Applications réseaux

accès à UDP

Arnaud Carayol

Etienne Duris

UNIVERSITÉ PARIS-EST
MARNE-LA-VALLÉE

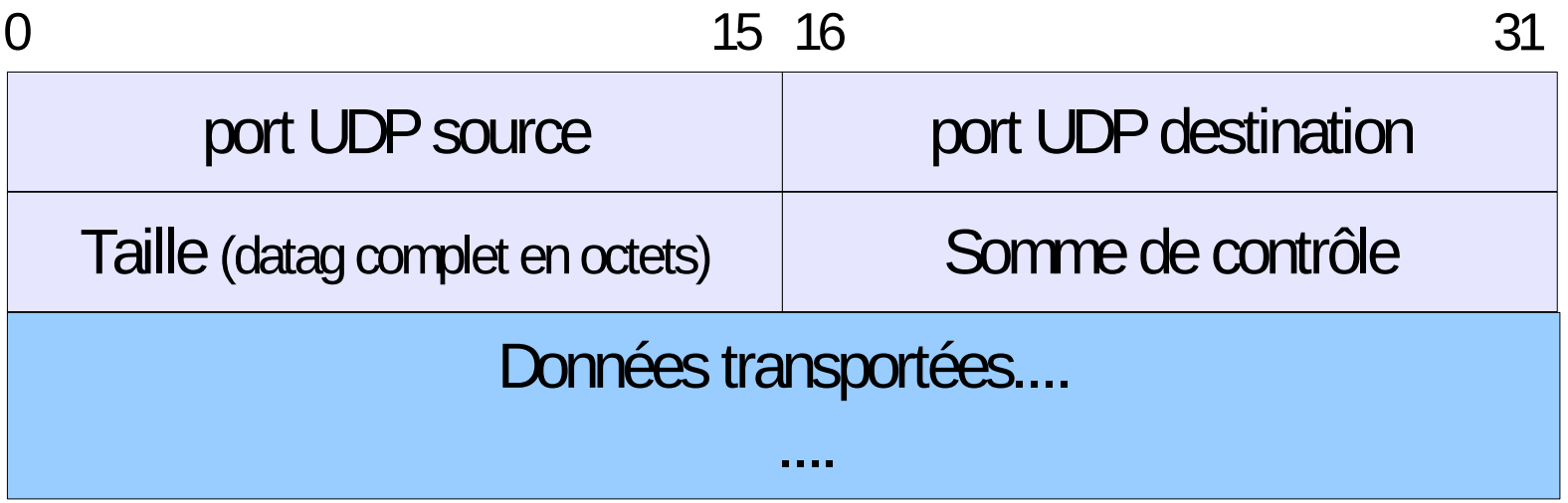
Esipe
INGÉNIEURS PARIS EST

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

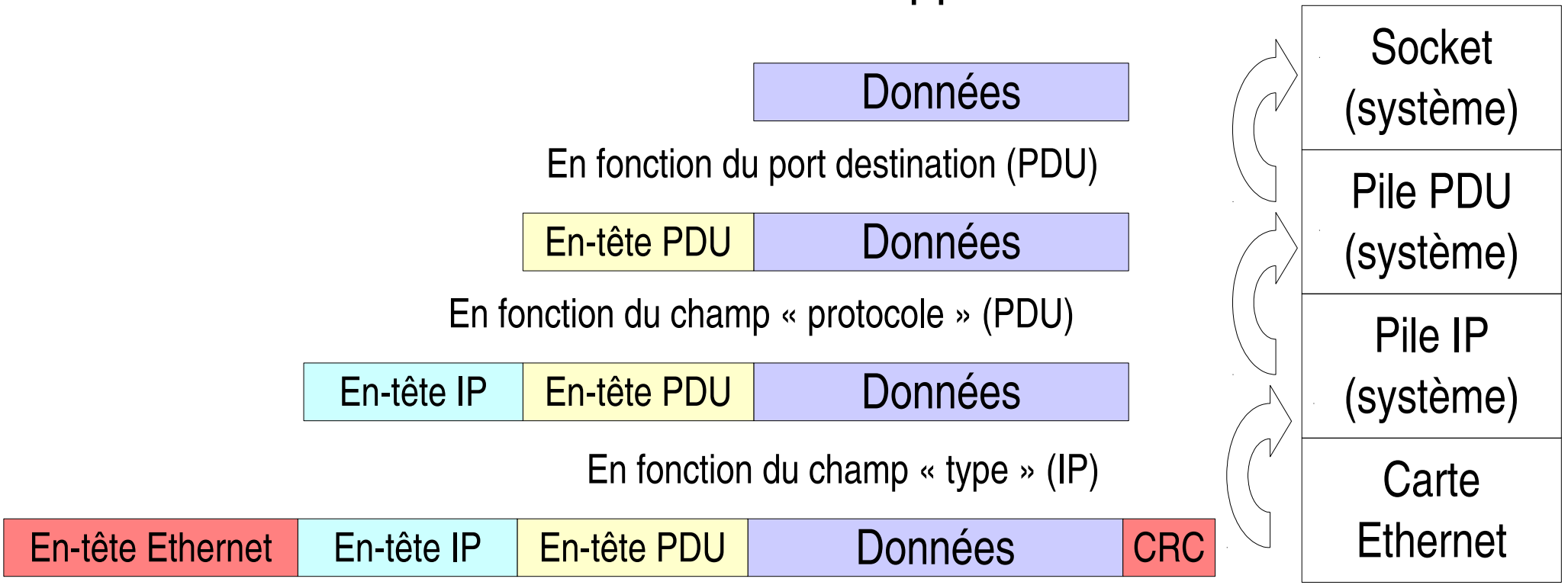
- Taille max des données transportées: (2^{16}-1-8)~64Ko
- Checksum optionnel en IP v4, obligatoire en IP v6
  - ajout (pour le calcul) d'un pseudo-en-tête avec @ dest et src



Exemple d'encapsulation dans une trame Ethernet:

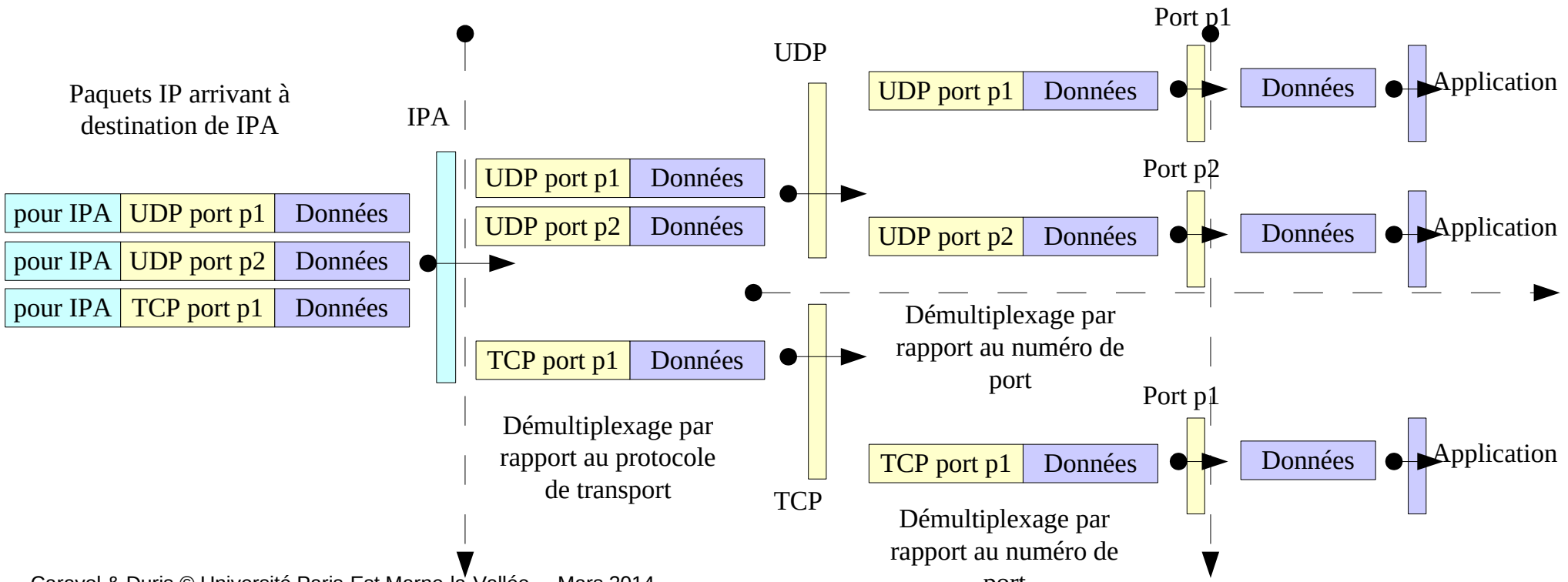
Encapsulation des données

- 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

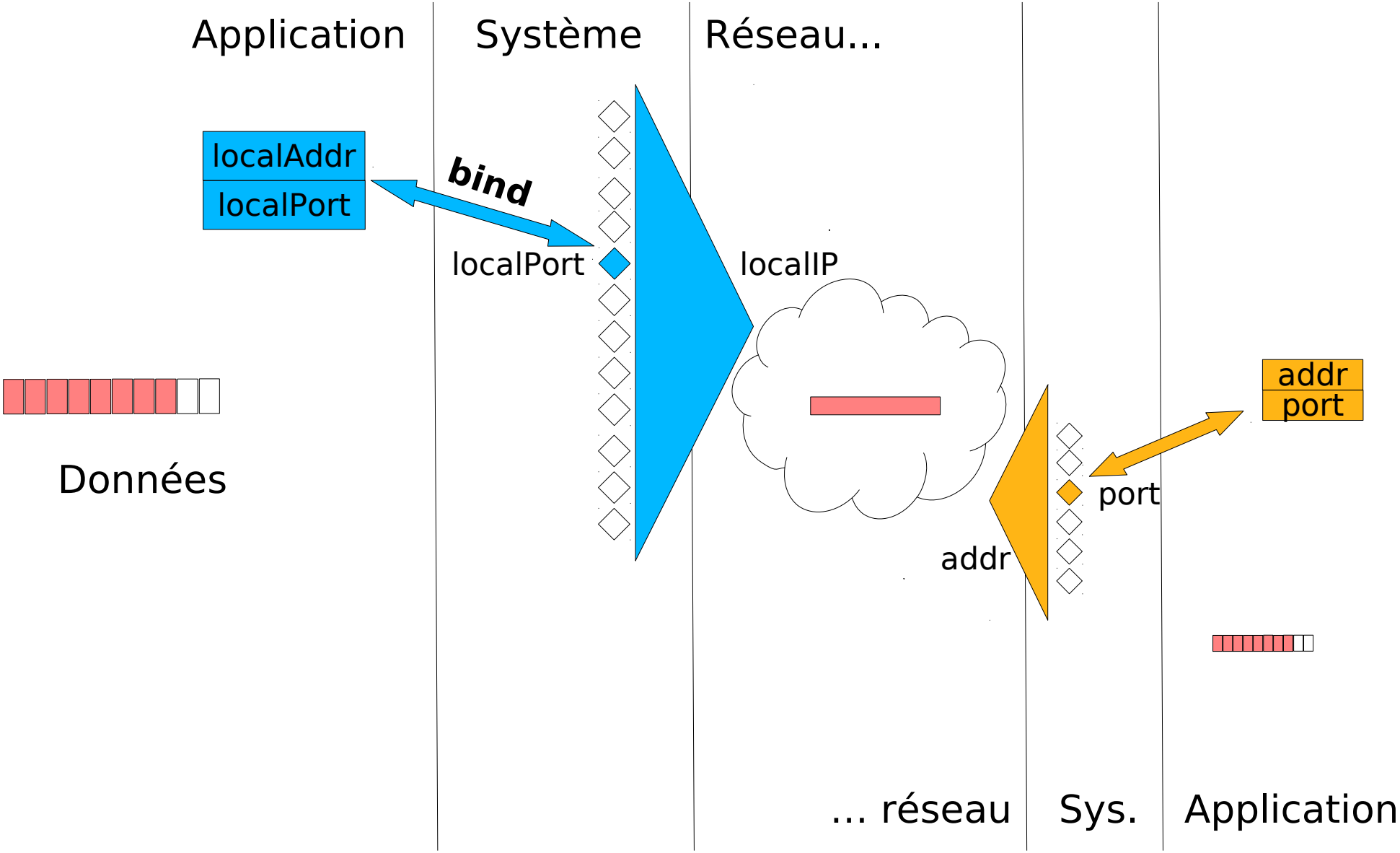


Multiplexage / démultiplexage

- 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 ou java.nio.channels.DatagramChannel pour UDP
  - java.net.Socket ou java.nio.channels.SocketChannel 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

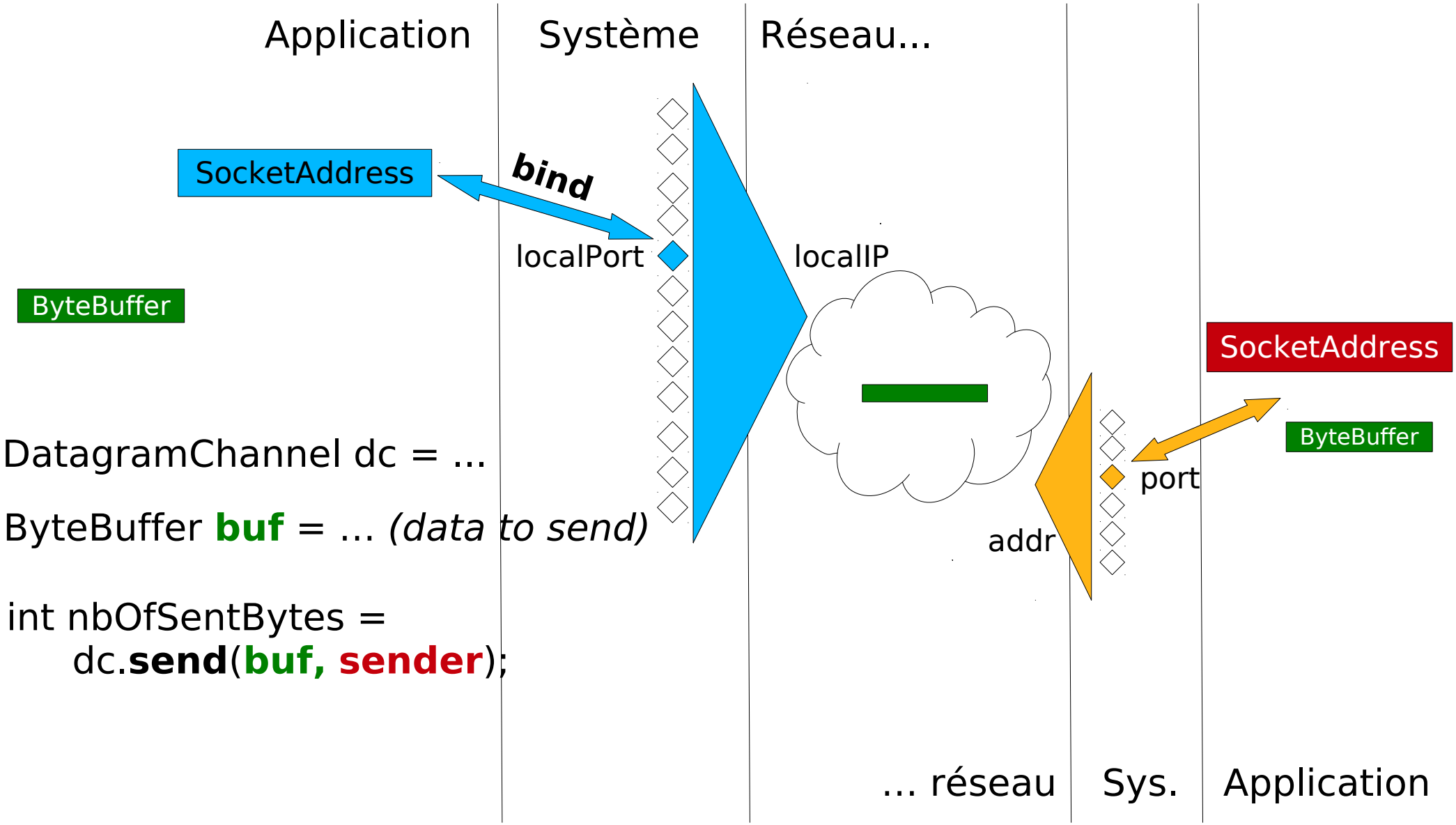
Accès Java à UDP (à l'ancienne)

- 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

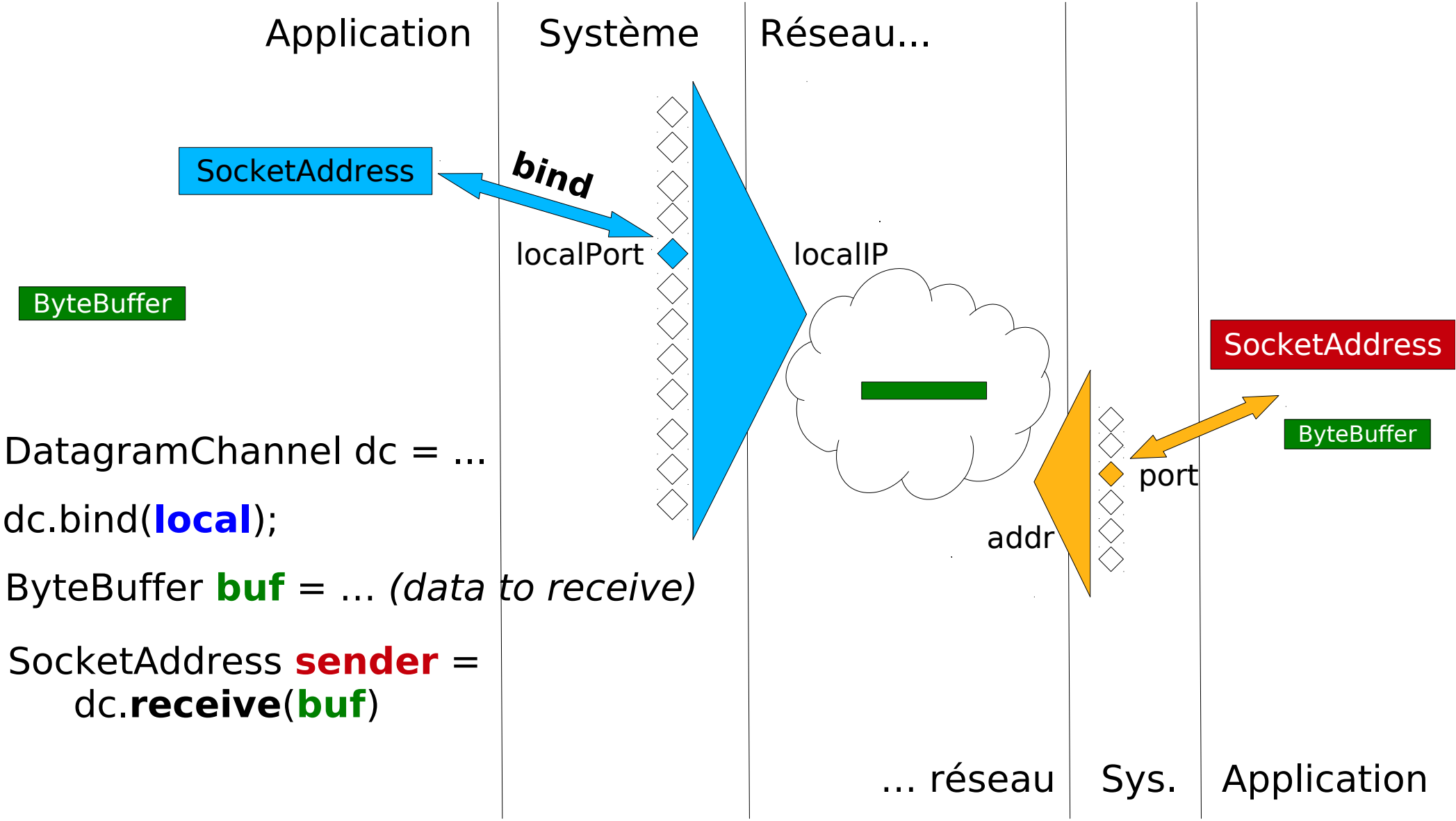
Accès Java à UDP (avec les nio)

- Le SDK 1.4 offre une autre manière d'accéder aux entrées/sorties (nio) à travers plusieurs nouveaux concepts/classes
  - ***Buffers*** (tampons mémoire) java.nio.*
  - ***Charsets*** (jeux de caractères) java.nio.charset.*
  - ***Channels*** (canaux) java.nio.channels.*
    - Gestion plus fine de la mémoire
    - Gestion plus performante des entrées-sorties
    - Gestion simplifiée des différents jeux de caractères
    - Interaction plus fine avec le système de fichiers
    - Utilisation d'entrées-sorties non bloquantes
  - Pour UDP, cela se traduit par une seule classe
    - java.nio.channel.DatagramChannel
      - représente un « canal » permettant de lire et d'écrire
      - Dans le cas d'UDP – non connecté – c'est plus réducteur :
        - **envoyer et recevoir**

Les différents niveaux impliqués (envoi)



Les différents niveaux impliqués (réception)



Un DatagramChannel en émission

- Un canal (non attaché) se crée avec la méthode open()
  - `DatagramChannel dc = DatagramChannel.open();`
- Si besoin, il peut être attaché à une adresse de socket locale
  - `dc.bind(localSocketAddr);`(attachement à un port libre si null)
- Pour envoyer, il faut disposer des données et du destinataire
  - `ByteBuffer buf = …`
  - `SocketAddress target = …`
  - `int nb = dc.send(buf,target);`
    - Retourne le nombre d'octets envoyés
    - En mode bloquant (par défaut) et en UDP c'est le nombre d'octets restants (remaining) dans buf – préservation des limites !

DatagramChannel en réception

- Pour recevoir, il faut disposer d'une zone de données, et l'opération de réception retourne la SocketAddress de l'émetteur
  - `ByteBuffer buf = …`
  - `SocketAddress target = dc.receive(buf);`
- En mode bloquant (par défaut), la méthode est bloquante tant que rien n'est reçu
  - Au plus `buf.remaining()` octets peuvent être reçus
  - Le reste est tronqué
- Deux envois ou deux réceptions sur un même canal ont toujours lieu l'une après l'autre entre plusieurs threads
  - Évite de devoir exclure mutuellement

Notions essentielles sur les buffers

- Utilisés par les primitives d'entrées-sorties de java.nio
  - Remplace les tableaux utilisés en java.io
    - IO « classiques » : ex. InputStream : read(byte[])
    - NIO : ex. Channel : read(ByteBuffer)
  - Buffer = zone de mémoire contiguë, permettant de stocker
    - une quantité de données fixée,
    - d'un type primitif donné
  - A priori pas prévus pour un accès concurrent
    - il faudra les protéger en cas de besoin...
- Représentés par des classes abstraites
  - Permet de dédier l'implémentation native à la plate-forme d'accueil

Classes abstraites des tampons

- Classe abstraite Buffer
  - factorise les opérations indépendantes du type primitif concerné
- Classe abstraite ByteBuffer
  - fournit un ensemble de méthodes et d'opérations plus riche que pour les autres types de buffer
- Classes abstraites dédiées à des types primitifs
  - CharBuffer, ShortBuffer, IntBuffer, LongBuffer, FloatBuffer et DoubleBuffer

Allocation et accès aux tampons

- Allocation de buffer d'un type primitif donné (*prim*):
  - méthodes statiques dans les classes *Prim*Buffer
    - *Prim*Buffer allocate(int capacity)
    - Ex : ByteBuffer buf = ByteBuffer.allocate(1024) ;
- Deux manières d'accéder aux éléments d'un buffer
  - Accès **aléatoire** (*absolute*)
    - Relativement à un indice (comme dans un tableau)
  - Accès **séquentiel** (*relative*)
    - Relativement à la position courante (comme un flot)
      - La position courante représente l'indice du prochain élément à lire ou à écrire

Accès aléatoire *vs* séquentiel

- Si *Prim*Buffer est un buffer d'éléments de type *prim :*
- Accès **aléatoire** (*absolute*) : comme un tableau
  - *prim* **get**(int index) donne l'élément à la position index
  - *prim*Buffer **put**(int index, *prim* value) ajoute value à l'indice index, et renvoie le buffer modifié
    - comme StringBuilder.append()
  - Peuvent lever IndexOutOfBoundsException
- Accès **séquentiel** (*relative*) : comme un flot
  - *Prim* **get**() resp. *prim*Buffer **put**(*prim* value)
    - Donne la valeur (resp. place value) à la position courante
  - Peuvent lever des exceptions BufferUnderflowException ou BufferOverflowException

Les attributs et méthodes d'un tampon

- **Capacité**: nombre d'éléments qui peuvent être contenus
  - fixée à la création du tampon
  - consultable par int capacity()
- **Limite**: indice du premier élément ne devant pas être atteint
  - par défaut, égale à la capacité.
  - Fixée par Buffer limit(int newLimit)
  - Connue par int limit()
- **Position courante**: indice du prochain élément accessible
  - Consultable: int position()
  - Modifiable: Buffer position(int newPosition)
- Invariant :

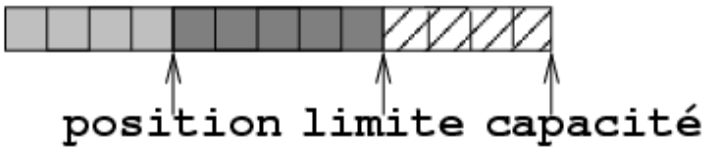
  0 <= position <= limite <= capacité

Les attributs et méthodes d'un tampon

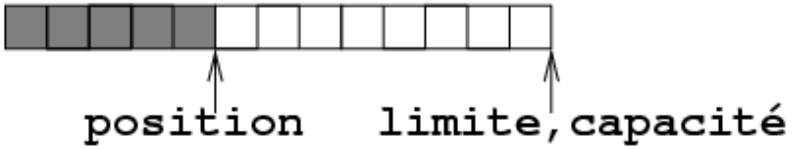
- Quand la position courante vaut la limite
  - Un appel à get() provoque BufferUnderflowException
  - Un appel à put() provoque BufferOverflowException
- Pour éviter ça:
  - int remaining() donne le nombre d'éléments entre la position courante et la limite
  - boolean hasRemaining() vaut vrai si la position est strictement inférieure à limite

Méthodes utilitaires sur les tampons

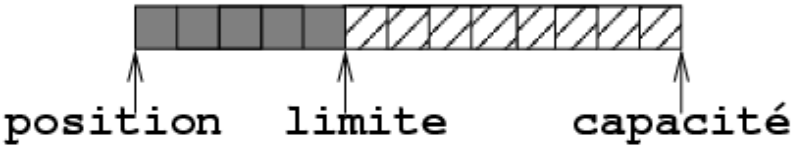
- compact()
  - Place l'élément à la position courante *p* à la position 0, l'élément *p+1* à la position 1, etc. La nouvelle position courante est placée après le dernier élément décalé. La limite est mise à la capacité et la marque effacée.
- flip()
  - limite <- position courante
    position <- 0. Marque indéfinie.
- rewind()
  - position <- 0. Marque indéfinie
- clear() N'efface pas le contenu!




```
bb.compact();
```

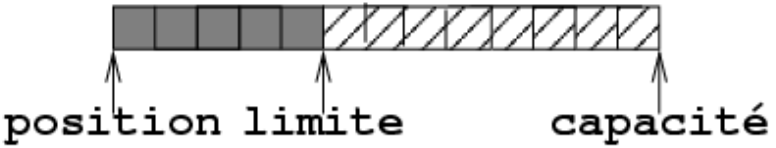



```
bb.flip();
```




```
bb.get();
```


```
bb.rewind();
```




```
bb.clear();
```