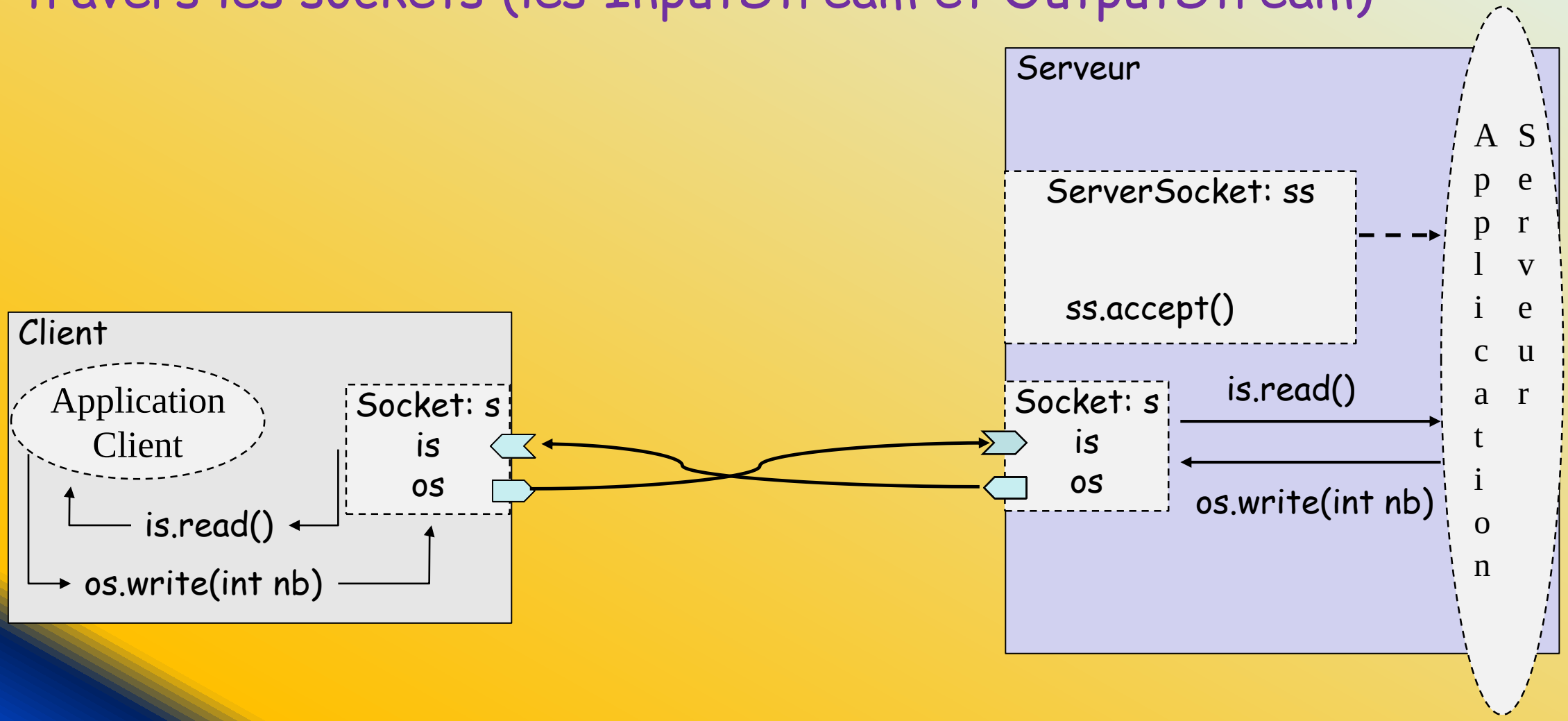
Le client obtient la socket qui est la deuxième extrémité de la connexion



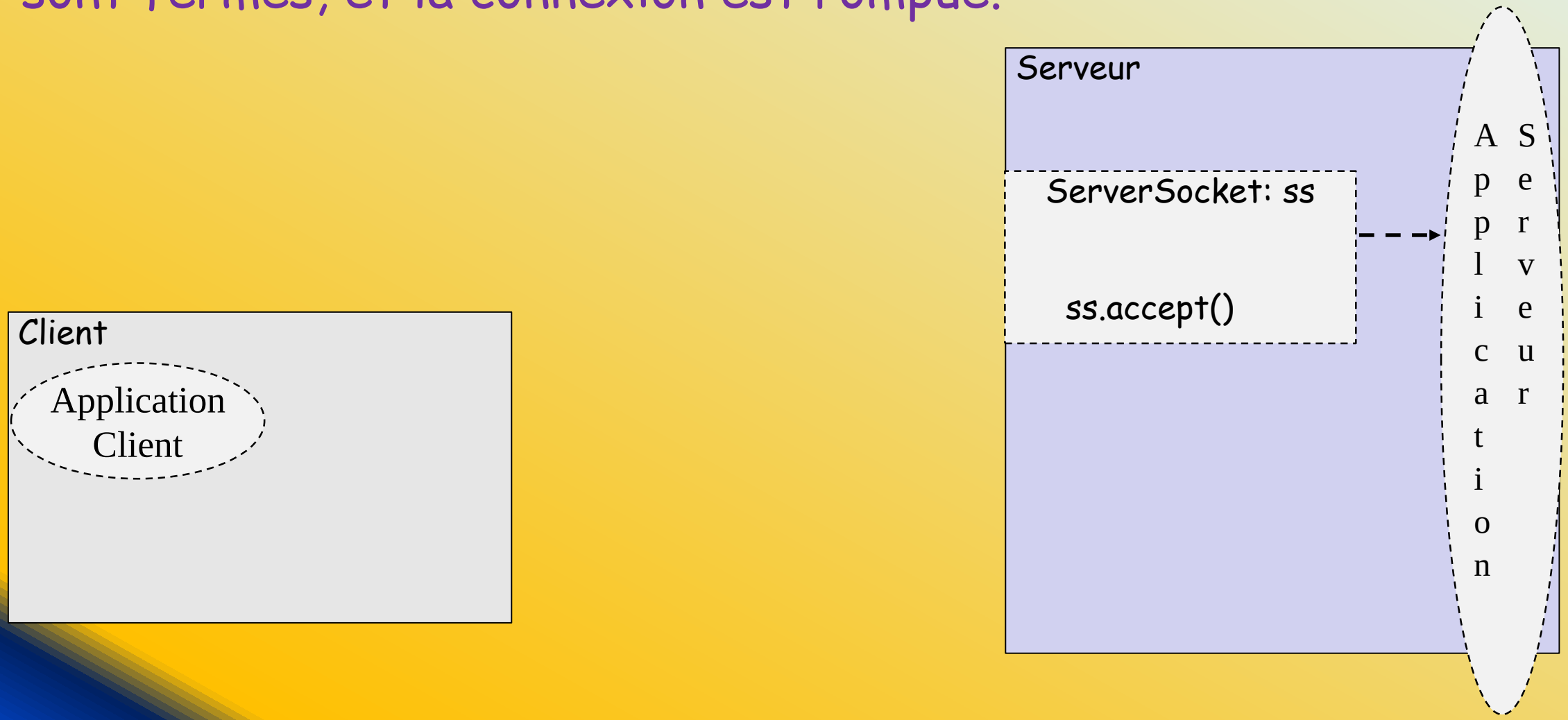
Le client et le serveur créent les InputStream et OutputStream associés aux deux sockets



Le client et le serveur commencent les échanges de données à travers les sockets (les InputStream et OutputStream)



A l'initiative de l'une des parties communicantes les sockets sont fermés, et la connexion est rompue.



II-1 Les Sockets Serveur

- Une socket serveur est une instance de la classe ServerSocket et peut être créee par l'un des ces constructeurs:

  ServerSocket(int port)

  ServerSocket(int port, int backlog)

- port: numéro de port sur lequel le serveur se mettra à l'écoute de requêtes de la part des clients.
- backlog: longeur maximum de la file des clients en attente d'être servis (50 par default).
- Les sockets serveur peuvent être créees avec les applications Java seulement, pas avec les applets.

II-1.2 Les méthodes des Sockets Serveur

- Socket accept()

  Attend une requête de connexion. Le thread qui exécute la méthode sera bloqué jusqu'à ce qu'une requête soit reçue, à ce moment la méthode retourne une socket client.

  Il y a deux faits importants à noter à propos de la méthode accept(). Le premier est que cette méthode est bloquante. Le deuxième est que cette méthode crée et retourne une instance de Socket.

- void close()

  Arrête de recevoir des requêtes des clients.!

II-1.2 Usage typique de ServerSocket

```
try {
        ServerSocket s = new ServerSocket(port);
    while (true) {
            Socket incoming = s.accept();
             «Traite un client»
            incoming.close();
    }
    s.close();
    } catch (IOException e) {
        «Traite l'exception» }
```

II-2 Les Sockets Client

- Une socket client est une instance de la classe Socket et peut être obtenue de deux manières:

(1) Côté serveur, comme valeur retournée par la méthode accept()

(2) Côté client en utilisant le constructeur

Socket(String host, int port)

host: l'adresse de la machine hôte

port: le numéro de port

II-2.1 Les méthodes des Sockets Client

- getInputStream()

  Retourne un objet InputStream pour recevoire les données(bytes)

- getOutputStream()

  Retourns un objet OutputStream pour envoyer des données(bytes)

- close()

  Ferme la socket(et par voie de conséquence la connexion)

II-2.2 Usage typique de Socket

```
try {
        Socket socket = new Socket(host, port);
        BufferedReader in = new BufferedReader(
        new InputStreamReader(socket.getInputStream()));
        PrintWriter out = new PrintWriter(
        new OutputStreamWriter(socket.getOutputStream()));
        «Envoie et reçoit les données»
        in.close();
        out.close(); socket.close();} catch (IOException e) {
        «Traite l'exception» }
```

III Les Threads

- Le problème fondamental avec la structure du programe serveur déjà présenté est que cette solution est difficilement extensible (eg. serveurs FTP et Web).

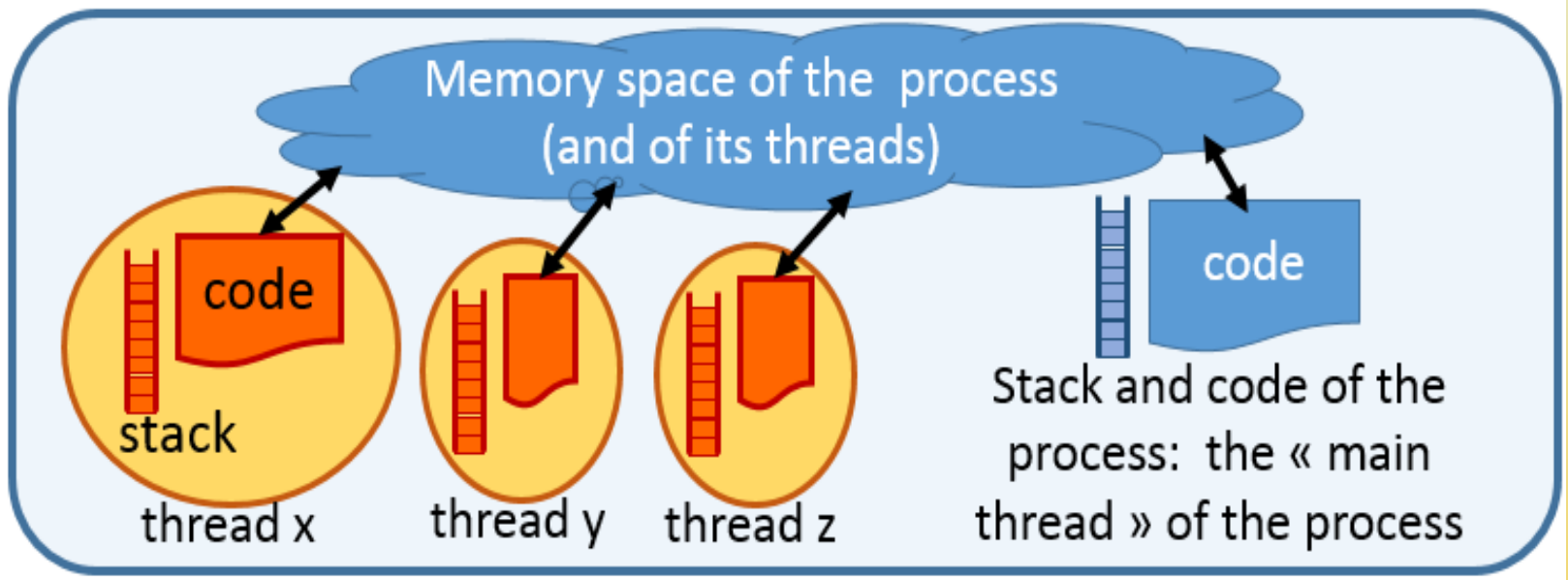
Bien qu'il soit relativement facil d'écrire du code qui traite chaque demande de connexion qui arrive et chaque nouvelle tache comme un processus séparé; quand le moment viendra où le serveur devra traiter un nombre élevé de connexions simultanées les performances s'écrouleront.

Il y a au moins deux solutions à ce problème.

1) La première solution est de réutiliser les processus au lieu d'en engendrer de nouveaux.

- Lorsque le serveur démarre, un nombre fixe de processus (disons 300) sont engendés pour traiter les requêtes.
- Les requêtes qui arrivent sont mises dans une file. Chaque processus retire une requête de la file, la sert, puis retire la requête suivante, etc.
- Il y toujours 300 processus séparés qui s'exécutent, mais parceque tout le travail supplémentaire pour construire et détruire les processus est évité, ces 300 processus peuvent maintenant faire le travail de 1000 processus.

2) La deuxième solution est d'utiliser des threads légers au lieu des processus lourds pour prendre en charge les connexions.



2) La deuxième solution est d'utiliser des threads légers

- Alors que chaque processus séparé a son prope bloc mémoire, les threads partagent le même epace mémoire.

- L'impact de l'exécution de plusieurs threads différents sur la machine du serveur est relativement minimal car tous les threads s'exécutent dans un processus.

- L'utilisation des threads apporte un gain de performance de l'ordre de trois par rapport à l'utilisation des processus.

- En combinant cela avec un pool de threads réutilisables le serveur peut devenir neuf fois plus rapide

Différences entre processus et threads (1)

- les processus sont plus lourds à créer que les threads
- les processus sont réellement indépendants
- les threads peuvent se partager des informations facilement
- les threads doivent eux-mêmes faire attention à ne pas se marcher sur les pieds

Différences entre processus et threads (2)

D'un point de vue programmation :

- pour passer un programme en multithread il faut avoir codé proprement et évité les variables globales
- pour passer un programme en multithread il faut faire attention aux accès en mémoire qui sont potentiellement partagés
- en multithread, fermer un fichier déjà ouvert le ferme pour tous les threads et pas seulement celui qui a appelé close()
- pour passer un programme en multiprocessus il suffit d'appeler fork
- en multiprocessus, fermer un fichier déjà ouvert ne le ferme que pour le fils qui a appelé close()

III-1 Exécuter des threads en Java

Pour lancer un nouveau thread s'executant sur la machine virtuelle, on construit une instance de la classe Thread et on invoke sa méthode start(), comme ceci :

```
Thread t = new Thread();
t.start();
```

Pour donner à ce thread quelque chose à faire, on doit:

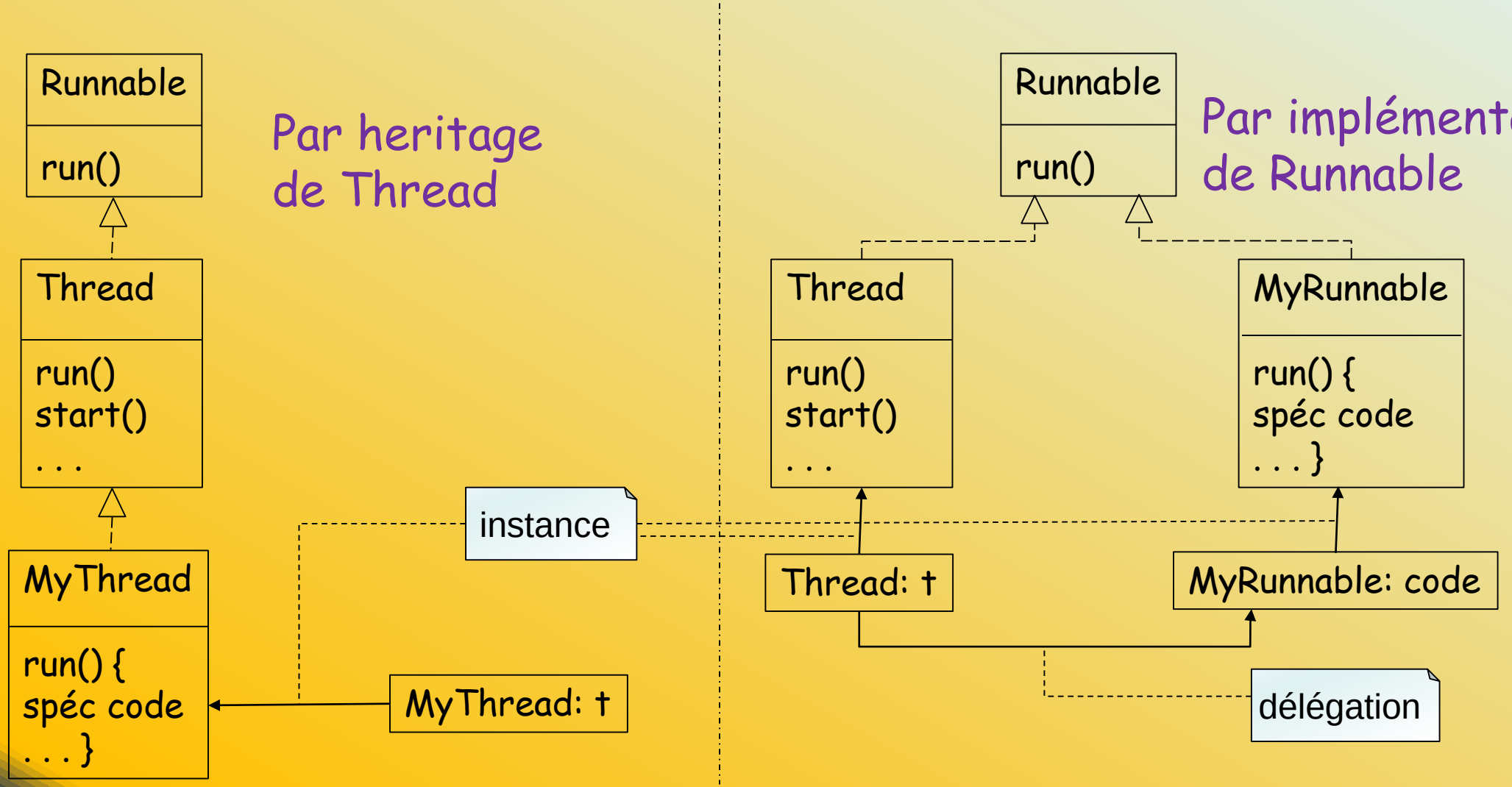
- soit créer une sous classe de Thread et surcharger sa méthode run()
- soit implémenter l'interface Runnable et passer l'objet Runnable au constructeur de Thread.

- Dans les deux cas, la clé c'est la méthode run(), qui a cette signature:

```
public void run()
```

- On va mettre tout le travail que le thread fait dans cette méthode.
- Cette méthode peut invoker d'autres méthodes; elle pourra construire d'autres objets; elle pourra même engendrer d'autres threads.
- Cependant, le thread commence là et s'arrête là aussi (dans run()).
- La méthode run() est au thread ce que la méthode main() est à un programme traditionnel.

III-2 Création de threads



III-2.1 Création par extension de la classe Thread

```
public class Multithread
{
    public static void main(String[] args)
    {
        int n = 8; // Number of threads
        for (int i=0; i<8; i++)
        {
            MthreadingDemo object = new MthreadingDemo();
            object.start();
        }
    }
}
```

```
class MthreadingDemo extends Thread {

    public void run()
    {
        try
        {
            // Affichage du thread qui s'exécute
            System.out.println ("Thread " +
                Thread.currentThread().getId() +
                " is running");

        }
        catch (Exception e)
        {
            // Renvoi d'une exception
            System.out.println ("Exception rencontrée");
        }
    }
}
```

III-2.2 Création par implémentation de Runnable

```
public class Multithread
{
    public static void main(String[] args)
    {
        int n = 8; // Number of threads
        for (int i=0; i<8; i++)
        {
          Thread object = new Thread(new MthreadingDemo());
          object.start();
        }
    }
}
```

```
class MthreadingDemo implements Runnable {

    public void run()
    {
        try
        {
            // Affichage du thread qui s'exécute
            System.out.println ("Thread " +
                  Thread.currentThread().getId() +
                  " is running");

        }
        catch (Exception e)
        {
            // Renvoi d'une exception
            System.out.println ("Exception rencontrée");
        }
    }
}
```

III-2.3 Différences entre Thread et Runnable dans Java

1. Chaque thread crée par extension de la classe Thread crée un objet unique ,pour lui, et il est associé avec cet objet. De l'autre côté, chaque thread crée par imlémentation de l'interface Runnable partage la même instance runnable.
2. Puisque chaque thread est associé à un objet unique lorsqu'il est crée par extension de la classe Thread, plus de mémoire est nécessaire. Alors que chaque thread crée par implementation de l'interface Runnable partage le même espace de l'objet donc, cela nécessite moins de mémoire.

3. Si on étend la classe Thread alors on ne pourra heriter aucune autre classe puisque Java ne permet pas l'heritage multiple alors que, implémenter Runnable donne une chance pour une classe d'heriter n'importe quelle autre.
4. On ne doit étendre la classe Thread que si on doit surchager ou spécialiser quelques autres méthodes de la classe Thread. On doit implementer l'interface Runnable si on veut spécialiser la méthode run() uniquement.

5. Etendre la classe Thread introduit un couplage fort dans le code puisque le code de Thread et le travail du thread est contenu dans la même classe. Implémenter l'interface Runnable introduit un couplage faible dans le code puisque le code de Thread est séparé du travail assigné au thread.

III-3 Serveur multi-threads

Il suffit de déléguer l'exécution du service à un Thread dédié…

```
// le service
class Service implements Runnable {
public Socket maSocket;
Service(Socket s) {
this.maSocket = s;}
void run() {
// code du service…
maSocket.close();
}
}
```

```
// squelette de serveur
ServerSocket socketAttente;
socketAttente = new ServerSocket(PORT);
do { // établissement d'une connexion (attente bloquante)
Socket s = socketAttente.accept();
// la communication est désormais possible, création du service
Thread t = new Thread(new Service(s));
// on démarre l'exécution concurrente du service
t.start();
} while (true);
socketAttente.close();
```

Extension de la classeThread …

```
// le service
class Service extends Thread {
public Socket maSocket;
Service(Socket s) {
this.maSocket = s;}
void run() {
// code du service…
maSocket.close();
}
}
```

```
// squelette de serveur
ServerSocket socketAttente;
socketAttente = new ServerSocket(PORT);
do { // établissement d'une connexion (attente bloquante)
Socket s = socketAttente.accept();
// la communication est désormais possible, création du service
Thread t = new  Service(s);
// on démarre l'exécution concurrente du service
t.start();
} while (true);
socketAttente.close();
```

VI - Programmation par Socket UDP

- Jusqu'a present, nous avons discuté des applications réseaux qui s'exécutent au dessus du protcole TCP, de la couche transport.
- TCP est conçu pour des transmissions fiables des données.

  Si des paquets arrivent en désordre, TCP les remet en ordre.

  Si des données sont perdues ou endommgées, TCP s'assure qu'elles seont retransmises.

  Si les données arrivent trop rapidemment pour la connexion, TCP réduit la vitesse pour que les paquets ne soient pas perdus.
- Un programe n'a pas à se soucier des données reçues qui sont incorrectes ou en désordre.
- Cependant, cette fiabilité a un prix. Ce prix est la vitesse.

  Etablir et couper une connexion TCP peut prendre un temps non négligeable, surtout pour des protocoles tels que HTTP, qui tendent à néccesiter plusieurs transmissions courtes.

- Le protocole UDP (un autre protcole, de la couche transport) est une alternative pour l'envoi de données sur IP de manière trés rapide, mais non fiable

.

- Lorsqu'on envoie des données UDP nous n'avons aucun moyen de savoir si ells sont bien arrivées; encore moins de savoir si les données qui arrivent sont bien dans l'ordre dans lequel elles ont été envoyées.

- Cependant, les morceaux de données qui sont reçues arrivent généralement rapidemment.

- La question qu'on peut se poser, alors, est: pourquoi quelqu'un voudrait-il utiliser un protocole non fiable?
- Il y a plusieurs types d'applications dans lequelles la vitesse brute est plus importante que le fait de recevoir chaque bit correctement.

  Par exemple, dans l'audio ou la video temps réel , les paquets de données perdus ou intervertis apparaissent simplement comme statiques. Ce phénomène est tolérable, mais les pauses génantes dans le stream audio, lorsque TCP demande une retransmission ou attends un paquet qui tarde à arriver, sont inacceptables.
- Dans d'autres applications, les tests de fiabilité peuvent être réalisés dans la couche application.

  Par exemple, si un client envoie une courte requête UDP à un serveur, il peut supposer que le paquet est perdu si aucune réponse n'est retournée au bout d'un certain lapse de temps; à la manière du DNS (il peut aussi opérer sur TCP).

- Java supporte la communication par Datagrammes à travers les deux classes suivantes :

  DatagramPacket

  DatagramSocket

- La classe DatagramPacket compacte des octets de données dans des paquets UDP appelés datagrammes et permet de décompacter les datagrammes reçus.
- La classe DatagramSocket envoie et recoit des datagrammes UDP. Pour envoyer des données, on met les données dans DatagramPacket et on envoie le paquet en utilisant DatagramSocket. Pour recevoir des données, on prend un objet DatagramPacket d'un DatagramSocket, puis on inspecte le contenu du paquet.

- Dans UDP, tout ce qui concerne un datagramme, y compris l'adresse à laquelle il est destine, est inclu dans le paquet lui même, la socket a, seulement, besoin de connaître le port local sur lequel écouter ou envoyer.
- Un seul DatagramSocket peut envoyer des données à, et recevoir des données de, plusieurs machines indépendantes. La socket n'est pas dédiée à une seule connexion, comme elle l'est dans TCP. En fait, UDP n'a aucune notion d'une connexion entre deux machines hôtes.
- Les sockets TCP traitent une connexion réseau comme un stream. UDP ne supporte pas cela; on travaille toujours avec des paquets individuels. Un paquet n'est pas nécessairement relié au suivant. Etant donné deux paquets, il n'y a pas moyen de déterminer lequel fût envoyé en premier.

- La classe DatagramPacket utilise différents constructeurs selon que la packet est utilisé pour transmettre ou reçevoir des données.
- Les deux constructeurs qui créent un nouveau objet DatagramPacket pour recevoir des données du réseau sont :

```
public DatagramPacket(byte[] buffer, int length)
public DatagramPacket(byte[] buffer, int offset, int length)
```

- Lorsqu'une socket reçoit un datagramme, elle sauvegarde la partie données du datagramme dans buffer commencant à buffer[0] (ou buffer[offset] et continue jusqu'à ce que le paquet soit complètement sauvegardé ou que length octets soient écrits dans buffer.

- Les quatre constructeurs qui créent un nouveau objet DatagramPacket pour transmettre des données sur le réseau sont :

```
public DatagramPacket(byte[] data, int length,
       InetAddress destination, int port)
public DatagramPacket(byte[] data, int offset, int length,
       InetAddress destination, int port)
public DatagramPacket(byte[] data, int length,
       SocketAddress destination)
public DatagramPacket(byte[] data, int offset, int length,
       SocketAddress destination)
```

DatagramPacket a six méthodes qui récupèrent les différentes parties d'un datagramme: les données plus plusieurs champs de son entête. Ces méthodes sont utilisées principalement pour les datagrammes reçus du réseau.

- public InetAddress getAddress()

  Cette méthode est communément utilisée pour déterminer l'adresse de la machine hôte qui a envoyé un datagramme UDP, pour que le destinataire puisse répondre.

- public int getPort()

  Cette méthode retourne un integer qui spécifie le port distant.

- public SocketAddress getSocketAddress()

  Cette méthode retourne un objet SocketAddress qui contient l'adresse IP et le port de la machine hôte distante.
- public byte[] getData()

  Cette méthode retourne un tableau d'octets contenant les données du datagramme.
- Il est souvent nécessaire de convertir les octets en quelque autre forme de données avant qu'ils puissent être utiles pour notre programme.

  Une manière de faire est de changer le tableau d'octets en une chaîne de caractères(String) `String s = new String(dp.getData(), "UTF-8");`

- Si le datagramme ne contient pas du texte, le convertir en des données Java est plus difficile.

  Une approche est de convertir le tableau d'octets retourné par `getData()` en un `ByteArrayInputStream`. Par exemple :

```
InputStream in = new ByteArrayInputStream(packet.getData(),
                                          packet.getOffset(), packet.getLength());
```

  On doit spécifier offset et length.

  Le `ByteArrayInputStream` peut alors être chaîné à un DataInputStream:

```
DataInputStream din = new DataInputStream(in);
```

  Les données peuvent être lues en utilisant les methods de `DataInputStream`: `readInt()`, `readLong()`, `readChar()`, et autres.

- public int getLength()

  Cette méthode retourne le nombre d'octets de données dans le datagramme. Ce n'est pas nécessairement le même que la longeur du tableau retournée par getData().

- public int getOffset()

  Cette méthode retourne simplement le point dans le tableau retourné par getData() où les données du datagramme commencent.

- Pour échanger des DatagramPacket, on doit ouvrir une socket de datagramme. En Java, une telle socket est créée et accèdée à travers la classe DatagramSocket.
- Tout les sockets de datagramme s'attachent à un port local, sur lequel ils se mettent à l'écoute de données arrivantes et qu'ils placent dans l'entête des datagrammes partants.
- IL n'y a pas de distinction entre les sockets client et les socket serveur, comme cela est la cas avec TCP.

  Le seul détail qui les différencie est que dans la cas d'une socket client le port est anonyme(assigné par le système) alors que dans le cas d'une socket serveur le port doit être connu des clients.

- Les constructeurs de DatagramSocket sont utilisés dans différentes situations, comme ceux de DatagramPacket.
- public DatagramSocket() throws SocketException

  Ce constructeur crée une socket qui est attachée à un port anonyme (pas de souci pour trouver un port libre). Il est utilisé pour un client qui initie une conversation avec un serveur.
- public DatagramSocket(int port) throws SocketException

  Ce constructeur crée une socket pour se mettre à l'écoute des datagrammes qui arrivent sur port particulier, spécifié par l'argument port. Utilser ce constructeur pour écrire un serveur qui écoute sur un port bien connu.

- public DatagramSocket(int port, InetAddress interface) throws SocketException

  Ce constructeur crée une socket pour se mettre à l'écoute des datagrammes qui arrivent sur un port et une interface réseau spécifiques.

- public DatagramSocket(SocketAddress interface) throws SocketException

  Similaire au precedent sauf que l'adresse de l'interface réseau et le port sont lus à partir d'une SocketAddress.

- La tache première de la classe DatagramSocket est d'envoyer et recevoir des datagrammes UDP.

  Une socket peu aussi bien envoyer que recevoir. En effet, elle peu envoyer et recevoir à et de multiple machines en même temps.

- public void send(DatagramPacket dp) throws IOException

  Envoie le paquet en le passant à la méthode send de la socket.

- public void receive(DatagramPacket dp) throws IOException

  Cette méthode recoit un seul datagramme UDP à partir du réseau et le sauvegarde dans un objet DatagramPacket préexistent. C'est une opération blocante.

  Si le programme fait autre chose en plus d'attendre les datagrammes, on doit penser à appeler receive() dans un thread séparé

- Un simple serveur UDP qui attend des requêtes d'un client, accepte le datagramme envoyé puis renvoie le même message est donné ci-dessous:
- // UDPServer.java: A simple UDP server program.
- import java.net.*;
- import java.io.*;
- Public class UDPServer {
- public static void main(String args[]){
- // args donne le ports sur lequel le serveur doit se mettre à l'écoute
- DatagramSocket aSocket = null;

```
if (args.length < 1) {
System.out.println("Usage: java UDPServer <Port Number>");
System.exit(1);}
try {
int socket_no = Integer.valueOf(args[0]).intValue();
aSocket = new DatagramSocket(socket_no);
byte[] buffer = new byte[1000];
while(true) {
DatagramPacket request = new DatagramPacket(buffer,
buffer.length);
```

```
aSocket.receive(request);
DatagramPacket reply = new DatagramPacket(request.getData(),
request.getLength(),request.getAddress(),
request.getPort());
aSocket.send(reply);  } }
catch (SocketException e) { System.out.println("Socket: " +
e.getMessage()); }
catch (IOException e) {System.out.println("IO: " + e.getMessage());}
finally {
if (aSocket != null)  aSocket.close();}
} }
```

```
// UDPClient.java: A simple UDP client program.
import java.net.*;
import java.io.*;
public class UDPClient {
public static void main(String args[]){
// args donne le contenu du message et le hostname du serveur
DatagramSocket aSocket = null;
if (args.length < 3) {
System.out.println(
"Usage: java UDPClient <message> <Host name> <Port number>");
System.exit(1);
```

```
}
try {
aSocket = new DatagramSocket();
byte [] m = args[0].getBytes();
InetAddress aHost = InetAddress.getByName(args[1]);
int serverPort = Integer.valueOf(args[2]).intValue();
DatagramPacket request =
new DatagramPacket(m, args[0].length(), aHost, serverPort);
aSocket.send(request);
byte[] buffer = new byte[1000];
```

```
DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
aSocket.receive(reply);
System.out.println("Reply: " + new String(reply.getData()));
}
catch (SocketException e) {
System.out.println("Socket: " + e.getMessage());
}
catch (IOException e) {
System.out.println("IO: " + e.getMessage());
}
finally {
if (aSocket != null)
aSocket.close();
                    }}}
```

V - Programmation par Sockets Multicast

Programmation par Sockets Multicast

Broadcasting

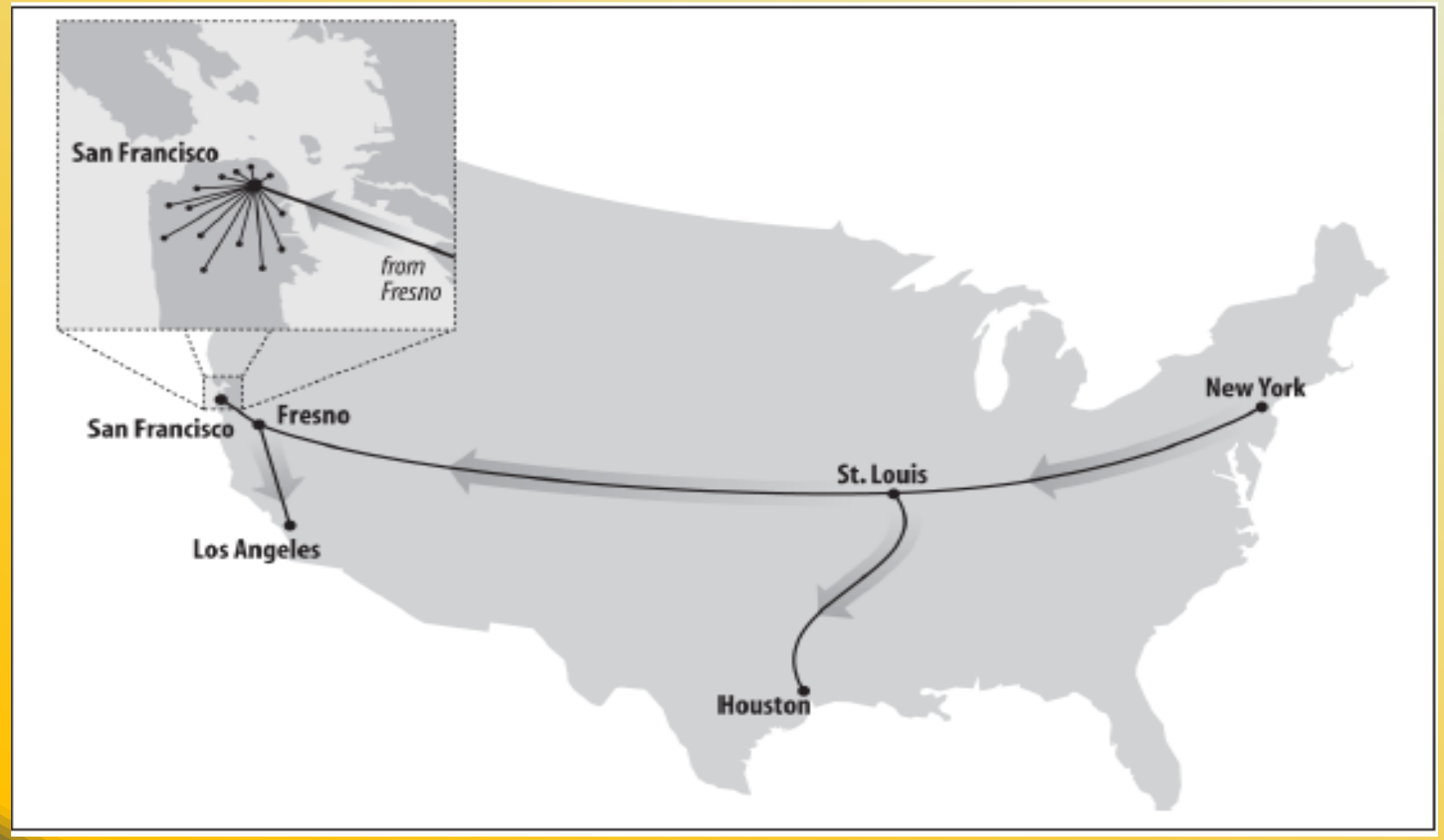
- IP supporte le broadcasting, mais son utilisation est strictement limitée.
- Même un petit nombre de broadcast globaux pourraient mettre l'internet à genoux.
- Imginez ce qui pourrait se produire si une video temps-reel etait copiées à tous les millions d'utilsateurs Internet, qu'ils souhaitent la voir ou non.

Programmation par Sockets Multicast

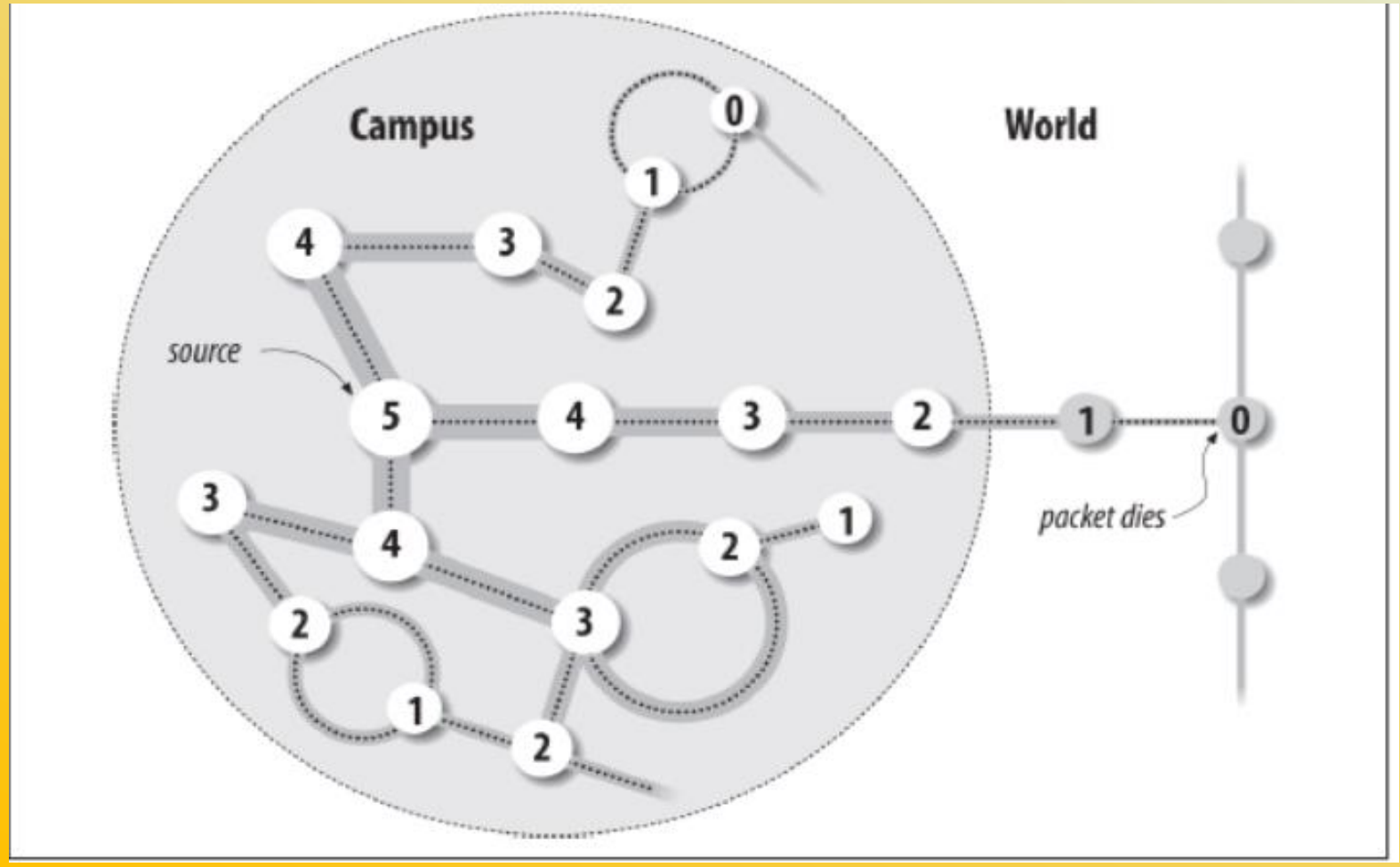
Broadcasting

- Il n'y a pas de raison d'envoyer une vidéo à des hôtes qui ne sont pas intéressés par celle-çi.

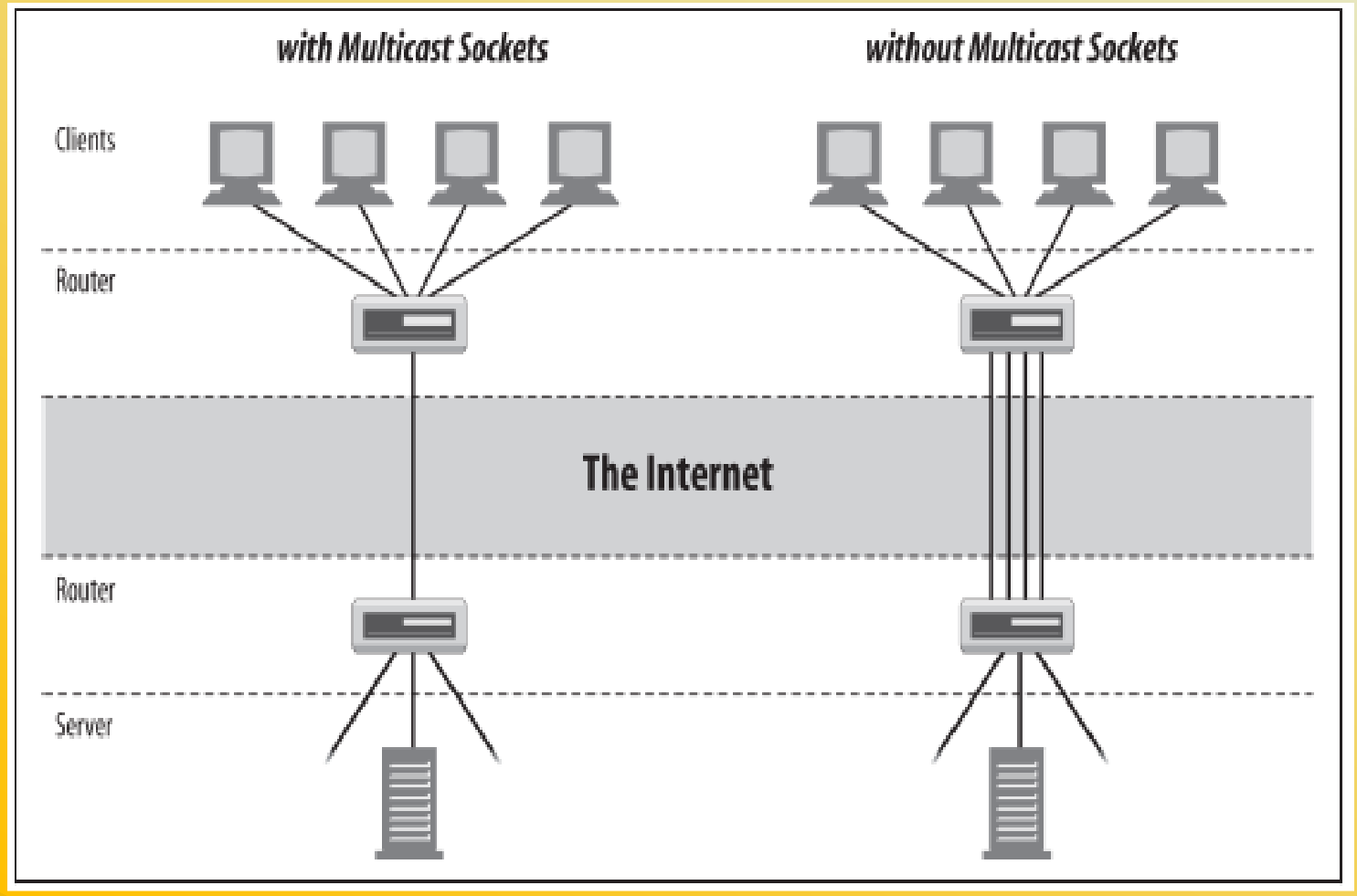
Programmation par Sockets Multicast



Programmation par Sockets Multicast



Programmation par Sockets Multicast



Programmation par Sockets Multicast

- Dans Java, le multicasting se fait à travers la classe java.net.MulticastSocket, qui est une sous classe de java.net.DatagramSocket:

  public class MulticastSocket extends DatagramSocket
  implements Closeable, AutoCloseable

- Comme on peut le prévoir, le comportement de MulticastSocket est similaire à celui de DatagramSocket:

  on met nos données dans un objet DatagramPacket qu'on peut envoyer ou recevoir avec MulticastSocket.

Programmation par Sockets Multicast

Les Constructeurs:

- Les constructeurs sont simples. On peut choisir un port sur lequel écouter ou laisser Java choisir pour nous un port anonyme:

```
public MulticastSocket() throws SocketException
public MulticastSocket(int port) throws SocketException
public MulticastSocket(SocketAddress bindAddress) throws
                                                    IOException
```

- Exemples:
- `MulticastSocket ms1 = new MulticastSocket();`
- `MulticastSocket ms2 = new MulticastSocket(4000);`
- `SocketAddress address = new InetSocketAddress("192.168.254.32", 4000);`
- `MulticastSocket ms3 = new MulticastSocket(address);`

Programmation par Sockets Multicast

Communiquer avec un Groupe Multicast

Une fois créée, une MulticastSocket peut effectuer les opérations suivantes:

1. Rejoindre un groupe multicast.
2. Envoyer des données aux membres du groupe.
3. Recevoir des données du groupe.
4. Quitter le groupe multicast.

Programmation par Sockets Multicast

Communiquer avec un Groupe Multicast

- La clase MulticastSocket a des méthodes pour les opérations 1 et 4.
- Pour l'envoi et la réception des données les méthodes de la superclasse DatagramSocket suffisent.
- On peut effectuer ces opérations dans n'importe quel ordre.
- Avant de pouvoir recevoir des données on doit d'abord rejoindre un groupe.
- On n'a pas besoin de rejoinder un groupe pour lui envoyer des données.
- On peut librement mixer des réceptions et des envois de données.

Programmation par Sockets Multicast

Rejoindre un Groupe Multicast

- Pour rejoindre un groupe multicast, passer un InetAddress ou une SocketAddress du groupe multicast à la methods joinGroup():

- **public void** joinGroup(InetAddress address) **throws** IOException
- **public void** joinGroup(SocketAddress address, NetworkInterface interface) **throws** IOException

- Une fois qu'on a rejoint un groupe multicast, on reçoit des datagrammes exactement comme on reçoit des datagrammes unicast.

Programmation par Sockets Multicast

Rejoindre un Groupe Multicast

- Une seule MulticastSocket peut rejoinder plusieurs groups multicast.
- L'information sur l'appartenance à des groups multicast est sauvegardée dans les routeurs multicast, non dans l'objet.
- Dans ce cas, on utilisera l'adresse sauvegardée dans le datagramme arrivant pour déterminer à quelle adresse un paquet était destiné.

Programmation par Sockets Multicast

Rejoindre un Groupe Multicast

- Des sockets multicast multiples sur la même machine et même dans la même application Java peuvent rejoinfre le même groupe.
- Si c'est le cas, chaque socket reçoit une copie complète des données adressées à ce groupe et qui arrivent à la machine hôte locale.
- Un second argument permet de rejoinder un groupe multicast sur une interface de réseau local spécifiée uniquement.

Programmation par Sockets Multicast

Quitter un Groupe et fermer la connection

- Appeller la méthode leaveGroup() lorsqu'on ne veut plus recevoir des datagrammes du groupe multicast spécifé, sur tous les interfaces réseaux ou bien une interface spécifiée:

- **public void** leaveGroup(InetAddress address) **throws** IOException
- **public void** leaveGroup(SocketAddress multicastAddress,
- NetworkInterface interface) **throws** IOException

- Elle signale au routeur multicast local, disant d'arrêter de nous envoyer des datagrammes.

Programmation par Sockets Multicast

Envoi de données multicast

- L'envoi de données avec MulticastSocket est similaire à l'envoi de données avec DatagramSocket.
- Ranger les données dans un objet DatagramPacket et envoyer en utilisant la méthode send() héritée de DatagramSocket.
- Les données sont envoyées à chaque member du groupe multicast auquel le paquet est adressé

Programmation par Sockets Multicast

Exemple : Envoi de données multicast

```
try {
InetAddress ia = InetAddress.getByName("experiment.mcast.net");
byte[] data = "Here's some multicast data\r\n".getBytes();
int  port = 4000;
DatagramPacket dp = new  DatagramPacket(data, data.length, ia,
port);
MulticastSocket ms = new  MulticastSocket();
ms.send(dp);
} catch  (IOException ex) {System.err.println(ex);}
```

Programmation par Sockets Multicast

Envoi de données multicast

- Par défaut, les sockets multicast utilisent un TTL égal à 1 (la paquet ne sort pas du sous-réseau local).
- La méthode setTimeToLive() fixe la valeur par défaut de TTL pour les paquets envoyés par la méthode send(DatagramPacket dp) héritée de DatagramSocket.
- Par opposition, dans MulticastSocket on utilise la méthode send(DatagramPacket dp, byte ttl).
- La méthode getTimeToLive() retourne la valeur par défaut de TTL de MulticastSocket:

Programmation par Sockets Multicast

Exemple :Envoi de données multicast

```
try {
InetAddress ia = InetAddress.getByName("experiment.mcast.net");
byte[] data = "Here's some multicast data\r\n".getBytes();
int port = 4000;
DatagramPacket dp = new DatagramPacket(data, data.length, ia, port);
MulticastSocket ms = new MulticastSocket();
ms.setTimeToLive(64);
ms.send(dp);
} catch (IOException ex) {System.err.println(ex);}
```

Programmation par Sockets Multicast

Interfaces Réseaux

- Sur une machine avec plusieurs interfaces réseaux, les méthodes setInterface() et setNetworkInterface() choisissent l'interface réseau utilisé pour l'envoi et la réception multicast:

- public void setInterface(InetAddress address) throws SocketException
- public InetAddress getInterface() throws SocketException
- public void setNetworkInterface(NetworkInterface interface) throws SocketException
- public NetworkInterface getNetworkInterface() throws SocketException

Programmation par Sockets Multicast

Interfaces Réseaux

- La méthode setNetworkInterface() est utilisée pour la même raison que la méthode setInterface(), c.a.d choisir l'interface réseau utilisé pour l'envoi et la réception multicast.
- Cependant, setNetworkInterface() choisit l'interface sur la base du nom de l'interface tel que "eth0" (comme encapsulé dans l'objet NetworkInterface) plutôt que sur la base de l'adresse IP (encapsulée dans l'objet InetAddress).
- La méthode getNetworkInterface() retourne un objet NetworkInterface representant l'interface réseau sur lequel MulticastSocket se met à l'écoute de données.

Programmation par Sockets Multicast

Exemple 1 : *Multicast sniffer*

- La classe MulticastSniffer lit le nom d'un groupe multicast à partir de la ligne de commande, construit une InetAddress à partir du ce nom, et crée une MulticastSocket, qui tente de rejoinder le groupe multicast à ce nom de machine. Si la tentative réussit, MulticastSniffer reçoit les datagrammes à partir de la socket et imprime leurs contenus sur System.out.

Programmation par Sockets Multicast

Exemple 1 : *Multicast sniffer*

```
import java.io.*;
import java.net.*;
public class MulticastSniffer {
public static void main(String[] args) {
InetAddress group = null;
int port = 0;
// read the address from the command line
```

Programmation par Sockets Multicast

Exemple 1 : *Multicast sniffer*

```
try {
group = InetAddress.getByName(args[0]);
port = Integer.parseInt(args[1]);
} catch (ArrayIndexOutOfBoundsException | NumberFormatException
| UnknownHostException ex) {System.err.println( "Usage: java MulticastSniffer multicast_address port");
System.exit(1);}
MulticastSocket ms = null;
try { ms = new MulticastSocket(port);
ms.joinGroup(group);
byte[] buffer = new byte[8192];
while (true) {
 DatagramPacket dp = new DatagramPacket(buffer, buffer.length);
ms.receive(dp);
```

Programmation par Sockets Multicast

Exemple 1 : *Multicast sniffer*

```
String s = new String(dp.getData(), "8859_1");
System.out.println(s);
}
} catch (IOException ex) {
System.err.println(ex);
} finally {
if (ms != null) {
try {
ms.leaveGroup(group);
ms.close();
} catch (IOException ex) {}
}}}}
```

Programmation par Sockets Multicast

Exemple 2 : *MulticastSender*

- La classe MulticastSender lit les entrées a partir de la ligne de commande et les envoie à un groupe multicast.

```
import java.io.*;
import java.net.*;
public class MulticastSender {
public static void main(String[] args) {
InetAddress ia = null;
int port = 0;
byte ttl = (byte) 1;
// read the address from the command line
```

Programmation par Sockets Multicast

Exemple 2 : *MulticastSender*

```
try {
ia = InetAddress.getByName(args[0]);
port = Integer.parseInt(args[1]);
if (args.length > 2) ttl = (byte) Integer.parseInt(args[2]);
} catch (NumberFormatException | IndexOutOfBoundsException
| UnknownHostException ex) {
System.err.println(ex);
System.err.println(
"Usage: java MulticastSender multicast_address port ttl");
System.exit(1);
}
```

Programmation par Sockets Multicast

Exemple 2 : *MulticastSender*

```
byte[] data = "Here's some multicast data\r\n".getBytes();
DatagramPacket dp = new DatagramPacket(data, data.length, ia, port);
try (MulticastSocket ms = new MulticastSocket()) {
ms.setTimeToLive(ttl);
ms.joinGroup(ia);
for (int i = 1; i < 10; i++) {
ms.send(dp);
}
ms.leaveGroup(ia);
} catch (SocketException ex) {
System.err.println(ex);
} catch (IOException ex) {
System.err.println(ex); }}}
```