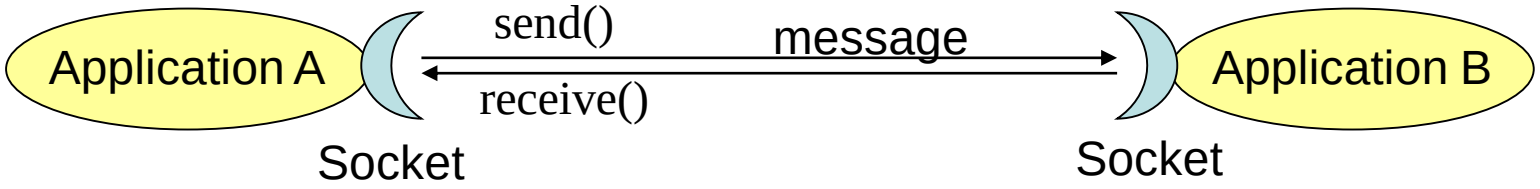
TP 3: **Communication par message, Socket**

a. Introduction sur les sockets
b. Sockets en mode flux (datastream)
c. Sockets en mode datagram
d. Sockets en mode multicast

a. Introduction sur les sockets

Socket = interface de programmation (API) avec les services du système d'exploitation pour exploiter les services de communication du système (local ou réseau)

Une socket est un demi-point de connexion d'une application
(≠ communication par message/MOM)



- Une socket est caractérisée par une adresse
- Plusieurs domaines de sockets existent :
  - Socket Unix (local) = un chemin dans le système de fichier
  - Socket Inet (réseau TCP, UDP ou IP) = adresse IP + port

a. Introduction sur les sockets

Couche 7	Applicative	Logiciels	NFS
Couche 6	Présentation	Représentation indépendante des données	XDR
Couche 5	Session	Établit et maintient des sessions	RPC
Couche 4	Transport	Liaison entre applications de bout en bout, fragmentation, éventuellement vérification	TCP, UDP, Multicast
Couche 3	Réseau	Adressage et routage entre machines	IP
Couche 2	Liaison	Encodage pour l'envoi, détection d'erreurs, synchronisation	Ethernet
Couche 1	Physique	Le support de transmission lui-même	

- Sockets en Java : uniquement orientée transport (couche 4)
- Deux API pour les sockets
  - java.net : API bloquante (étudié ici)
  - java.nio.channels (> 1.4) : API non bloquante (non étudiée dans ce cours)

a. Introduction sur les sockets

Mode connecté : la communication entre un client et un serveur est précédée d'une connexion et suivi d'une fermeture

- Facilite la gestion d'état
- Meilleurs contrôle des arrivées/départs de clients
- Uniquement communication unicast
- Plus lent au démarrage

Mode non connecté : les messages sont envoyés librement

- Plus facile à mettre en œuvre
- Plus rapide au démarrage

💣 Il ne faut pas confondre connexion au niveau transport et au niveau applicatif!

- HTTP est un protocole non connecté alors que TCP l'ai
- FTP est un protocole connecté et TCP aussi
- RPC est un protocole non connecté et UDP non plus

a. Introduction sur les sockets

Liaison par **flux** : Socket/ServerSocket (TCP)

- Connecté : protocole de prise de connexion (lent)
  ⇒ communication uniquement point à point
- Sans perte : un message arrive au moins une fois
- Sans duplication : un message arrive au plus une fois
- Avec fragmentation : les messages sont coupés
- Ordre respecté
- ✓ Communication de type téléphone

Liaison par **datagram** : DatagramSocket/DatagramPacket (UDP)

- Non connecté : pas de protocole de connexion (plus rapide)
- Avec perte : l'émetteur n'est pas assuré de la délivrance
- Avec duplication : un message peut arriver plus d'une fois
- Sans fragmentation : les messages envoyés ne sont jamais coupés
  ⇒ soit un message arrive entièrement, soit il n'arrive pas
- Ordre non respecté
- ✓ Communication de type courrier

a. Introduction sur les sockets

Une socket est identifiée par

- Une adresse IP : une des adresses de la machine (ou toutes)
- Un port : attribué automatiquement ou choisi par le programme

Adresse de Socket = Adresse IP + port

Une socket communique avec une autre socket via son adresse

- Flux : une socket se connecte à une autre socket via son adresse de socket
- Datagram : une socket envoie/reçoit des données à/d'une autre socket identifiée par son adresse de socket

a. Introduction sur les sockets

Adressage

- Une adresse IP : identifie une carte réseau d'une machine
  (par exp : 195.83.118.1)
  ⇒ une machine peut posséder plusieurs adresses (penser aux routeurs)
- Un port : identifie l'application (par exp 21/ftpd)
  - Seul l'administrateur peut ouvrir des ports < 1024

Nom symbolique (Domain Name Server)

- Associe une adresse IP à un nom symbolique
  (par exp ftp.lip6.fr => 195.83.118.1)
- Une adresse peut être associée à plusieurs noms
  (par exp nephtys.lip6.fr ⇒ ftp.lip6.fr ⇒ 195.83.118.1)

Classes d'adresses : A (1-126), B (128-191), C (192-223),
D/Multicast (224-239), Locale (127)

a. Introduction sur les sockets

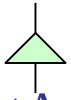
java.net.InetAddress : objet représentant une adresse IP

static InetAddress InetAddress.getByAddress(byte ip[])	construit un objet d'adresse ip
static InetAddress InetAddress.getByName(String name)	renvoie l'Adresse IP de name
static InetAddress InetAddress.getLocalHost()	renvoie notre adresse
String InetAddress.getHostName()	renvoie le nom symbolique
byte[] InetAddress.getHostAddr()	renvoie l'adresse IP

```
public class Main {
  public static void main(String args[]) {
    byte ip[] = {195, 83, 118, 1 };
    InetAddress addr0 = InetAddress.getByAddress(ip);
    InetAddress addr1 = InetAddress.getByName("ftp.lip6.fr");
    …
  }
}
```

a. Introduction sur les sockets

java.net.SocketAddress : objet représentant une adresse de Socket sans protocole attaché



java.net.InetSocketAddress : objet représentant une adresse IP + port

InetSocketAddress(InetAddress addr, int port);	Construit une adresse de Socket
InetSocketAddress(String name, int port);	Construit une adresse de Socket
InetAddress InetSocketAddress.getAddress()	Renvoie l'adresse IP
int InetSocketAddress.getPort();	Renvoie le port
String InetSocketAddress.getHostName();	Renvoie le nom symbolique de l'IP

```
public class Main {
  public static void main(String args[]) {
    byte ip[] = {195, 83, 118, 1 };
    InetAddress addr = InetAddress.getByAddress(ip);
    InetSocketAddress saddr0 = new InetSocketAddress(addr, 21);
    InetSocketAddress saddr1 = new InetSocketAddress("ftp.lip6.fr", 21);
    … } }
```

a. Introduction sur les sockets

Problème : plusieurs clients **colocalisés** doivent se connecter à un serveur unique

- Temps d'ouverture/fermeture de connexion long
- Tous les clients ne sont pas forcement connectés à chaque instant
- Apparition/disparition de clients

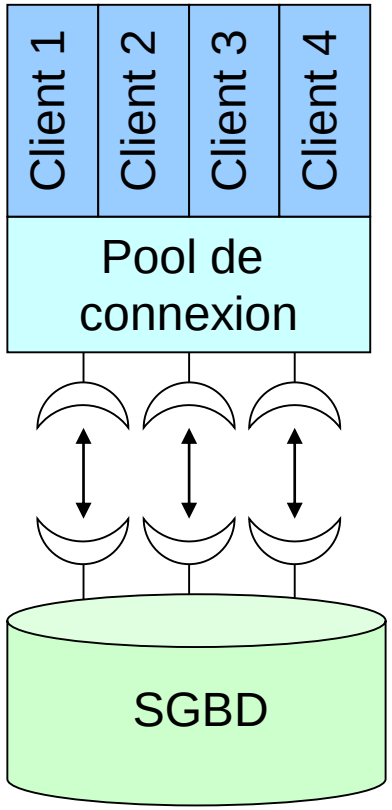
Solution : mutualiser les connexions

- Pool de connexions ouvertes en permanence
- Les clients (ré)utilisent les connexions ouvertes

Exemple : pool de connexions à un SGBD

⇒ Politique de gestion de ressources partagées

Problèmes classiques de réservation de ressources, d'interblocages…



a. Introduction sur les sockets

Problème :

- Passage de firewalls
- Optimisation du nombre de connexions

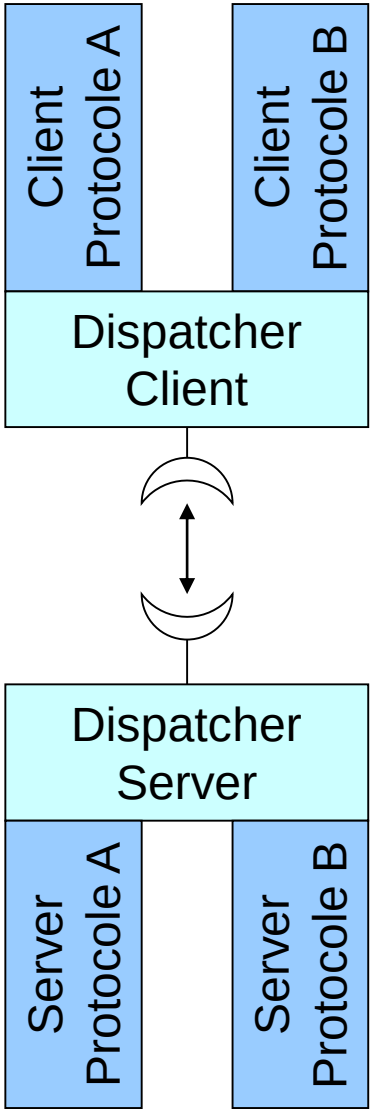
Solution : Multiplexer une connexion

- Plusieurs protocoles transitent par la même socket

⇒ Distinguer les flux de données

- Encadrer les protocoles par des méta-données
- Acheminer le message vers la bonne application

Peut être couplé avec un pool de connexions



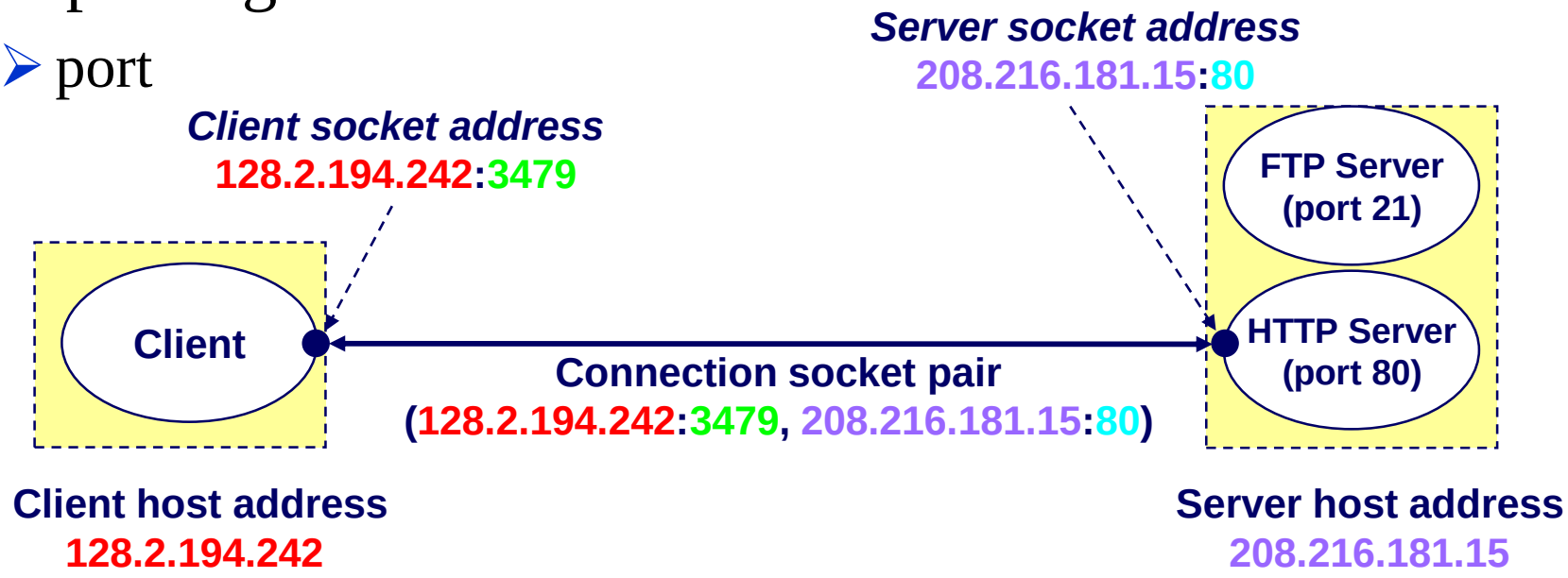
Identifier la Destination

Addressing

- IP address
- hostname (resolve to IP address via DNS)

Multiplexing

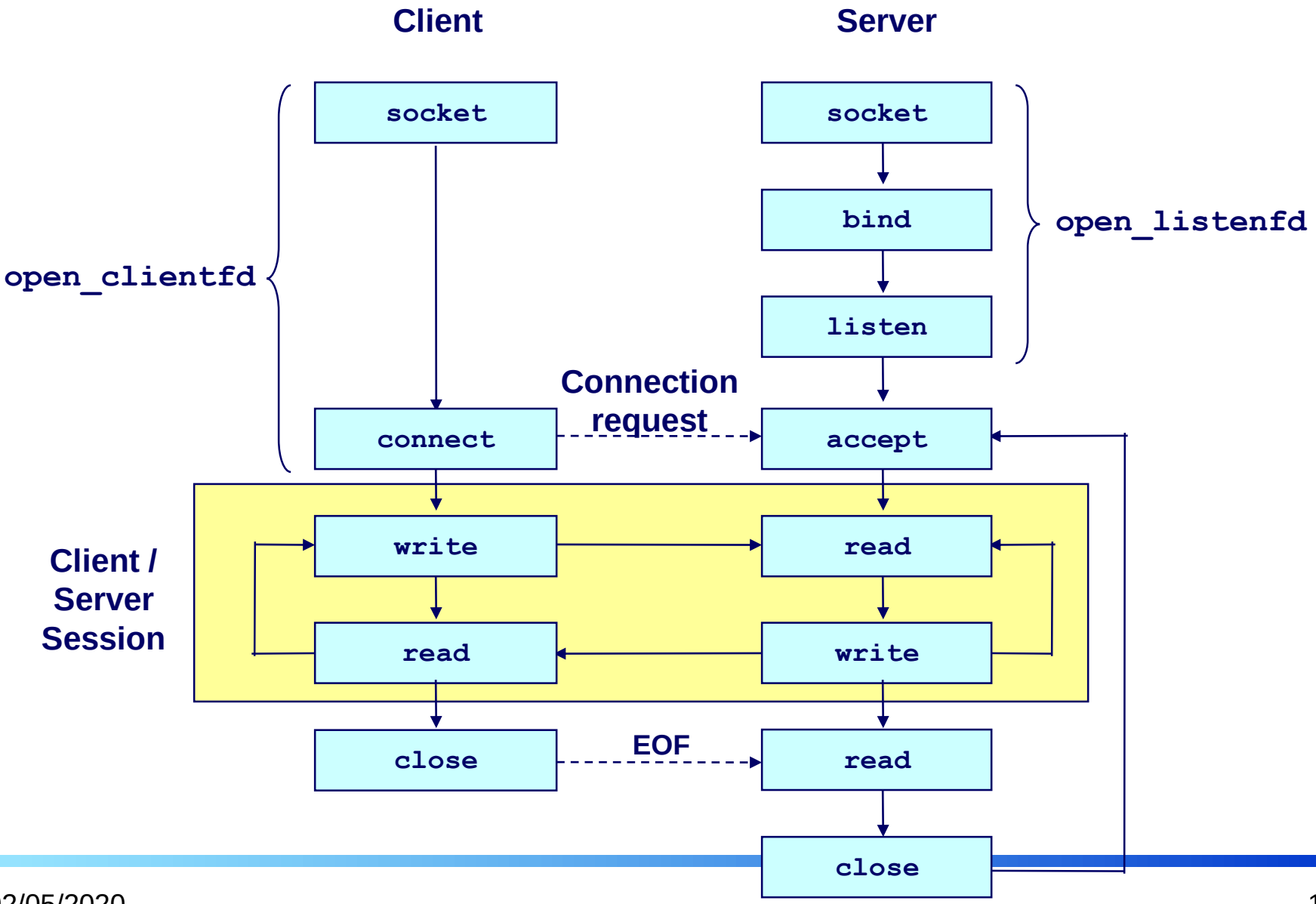
- port



How to use sockets

- Setup socket
  - ✓ Where is the remote machine (IP address, hostname)
  - ✓ What service gets the data (port)
- Send and Receive
  - ✓ Designed just like any other I/O in unix
  - ✓ send -- write
  - ✓ recv – read
- Close the socket

a. Introduction sur les sockets



b. Sockets en mode flux

Socket en mode flux de Java

- Repose sur TCP

Propriétés

- Taille des messages quelconques
- Envoi en général bufferisé (i.e. à un envoi OS correspond plusieurs écritures Java)
- Pas de perte de messages, pas de duplication
- Les messages arrivent dans l'ordre d'émission
- Contrôle de flux (i.e. bloque l'émetteur si le récepteur est trop lent)
- Pas de reprise sur panne
  Trop de perte ou réseau saturé ⇒ connexion perdue

Nombreuses utilisations : HTTP, FTP, Telnet, SMTP, POP…

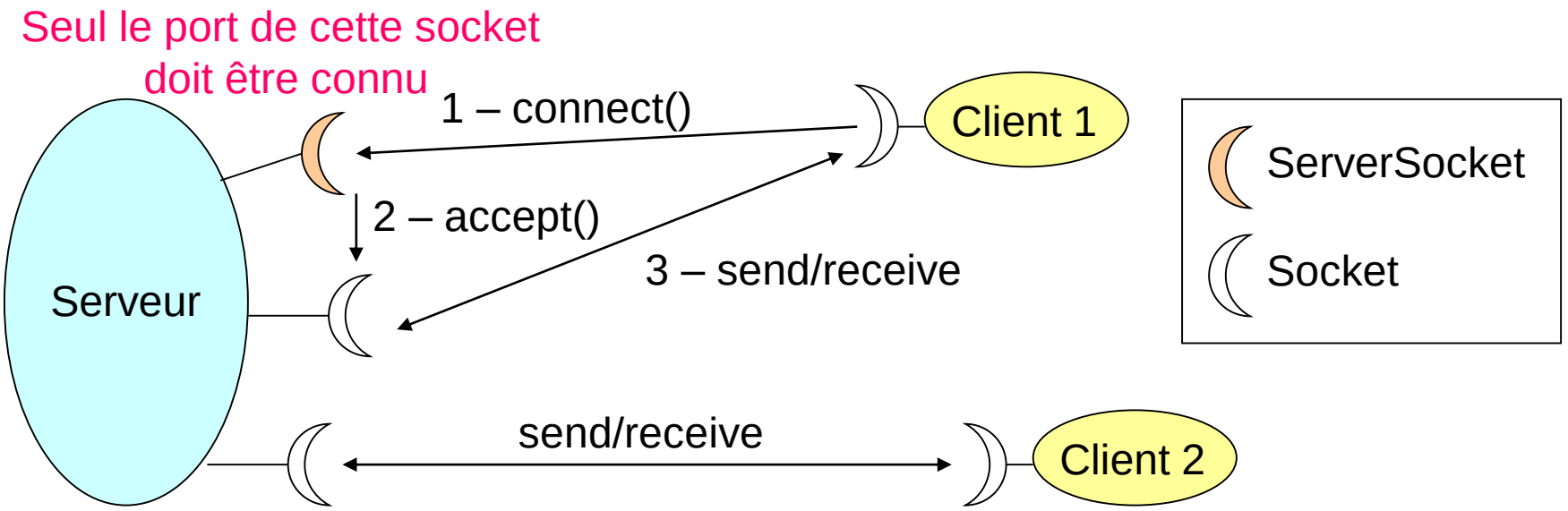
b. Sockets en mode flux

Nécessite une phase de connexion

- Serveur : attend des connexion ⇒ une socket de connexion (ServerSocket)
- Client : se connecte au serveur (Socket)

Le serveur doit maintenir des connexions avec plusieurs clients

⇒ Une socket de communication par client (Socket)



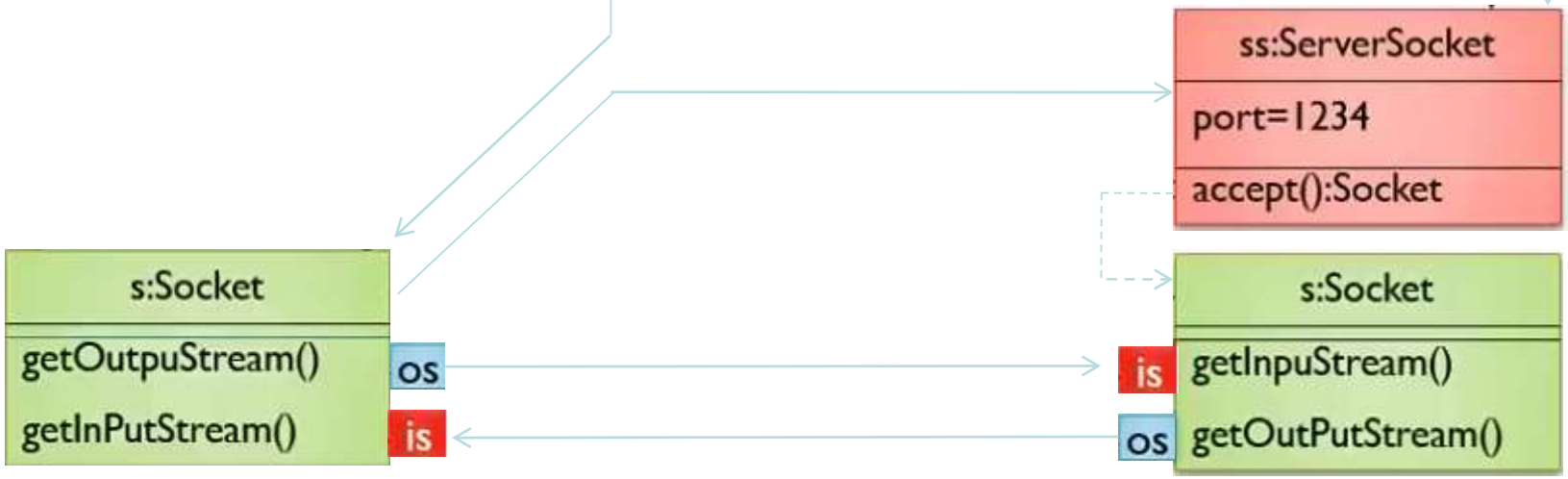
b. Sockets en mode flux

Client

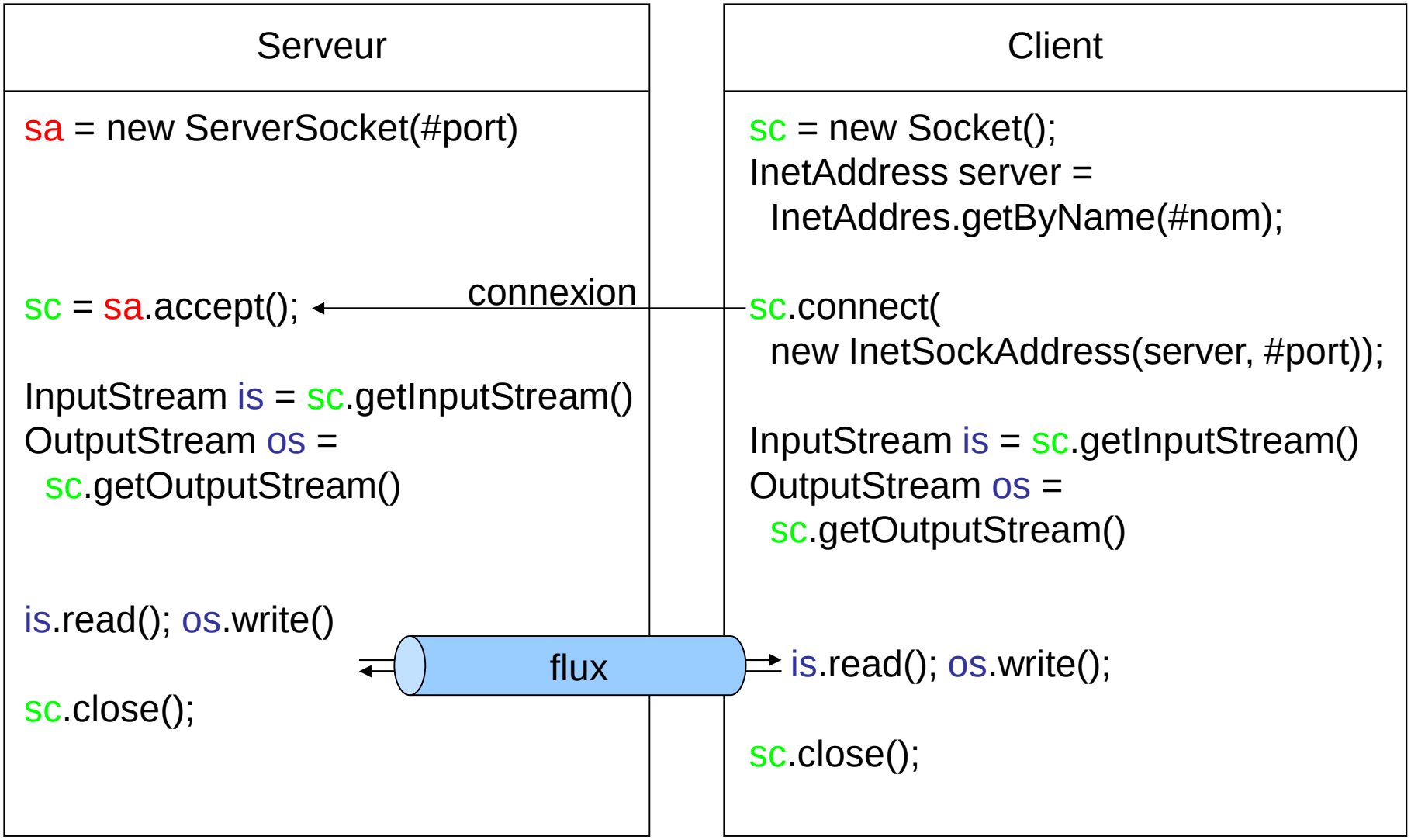
```
Socket s=new Socket("192.168.1.23",1234)

InputStream is=s.getInputStream();
OutputStream os=s.getOutputStream();
os.write(23);
int rep=is.read();
System.out.println(rep);
```

Serveur

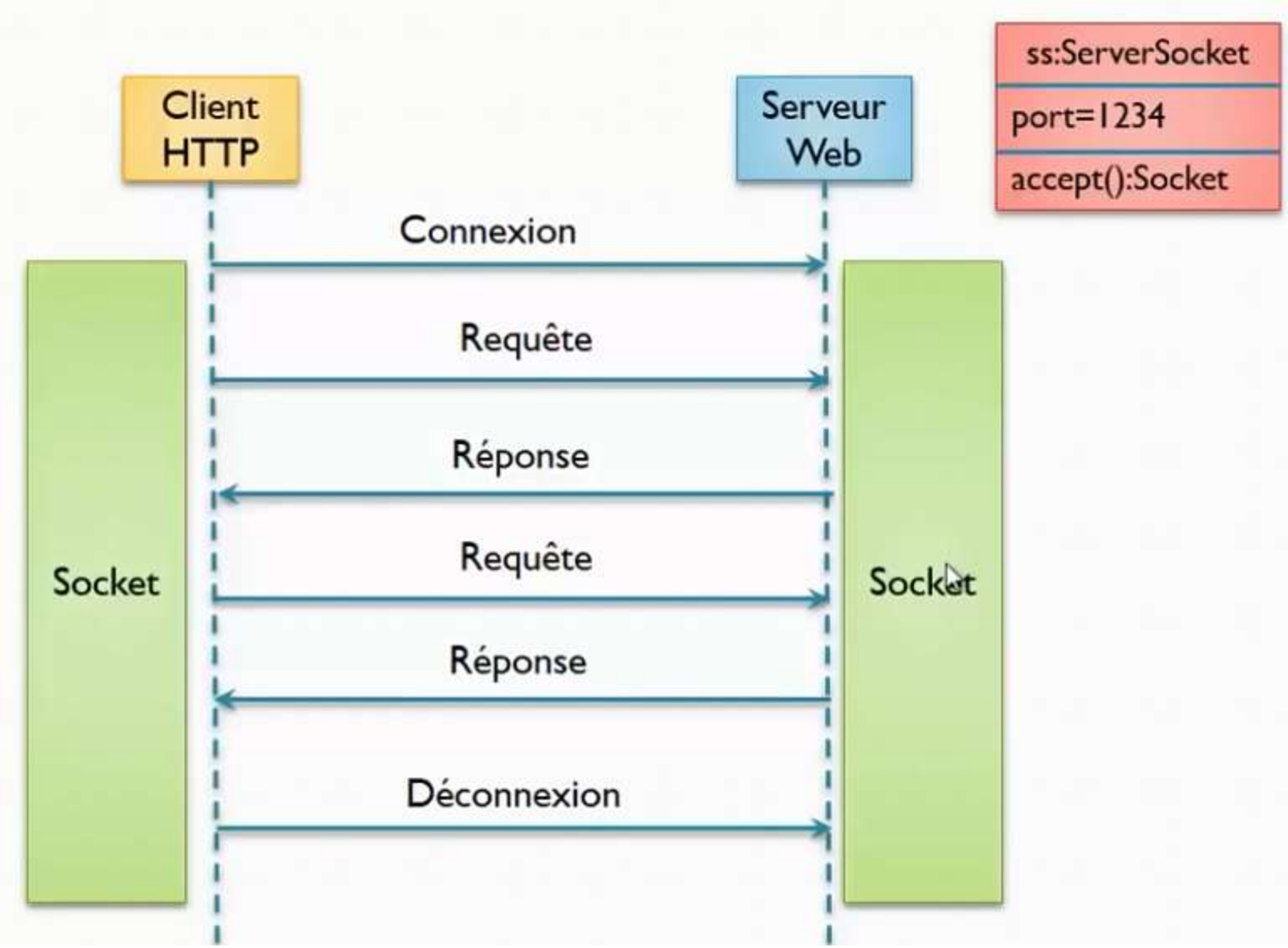
```
ServerSocket ss=new ServerSocket(1234);

InputStream is=s.getInputStream();
OutputStream os=s.getOutputStream();
int nb=is.read();
int rep=nb*2;
os.write(rep);
```



b. Sockets en mode flux



b. Sockets en mode flux



b. Sockets en mode flux

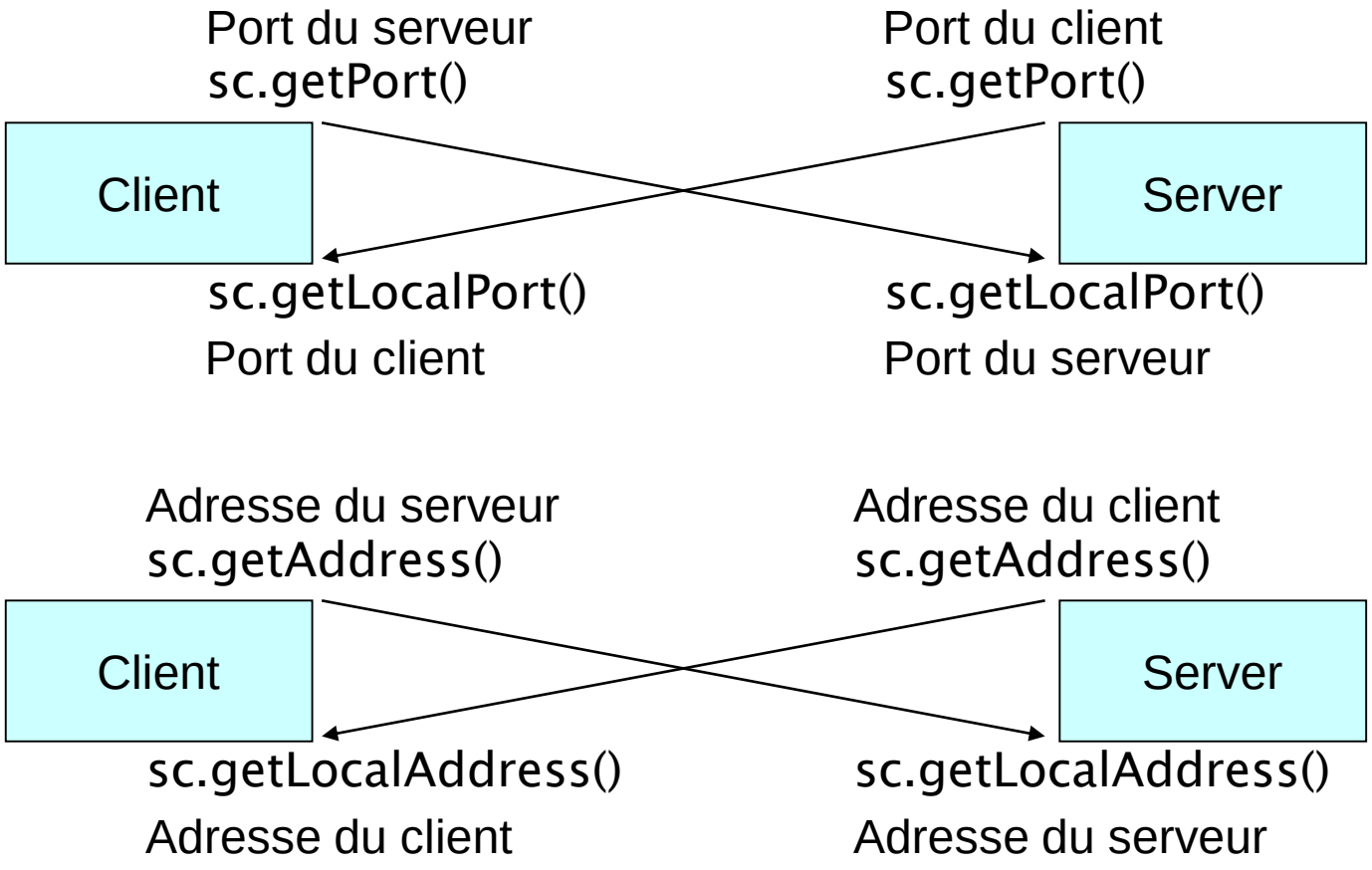
Remarques

- La socket du serveur possède le port #port qui identifie le serveur
- La socket de communication du serveur possède un port attribuée automatiquement par Java lors de l'accept()
- La socket de communication du client possède un port attribuée automatiquement par Java lors du connect()

- Il est possible de fixer le ports de la socket du client par
  sc.bind(new SocketAddress(InetAddress.getLocalHost(), #port));
- Il est possible de fixer le port de la socket de connexion du serveur après sa création par
  ServerSocket sa = new ServerSocket();
  sa.bind(**new** SocketAddress(InetAddress.getLocalHost(), #port));

b. Sockets en mode flux

Retrouver les adresses IP et les ports



b. Sockets en mode flux

Remarque : les flux associés aux sockets peuvent être encapsulés dans n'importe quels autres flux

```
ObjectInputStream is =
    new ObjectInputStream(
      new GZipInputStream(sc.getInputStream()));

ObjectOutputStream os =
    new ObjectOutputStream(
      new GZipOutputStream(sc.getOuputStream()));
```

Émission : os.writeObject("Hello, World!!!");

Réception : System.out.println(is.readObject());

c. Sockets en mode datagram

Socket en mode datagram de Java

- Repose sur UDP

Propriétés

- Taille des messages fixe et limitée (64ko)
- Envoi non bufférisé
- Possibilité de perte de messages, Duplication
- Les messages n'arrivent pas forcement dans l'ordre d'émission
- Aucun contrôle de flux
- Pas de détection de panne (même pas assuré que les messages arrivent)
- **Faible latence** (car aucun contrôle de flux, pas de connexion)

Nombreuses utilisations :

- DNS, TFTP, …

c. Sockets en mode datagram

DatagramSocket : Socket orientée Datagram

- Liée à un port
  - ✓ Assignation explicite **new DatagramSocket(#port)** ou **socket.bind(SocketAddress saddr);**
  - ✓ Assignation automatique lors de la première entrée/sortie
- Communique uniquement via des **DatagramPacket** (pas de flux!)

DatagramPacket : représente un message

- En réception : **DatagramPacket(byte buf[], int offset, int length);**
- En émission : **DatagramPacket(byte buf[], int offset, int length, InetSocketAddress saddr);**

💣 Attention : si la taille du buffer de réception est trop petite, la fin du message est perdu!

c. Sockets en mode datagram

Serveur

```
sc = new DatagramSocket(#port);

byte msg[] = new byte[256];
DatagramPacket in =
  new DatagramPacket(msg, 256);

sc.receive(in);
System.out.println(new String(msg));

msg = "Bien recu".getBytes();

sc.send(
  new DatagramPacket(msg, 0,
    msg.length,
    in.getSocketAddress()));
```

Client

```
sc = new DatagramSocket();
InetAddress server =
  InetAddres.getByName(#nom);

byte msg[] =
  "Hello, World!!!".getBytes();
sc.send(
  new DatagramPacket(msg, 0
    msg.length, server, #port);

msg = new byte[256];
DatagramPacket in =
  new DatagramPacket(msg, 256);

sc.receive(in);
System.out.println(new String(msg));
```

c. Sockets en mode datagram

Remarques sur les ports

- Le client a besoin de connaître l'adresse IP et le port du serveur
- La socket du client se voit assigner un port lors de l'envoi

Remarques sur la sérialisation

- Une DatagramSocket ne possède pas de flux (car ce n'est pas un flux!!!)
- Envoi uniquement de tableaux de bytes
- Réception uniquement de tableaux de bytes

La sérialisation est tout de même possible en utilisant

- **ByteArrayInputStream** : flux d'entrée qui lit à partir d'un tableau de bytes
- **ByteArrayOutputStream** : flux de sortie qui écrit dans un tableau de bytes

- Les messages restent limités en taille et le récepteur doit prévoir à priori un tampon suffisamment grand

c. Sockets en mode datagram

```
DatagramPacket serialize(Object o) {
  ByteArrayOutputStream bos = new ByteArrayOutputStream();
  ObjectOutputStream os = new ObjectOutputStream(bos);
  os.writeObject(o);
  byte msg[] = bos.toByteArray();
  return new DatagramPacket(msg, msg.length);
}

Object deserialize(DatagramPacket packet) {
  ObjectInputStream is =
    new ObjectInputStream(
      new ByteArrayInputStream(packet.getData(),
                               packet.getOffset(),
                               packet.getLength());
  return is.readObject();
}
```

```
Object receive(DatagramSocket s) {
  byte buf[] = new byte[????];
  DatagramPacket packet =
    new DatagramPacket(buf,
      buf.length);
  s.receive(packet);
  return deserialize(packet);
}
```

Comment prévoir la taille des messages?

c. Sockets en mode datagram

Problème : définir des sockets personnalisées

(Passer des firewalls, utiliser un autre protocole que UDP, crypter les communication…)

Solution : Personnalisation des sockets

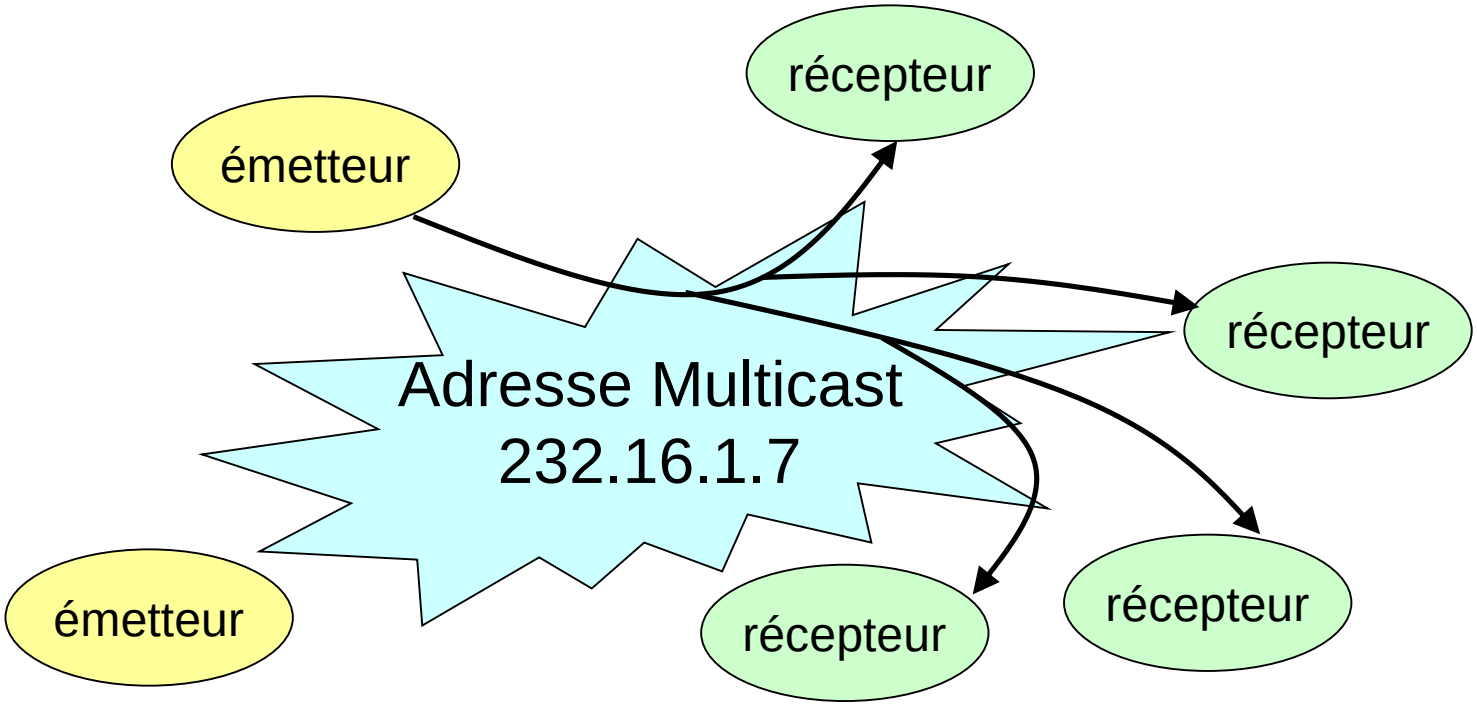
Même principe qu'avec les sockets en mode flux

- Définir une nouvelle implantation de DatagramSocket via DatagramSocketImpl
- Définir une nouvelle usine à DatagramSocket via DatagramSocketImplFactory
- Enregistrer l'usine via DatagramSocket.setDatagramSocketImplFactory(…)

d. Sockets en mode multicast

Multicast = diffusion de groupe

- Récepteur : s'abonne à une adresse IP de classe D
  - Adresse IP comprise entre 224.0.0.0 et 239.255.255.255
  - Certaines adresses sont déjà réservées
    - (voir http://www.iana.org/assignments/multicast-addresses)
- Émetteurs : émettent à destination de cette adresse IP



d. Sockets en mode multicast

Socket en mode multicast de Java

- Repose sur IP Multicast, lui-même basé sur UDP
- Indépendant de UDP

Propriétés : même propriétés qu'UDP

- Taille des messages fixe et limitée (64ko)
- envoi non bufferisé
- Possibilité de perte de messages ou de duplication pour certains récepteurs
- Les messages n'arrivent pas forcement dans l'ordre d'émission, pas forcement dans le même ordre chez tous les récepteurs
- Aucun contrôle de flux

Encore peu d'utilisations :

- Université
- Certaines webradio, certains fournisseurs d'accès pour de la diffusion vidéo
- La plupart des routeurs jettent les packets multicast!

d. Sockets en mode multicast

Groupe multicast = ensemble de récepteurs sur une adresse multicast

Émetteur :

- Émet sur une adresse de classe D + port
- Pas d'abonnement nécessaire
- Émission à tout instant

Récepteur :

- S'abonne à une adresse de classe D (⇒ abonnement de la machine)
  MulticastSocket.joinGroup(InetAddress group);
- Écoute sur un port donnée (⇒ abonnement de l'application)
- Se désabonne de la classe D avant de quitter
  MulticasrSocket.leaveGroup(InetAddress group);
- Peut rejoindre et quitter le groupe multicast à tout instant

d. Sockets en mode multicast