Applications réseaux

accès à UDP

Etienne Duris

Arnaud Carayol

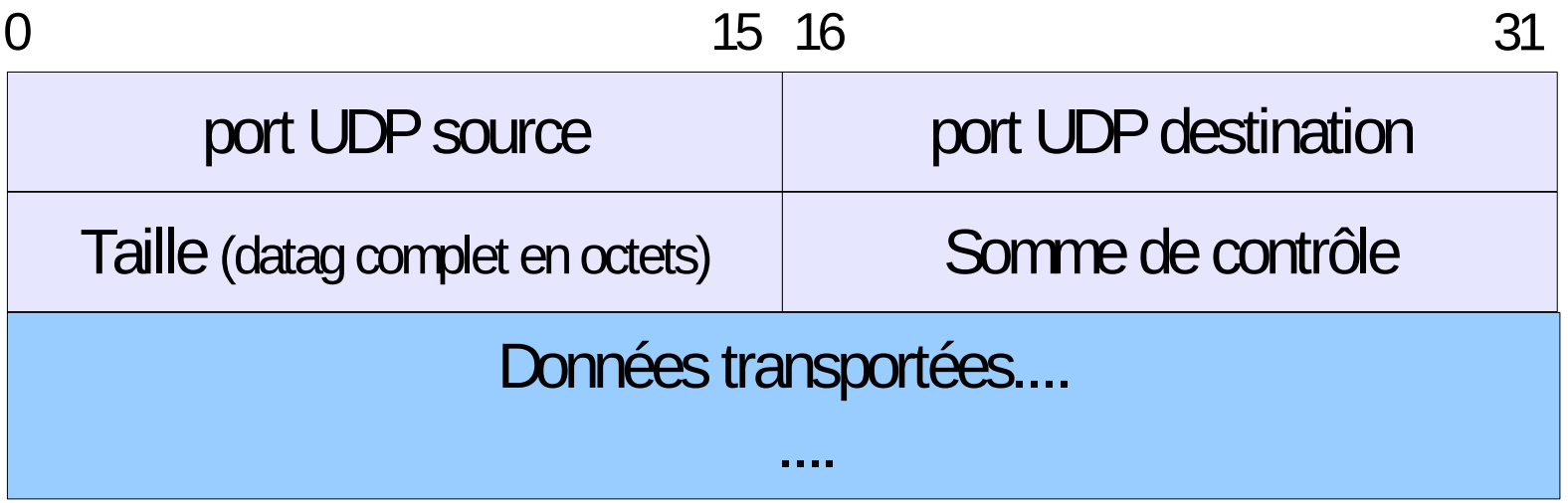
UNIVERSITÉ
PARIS-EST
MARNE-LA-VALLÉE

Le protocole UDP

- User Datagram Protocol (RFC 768)
  - acheminement de datagrammes au dessus de IP
  - pas de fiabilité supplémentaire assurée
  - assure la préservation des limites de chaque datagramme
  - autorise la diffusion
- Le multiplexage/démultiplexage au niveau des machines se fait *via* la notion de port
  - certains ports sont affectés à des services particuliers
    - RFC 1700 ou www.iana.org/assignments/port-numbers
    - En général, les n° de port inférieurs à 1024 sont réservés

Format

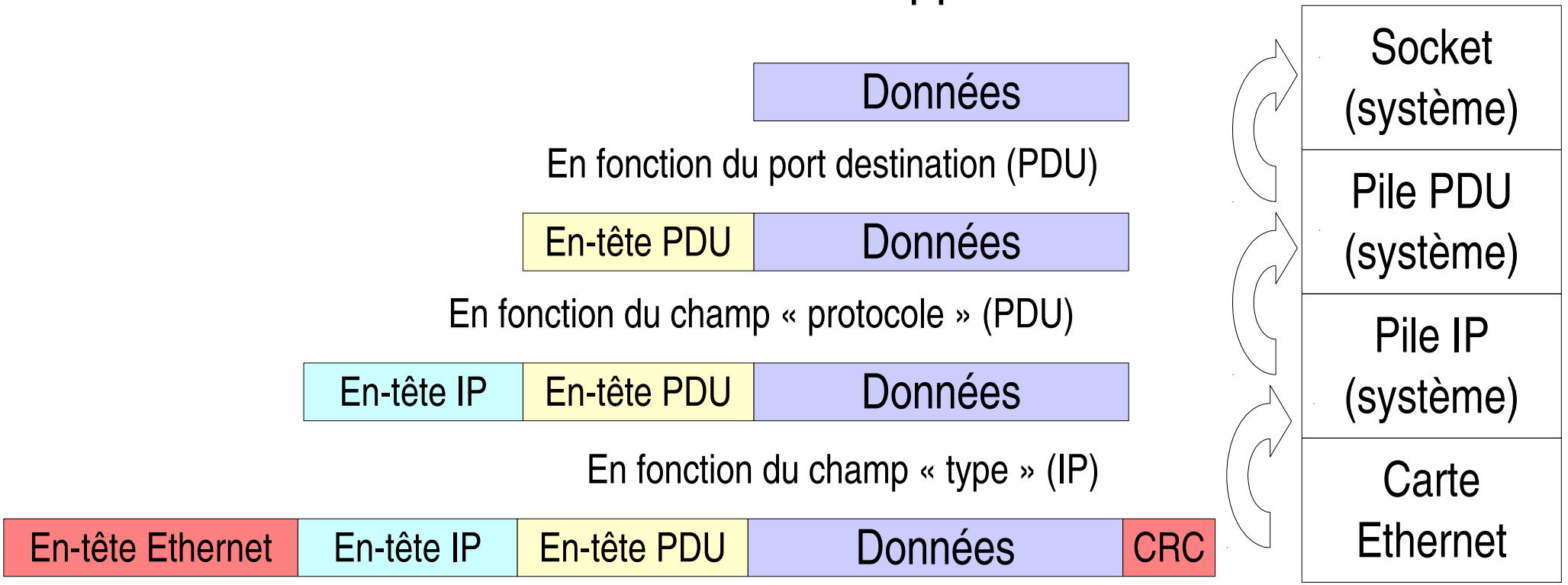
- Taille max des données transportées: (2^{16}-1-8)~64Ko
- Checksum optionnel en IP v4, obligatoire en IP v6
  - ajout (pour le calcul) d'un pseudo-en-tête avec @ dest et src



Exemple d'encapsulation dans une trame Ethernet:

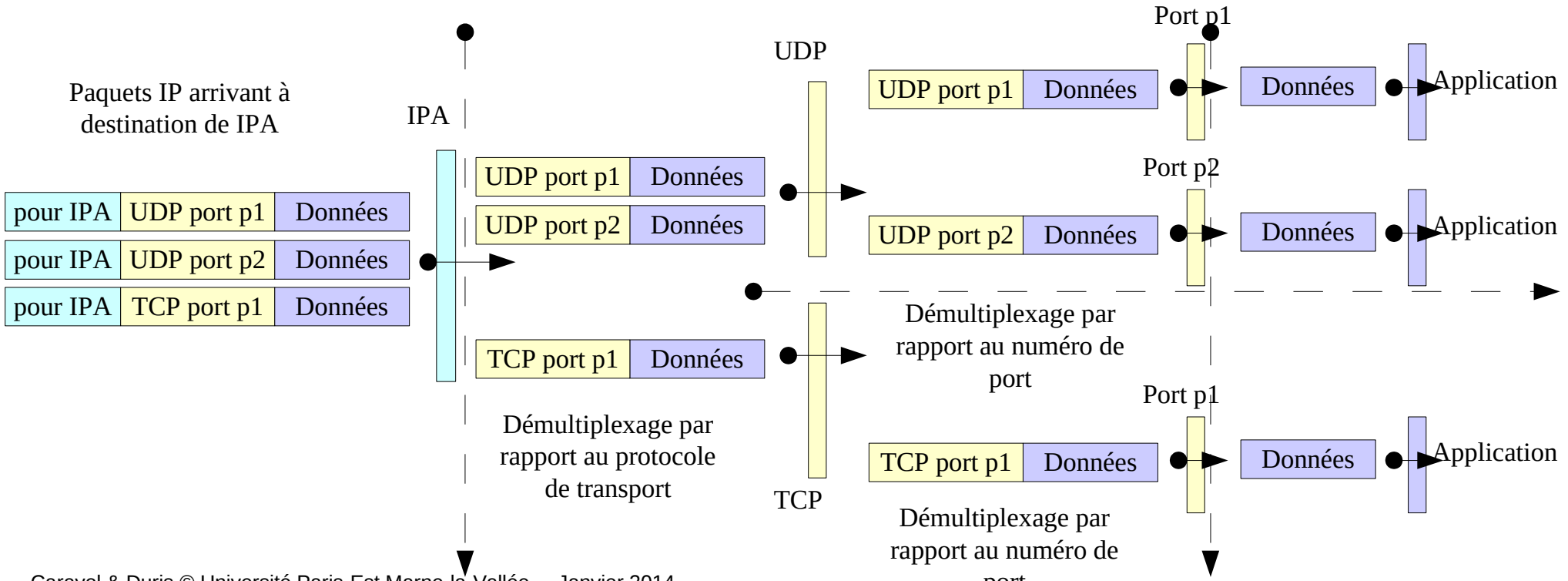
Encapsulation des données

- Une trame Ethernet est adressée à une adresse MAC
- Un paquet IP (couche 3) est destiné à une adresse IP
- Un PDU (couche 4 – UDP ou TCP) est destiné à un port
- Les données sont destinées à une application

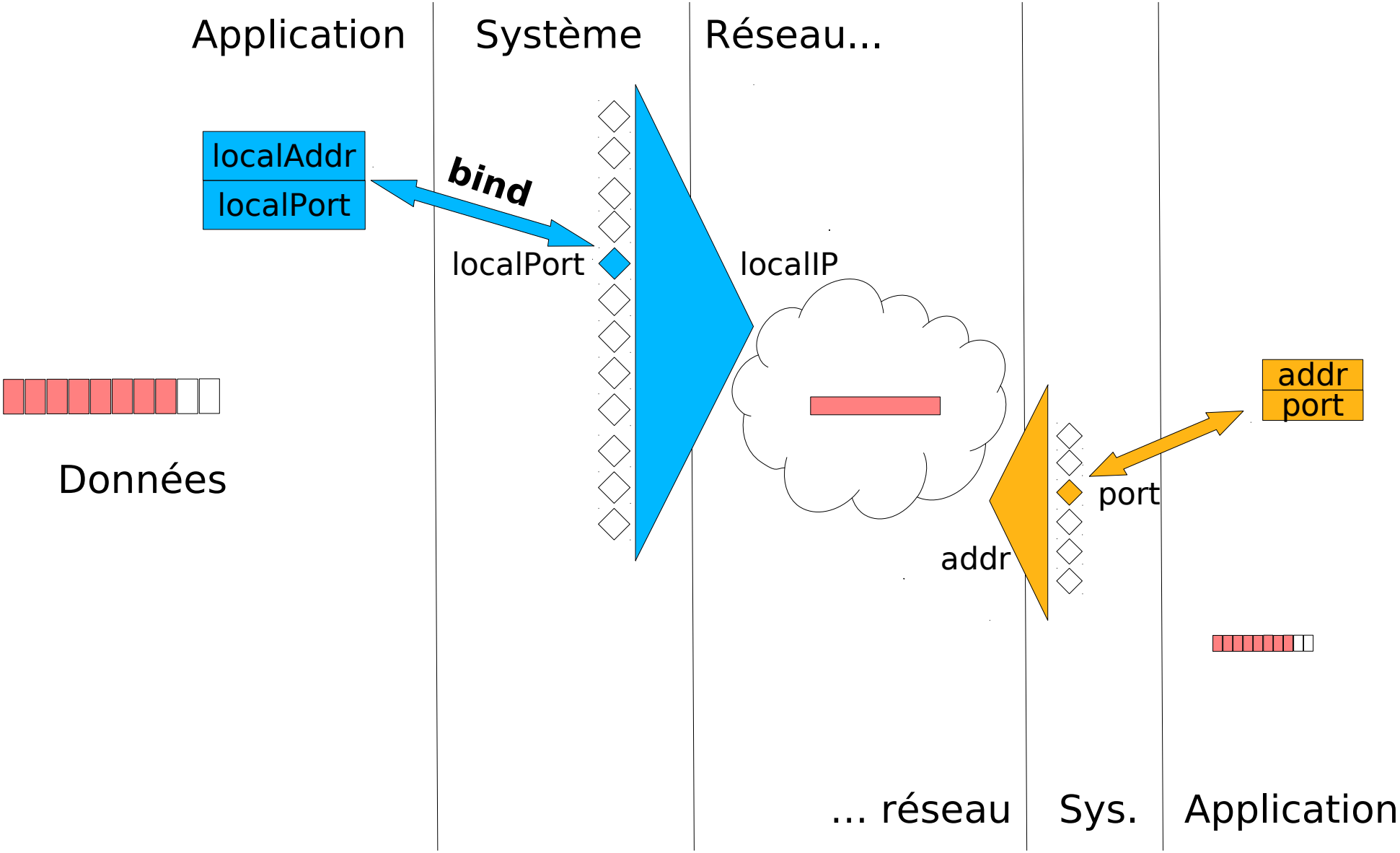


Multiplexage / démultiplexage

- Les applications se lient à des **sockets** pour accéder au réseau
- Ces sockets système sont identifiées par une **adresse IP** et un numéro de port
- Elles permettent de multiplexer/démultiplexer les communications des différentes applications



Les différents niveaux impliqués



Les adresses de socket

- Une socket identifie un point d'attachement à une machine. Selon le protocole, des classes sont dédiées pour représenter ces objets
  - java.net.DatagramSocket pour UDP
  - java.net.Socket pour TCP
- Les adresses de sockets identifient ces points d'attachement indépendamment du protocole
  - java.net.SocketAddress (classe abstraite)
  - *a priori* indépendant du protocole réseau, mais la seule classe concrète est dédiée aux adresses Internet (IP)
    - java.net.InetSocketAddress

Adresses de loopback et non spécifiée

- Loopback: ne correspond à aucun réseau physique
  - **127.0.0.1** en IP v4
  - **::1** en IP v6
  - nom en général associé: localhost
- Adresse non spécifiée
  - **0.0.0.0** en IP v4
  - **::0** en IP v6
  - En émission, elle vaut **l'une des** adresses IP de la machine locale **(anylocal)**
  - En réception, elle filtre **toutes** les adresses IP de la machine locale **(wildcard)**

Classe InetSocketAddress

- java.net.InetSocketAddress
  - adresse IP (InetAddress)
  - numéro de port (int)
- Trois constructeurs acceptant comme arguments
  - InetSocketAddress(InetAddress addr, int port)
    - Si addr est null, la socket est liée à l'adresse wildcard (non spécifiée)
  - InetSocketAddress(int port)
    - l'adresse IP est l'@ wildcard
  - InetSocketAddress(String hostName, int port)
    - Si hostName non résolu, l'adresse de socket est marquée comme étant non résolue
    - Peut être testé grâce à la méthode isUnresolved()

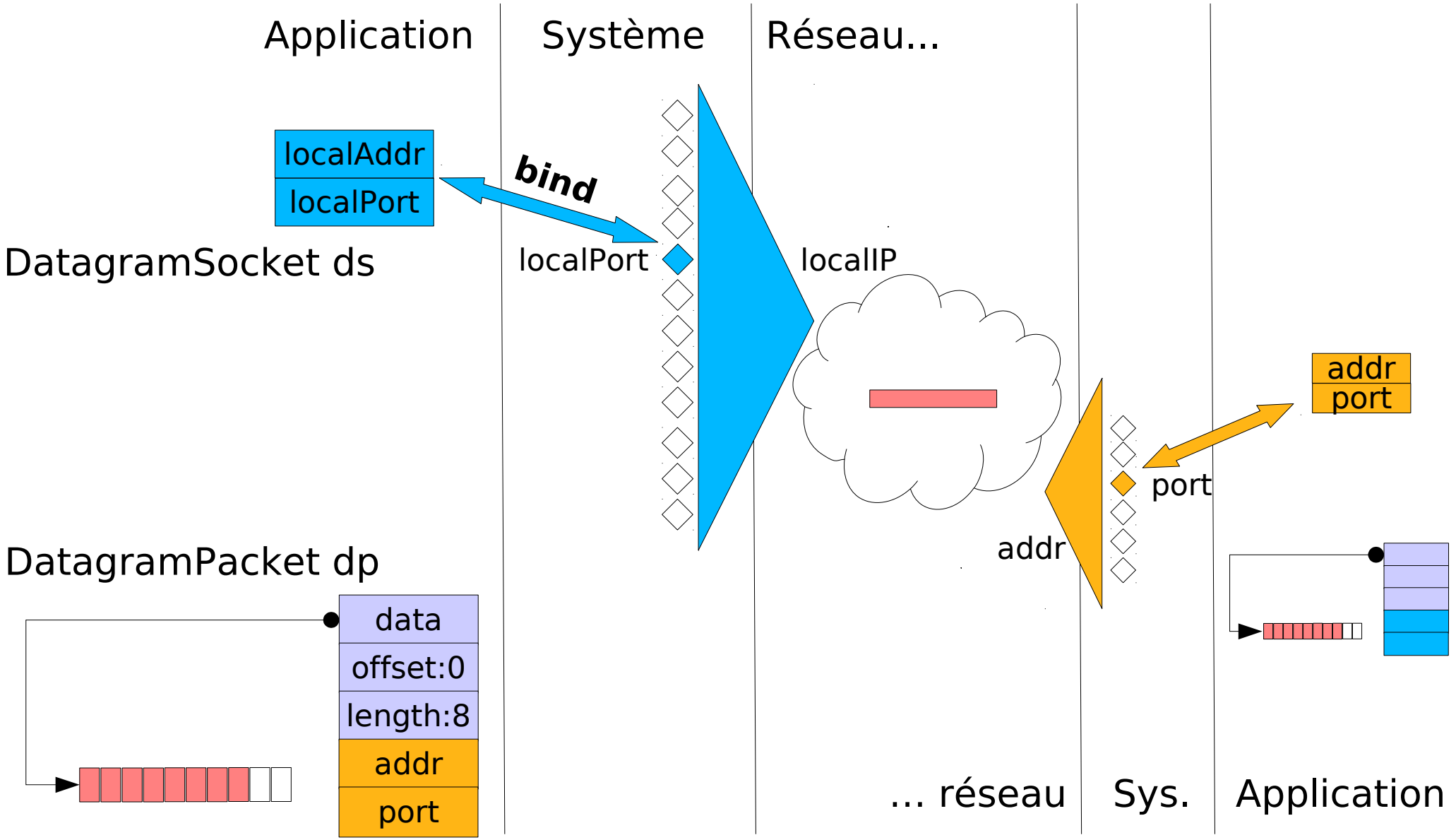
Adresse de socket (suite)

- Tous lèvent une IllegalArgumentException si le numéro de port est en dehors de [0..65535]
- Si le port spécifié vaut 0, un port éphémère sera choisi lors de l'attachement de la socket.
- La méthode toString() affiche **<@ip>:<n°port>**
- ```
  InetSocketAddress sa1 =
        new InetSocketAddress("un.truc.zarb",50);
  System.out.println(sa1+(sa1.isUnresolved()?" non résolue":"résolue"));
        // Affiche : un.truc.zarb:50 non résolue
  InetSocketAddress sa2 =
        new InetSocketAddress("java.sun.com",80);
  System.out.println(sa2 + (sa2.isUnresolved()?" non résolue":" résolue"));
        // Affiche : java.sun.com/192.18.97.71:80 résolue
  ```
  ```

Accès Java à UDP

- Classiquement, l'accès se fait grâce à deux classes et permet de manipuler des tableaux d'octets. Il faut les « **interpréter** » en fonction de l'application: encodage de caractères, format, etc.
- java.net.DatagramSocket
  - représente une socket d'attachement à un port UDP
    - Il en faut une sur chaque machine pour communiquer
- java.net.DatagramPacket qui représente deux choses:
  - Les données qui transitent:
    - un tableau d'octets,
    - l'indice de début et
    - le nombre d'octets
  - La machine distante:
    - son adresse IP
    - son port

Les différents niveaux impliqués



Usage classique d'un DatagramSocket

- L'instance représente un objet permettant d'envoyer ou recevoir des datagrammes UDP
- Côté client
  - `DatagramSocket ds = new DatagramSocket() ;`
  - Crée une instance attachée à l'adresse wildcard et à un port UDP libre (*ephemeral port*)
- Côté serveur :
  - `DatagramSocket ds = new DatagramSocket(7777) ;`
  - Crée une instance attachée à l'adresse wildcard et au port UDP spécifié
- Peut lever des SocketException (problème d'attachement)

D'autres constructeurs DatagramSocket

- On peut être plus précis lors de la création
- Différents constructeurs acceptant des arguments:
  - port + InetAddress
    - Attache au port sur l'adresse spécifiée
  - InetSocketAddress (@IP+port, éventuellement wildcard)
    - Idem
- Permet de choisir une interface plutôt qu'une autre, ou de retarder l'attachement

D'autres constructeurs DatagramSocket

- Choisir l'interface de communications

```
InetAddress ip = NetworkInterface.getByName("eth0")
   .getInetAddresses().nextElement();
// 0 pour choisir un port libre, et
// la socket écoute et envoie à partir de ip
DatagramSocket ds = new DatagramSocket(0,ip);
System.out.println(ds.getLocalSocketAddress());
// mymachine/10.1.54.126:7777
```

- Ou de retarder l'attachement :

```
DatagramSocket ds = new DatagramSocket(null);
// attente d'être prêt...
ds.bind(new InetSocketAddress(7777));
System.out.println(ds.getLocalSocketAddress());
// 0.0.0.0/0.0.0.0:7777
```

Observations sur DatagramSocket

- getLocalPort(), getLocalAddress() et getLocalSocketAddress()
  - retournent les infos sur la socket attachée localement
    - Numéro de port, InetAddress et InetSocketAddress locaux
- bind(SocketAddress)
  - attache la socket si elle ne l'est pas déjà
    - isBound() retourne false
- close()
  - ferme la socket (libère les ressources système associées)
- isClosed()
  - Permet de savoir si la socket est fermée

Créer un DatagramPacket

- Représente un objet spécifiant les données qui doivent transiter ainsi que l'interlocuteur
- Plusieurs constructeurs existent, qui spécifient:
  - les données
    - byte[] buffer
    - int offset
    - int length
  - L'interlocuteur distant
    - soit une InetSocketAddress
    - soit une InetAddress et un numéro de port (int)

Un objet, deux usages

- En émission, le DatagramPacket spécifie
  - les données à envoyer
  - la machine (et le port) vers qui les envoyer
- En réception, le DatagramPacket spécifie
  - la zone de données permettant de recevoir
  - mettra à jour, lors de la réception, la machine et le port depuis lesquels ces données ont été reçues
- Un même objet peut servir aux deux usages

Émission, réception

- Un DatagramPacket est fourni à (et pris en charge par) un DatagramSocket
- Pour émettre:
  datagramSocket.send(datagramPacket);
- Pour recevoir:
  datagramSocket.receive(datagramPacket);
  - Appel bloquant tant que rien n'est reçu
- Peuvent lever différentes IOException

Observations sur DatagramPacket

- Différentes méthodes d'accès aux champs
  - [set/get]Address(), [set/get]Port(), getSocketAddress()
    - Concerne la machine distante (qui a émis ou va recevoir)
  - [set/get]Data(), getOffset(), setLength()
    - permet de spécifier les données ou les bornes dans le tableau d'octets
  - getLength() comportement « normal » en émission mais:
  - getLength() **avant** réception: **taille de la zone** de stockage
  - getLength() **après** réception: **nombre d'octets reçus**

Précisions

- Les données reçues au delà de la taille de la zone de stockage sont perdues
- En pratique, les plateformes limitent le plus souvent à 8K
  - Peuvent se contenter d'accepter 576 octets, en-tête IP comprise
- Le système peut fixer des tailles des tampons de réception et d'émission (pour plusieurs paquets)
  - On peut demander à les changer, sans garantie de succès
    - [get/set]ReceiveBufferSize() et [get/set]SendBufferSize()
- Risque de perte de datagramme:
  - on peut vouloir limiter l'attente en réception
  - socket.setSoTimeout(int milliseconds) interrompt l'attente de réception au delà de milliseconds
    - lève alors une exception SocketTimeoutException

Exemple d'émission (client)

```
// socket sur un port libre et sur l'adresse wildcard locale
DatagramSocket socket=new DatagramSocket();

// tableau d'octets correspondant à la String "Hello" en ASCII
byte[] buf="Hello".getBytes("ASCII");

// création de l'adresse de socket destinataire :
// port 3333 de la machine de nom "serveur"
InetAddress ipServ=InetAddress.getByName("serveur");
SocketAddress dest=new InetSocketAddress(serv,3333);

// création d'un datagramme contenant ces données
// et destiné à cette adresse de socket
DatagramPacket packet=new DatagramPacket(buf,buf.length,dest);

// envoi du datagramme via la socket
socket.send(packet);
```

Exemple de réception (client)

```
// (sur la même socket qui a envoyé "Hello")
// allocation et mise en place d'un buffer pour la réception
byte[] receiveBuffer=new byte[1024];
packet.setData(receiveBuffer);
System.out.println(packet.getLength());
// affiche: 1024 (c'est la taille de la zone de stockage)

// mise en attente de réception
socket.receive(packet);
System.out.println(packet.getLength());
// affiche le nombre d'octets effectivement reçus (<= 1024)

// construction d'une String correspondant aux octets reçus
String s=new String(receiveBuffer, 0,
                    packet.getLength(), "ASCII");
System.out.println(s);
// Quelle est la taille de la zone de stockage ici?

socket.close();
```

Exemple de réception (serveur)

- Dans le cas d'un client, le choix du numéro de port d'attachement de la socket n'a pas d'importance
  - il sera connu par le serveur à la réception du datagramme émis par le client
- En revanche, dans le cas d'un serveur, il doit pouvoir être communiqué aux clients, afin qu'ils puissent l'interroger
  - nécessité d'attacher la socket d'écoute (d'attente des clients) à un port et une adresse spécifique et connue
  - utiliser un constructeur de DatagramSocket le spécifiant
    - risque que le port ne soit pas libre => SocketException

Réception serveur (suite)

```
// socket d'attente de client, attachée au port 3333
DatagramSocket socket = new DatagramSocket(3333);
// datagramme pour la réception avec allocation de buffer
byte[] buf = new byte[1024];
DatagramPacket packet = new DatagramPacket(buf,buf.length);
byte[] msg = "You're welcome!".getBytes("ASCII");
                // message d'accueil

while (true) {
  socket.receive(packet);// attente de réception bloquante
  // ce serveur ignore ce qu'il reçoit
  // Il place les données à renvoyer
  // (@ip et port distant sont déjà ok)
  packet.setData(msg);
  socket.send(packet);                  // envoie la réponse
  packet.setData(buf,0,buf.length); // replace la zone de
                                        // réception
}
```

La pseudo-connexion (faux ami !)

- Dans le cas où un serveur ne veut reçevoir QUE ce qui provient un client donné
  - possibilité de ne considérer **que** ce client durant ce qu'on appelle une « pseudo-connexion »
  - les sockets du serveur et du client sont dites « paires »
  - connect(InetAddress, int) ou connect(SocketAddress) pour établir la pseudo-connexion
  - disconnect() pour terminer la pseudo-connexion
  - pendant ce temps, tous les datagrammes en provenance d'autres ports/adresses IP sont ignorés
  - informations sur la pseudo-connexion: isConnected(), getInetAddress(), getPort(), getRemoteSocketAddress()
  - Attention à la différence entre getPort() et getLocalPort()
- Pseudo-connexion UDP : rien à voir avec une connexion (TCP) !

Communication en diffusion

- D'un émetteur vers un groupe ou un ensemble de récepteurs
  - le broadcast utilise les mêmes classes DatagramSocket et DatagramPacket que pour la communication *unicast*, avec une adresse destination de *broadcast*
  - le multicast utilise la classe DatagramPacket pour les datagrammes mais la classe MulticastSocket, spécifique pour les sockets
- Efficace lorsque le protocole de réseau sous-jacent offre la diffusion (ex. Ethernet):
  - un seul datagramme (UDP/IP) peut être utilisé pour joindre plusieurs destinataires
  - Une seule trame (Ethernet) dans le meilleur des cas...

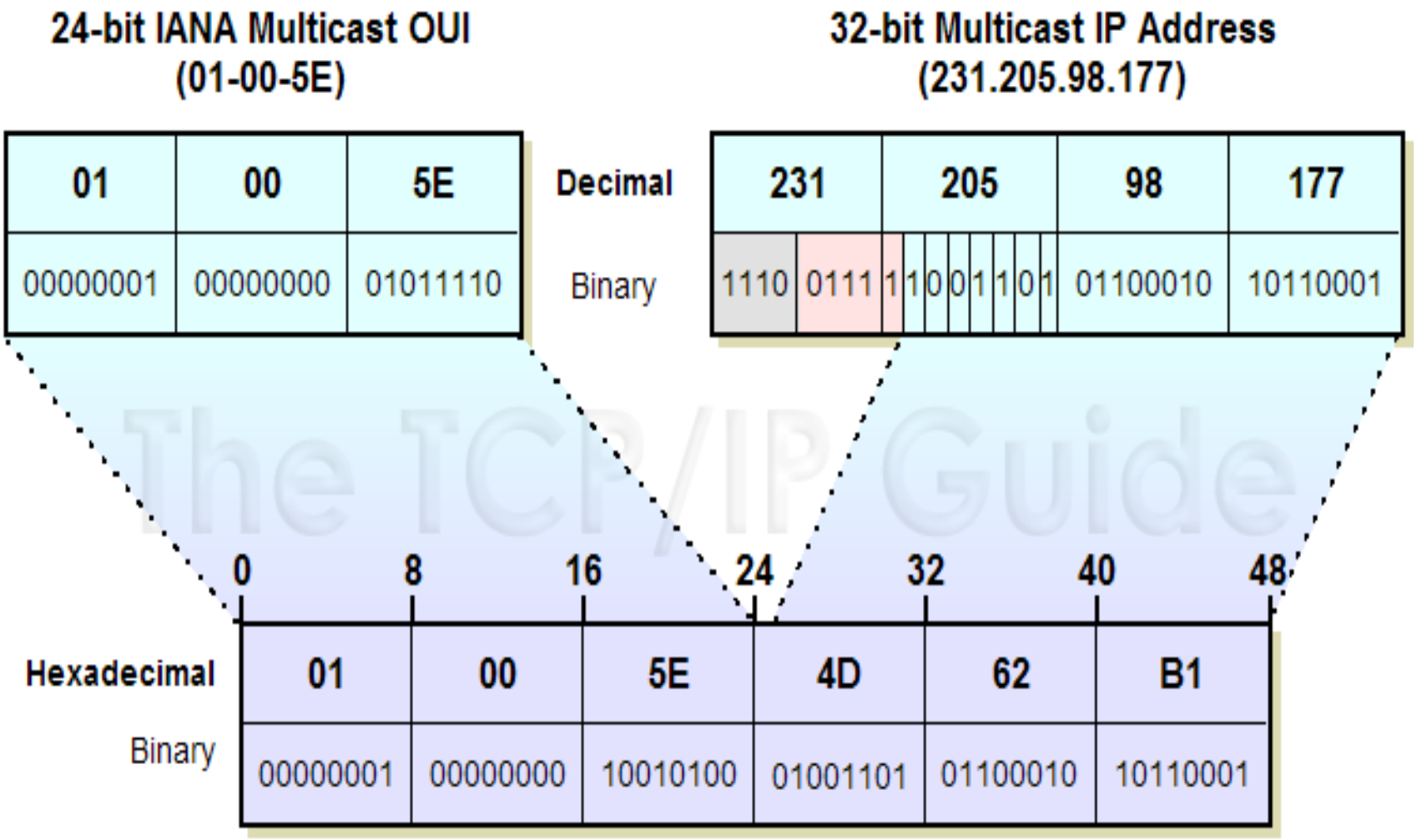
Le broadcast

- La classe DatagramSocket dispose de méthodes [set/get]Broadcast()
  - autorisation pour la socket d'émettre des *broadcasts*
  - SO_BROADCAST true par défaut
- En réception, une socket ne peut recevoir des *broadcasts* que si elle est attachée à l'adresse non spécifiée (*wildcard*)

Le multicast

- On s'adresse à un ensemble de machines (ou applications) ayant explicitement adhéré au groupe
  - adresses de classe D en IP v4: 224.0.0.0/4
  - adresses du réseau FF00::/8 en IP v6
  - les machines ayant rejoint un tel groupe acceptent les datagrammes à destination de l'adresse correspondante
- Traduction d'adresse IP multicast vers IP Ethernet
  - IP v4: l'@ Ethernet est le résultat du OU binaire entre les 23 bits de poids faible de l'adresse IP v4 et 01:00:5E:00:00:00
  - IP v6: l'adresse ffxx:xxxx:xxxx:xxxx:xxxx:xxxx:yyyy:yyyy produit l'adresse Ethernet 33:33:yy:yy:yy:yy

Mapping multicast IPv4



http://www.tcpipguide.com/free/diagrams/arpmulticast.png

java.net.MulticastSocket

- Hérite de DatagramSocket
- Trois constructeurs de MulticastSocket
  - sans arguments: attache à un port libre et à l'@ wildcard
  - numéro de port: attache à ce port et à l'@ wildcard
  - SocketAddress: attache à cette adresse de socket, ou bien n'attache pas la socket si l'argument vaut null
- Les constructeurs appellent la méthode setReuseAddress(true)
  - autorise plusieurs sockets à s'attacher à la même adresse
  - **MÊME PORT** pour tous les membres du groupe

MulticastSocket en émission

- En émission, MulticastSocket s'utilise comme DatagramSocket
  - possibilité de spécifier une adresse IP source pour l'émission des datagrammes: [set/get]Interface()
  - possibilité de spécifier une interface de réseau pour l'émission: [set/get]NetworkInterface()
  - pour émettre vers le groupe, l'adresse de destination et le numéro du port dans le DatagramPacket doivent être ceux du groupe

MulticastSocket en émission (suite)

- On peut spécifier une « durée de vie » pour le datagramme: plus exactement, un nombre de sauts
  - 0=émetteur, 1=réseau local, 16= site, 32=région, 48=pays, 64=continent, 128=monde
  - méthodes [set/get]TimeToLive()
  - vaut 1 par défaut ⇒ propagé par aucun routeur
- Envoi des datagrammes multicast sur loopback
  - [set/get]LoopbackMode() peut être refusé par le système
- Suffit pour envoyer vers le groupe, pas pour recevoir.

MulticastSocket en réception

- Il faut explicitement « rejoindre » le groupe
  - **joinGroup()** avec l'adresse IP multicast du groupe en argument (existe aussi avec SocketAddress et/ou NetworkInterface)
  - permet alors à la socket de recevoir tous les datagrammes destinés à cette adresse multicast
    - Ajoute à l'interface une nouvelle @ IP et une nouvelle @MAC
    - Annonce l'abonnement au groupe à l'intention du MRouteur...
  - peut rejoindre plusieurs groupes de multicast
  - **leaveGroup()** permet de quitter le groupe d'adresse spécifiée en argument