

# Module RSA

## Partie Réseaux

Isabelle Chrisment



2013-2014

# Plan

- ① Contrôle de congestion et TCP
- ② Programmation réseaux
  - socket, select(), options de socket, socket raw
- ③ Communication de groupe (multicast)
- ④ Introduction IPv6

## ① Contrôle de congestion et TCP

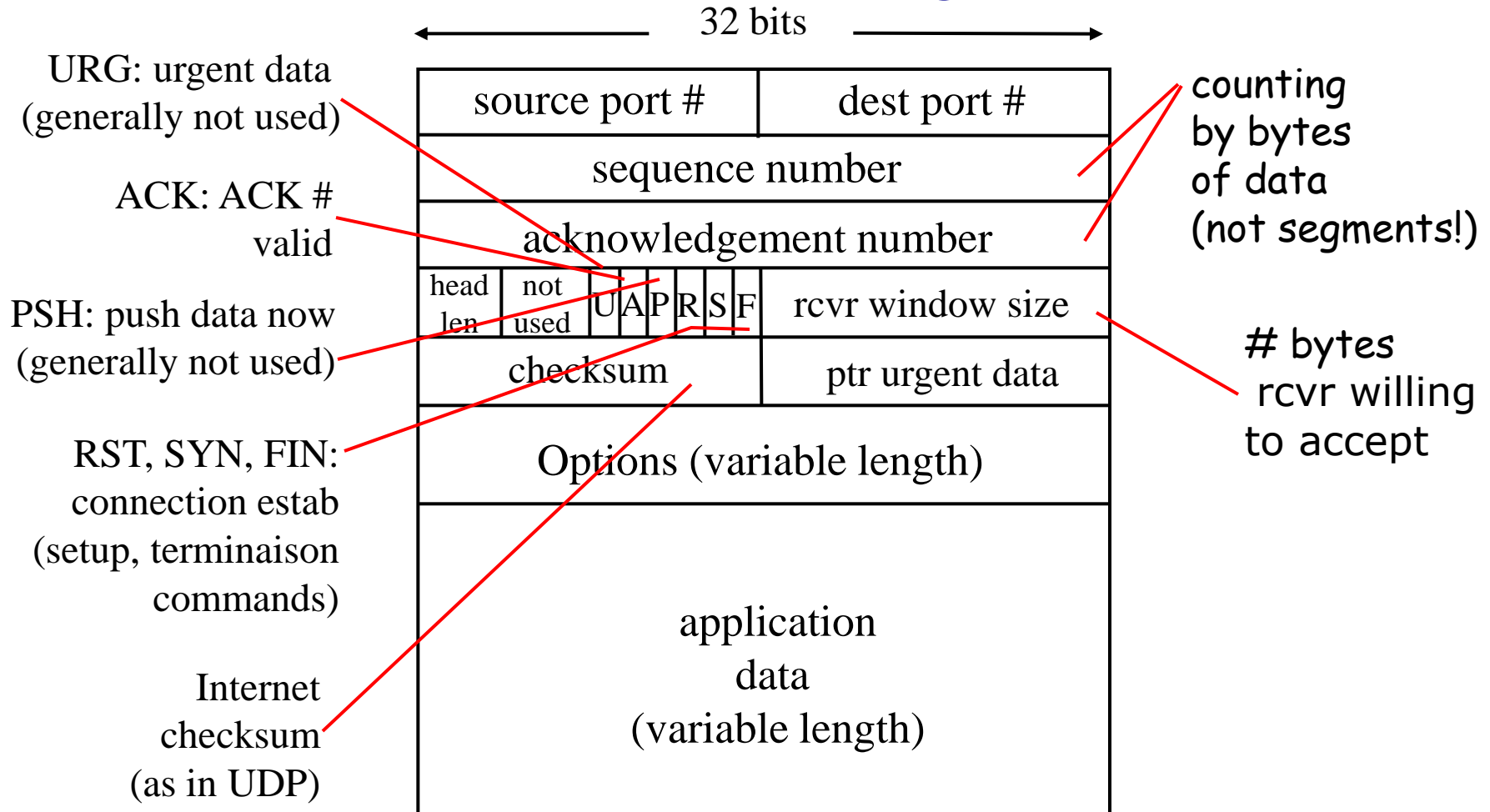
- TCP : quelques rappels
- Congestion Internet
- TCP Tahoe
- TCP Reno
- RTT et TCP
- TCP Vegas
- Équité TCP

# TCP : quelques rappels

RFCs: 793, 1122, 1323, 2018, 2581

- Point-à-point
- Orienté connexion
- Fiable et ordonné
- Pas de frontière de message
- Full duplex
- Flux contrôlé

# Structure d'un segment



Kurose &amp; Ross

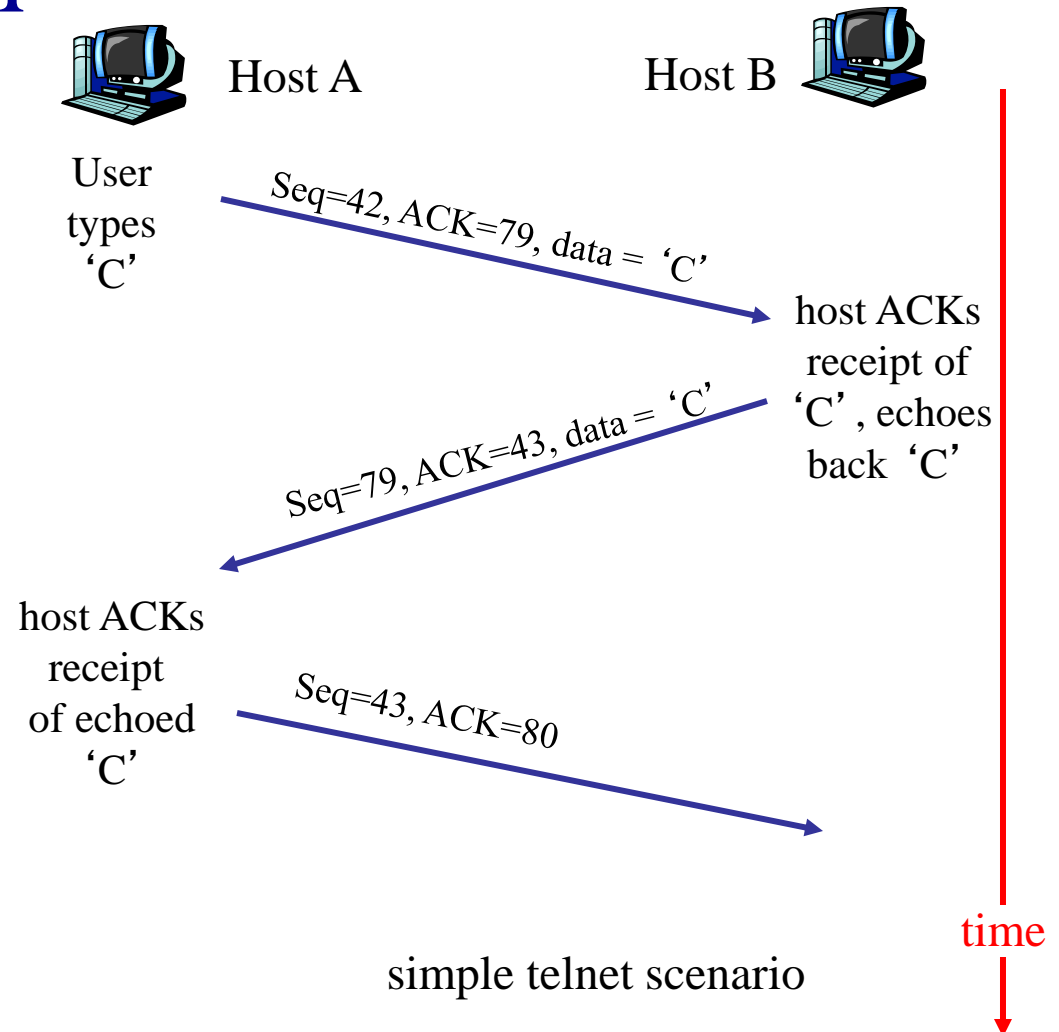
# TCP seq. #'s - ACKs

## Seq. #'s:

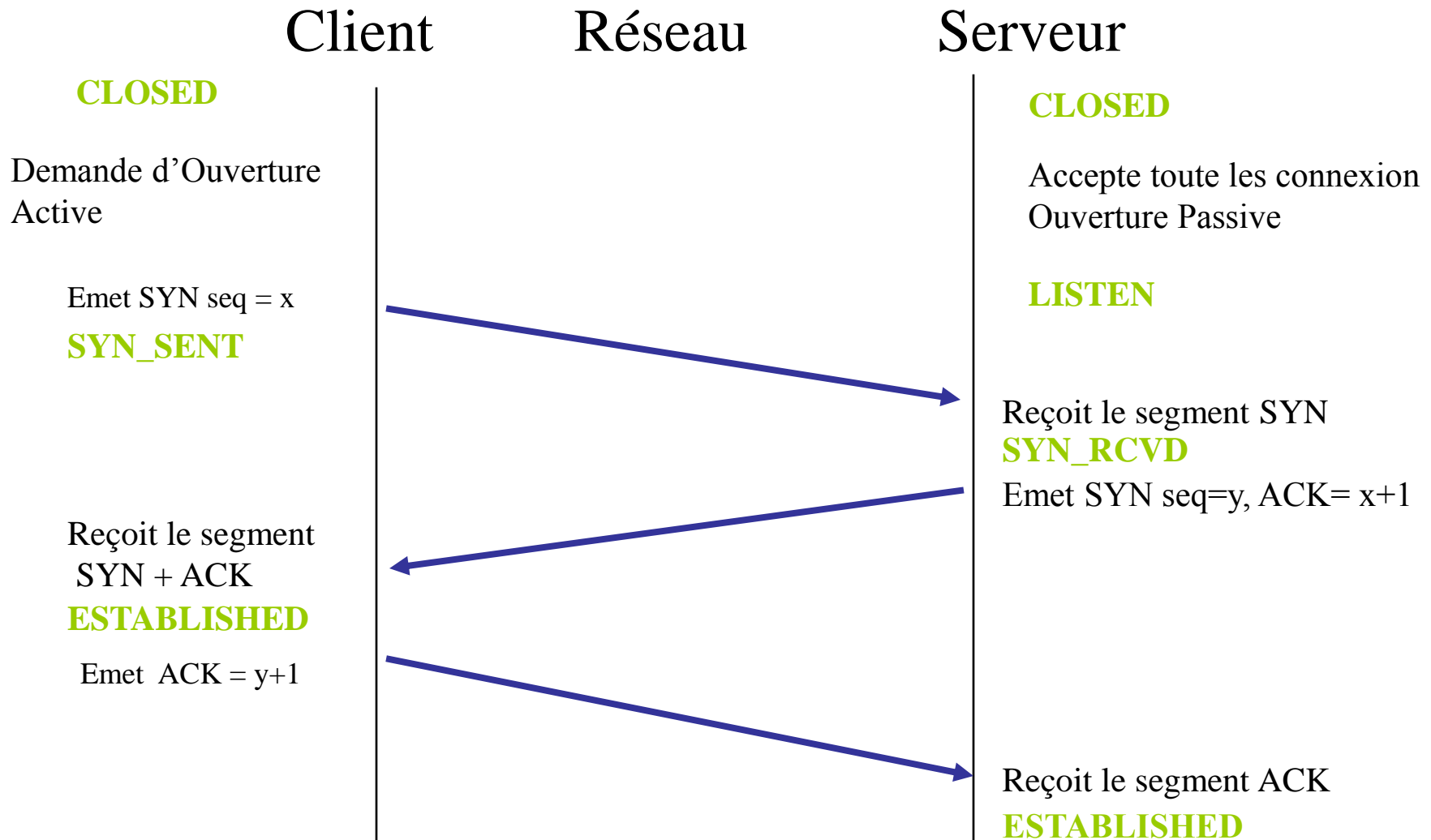
- Flux d'octets  
“numéro” du premier octet dans les données du segment

## ACKs:

- seq # du prochain octet attendu de l'autre côté
- ACK cumulatif

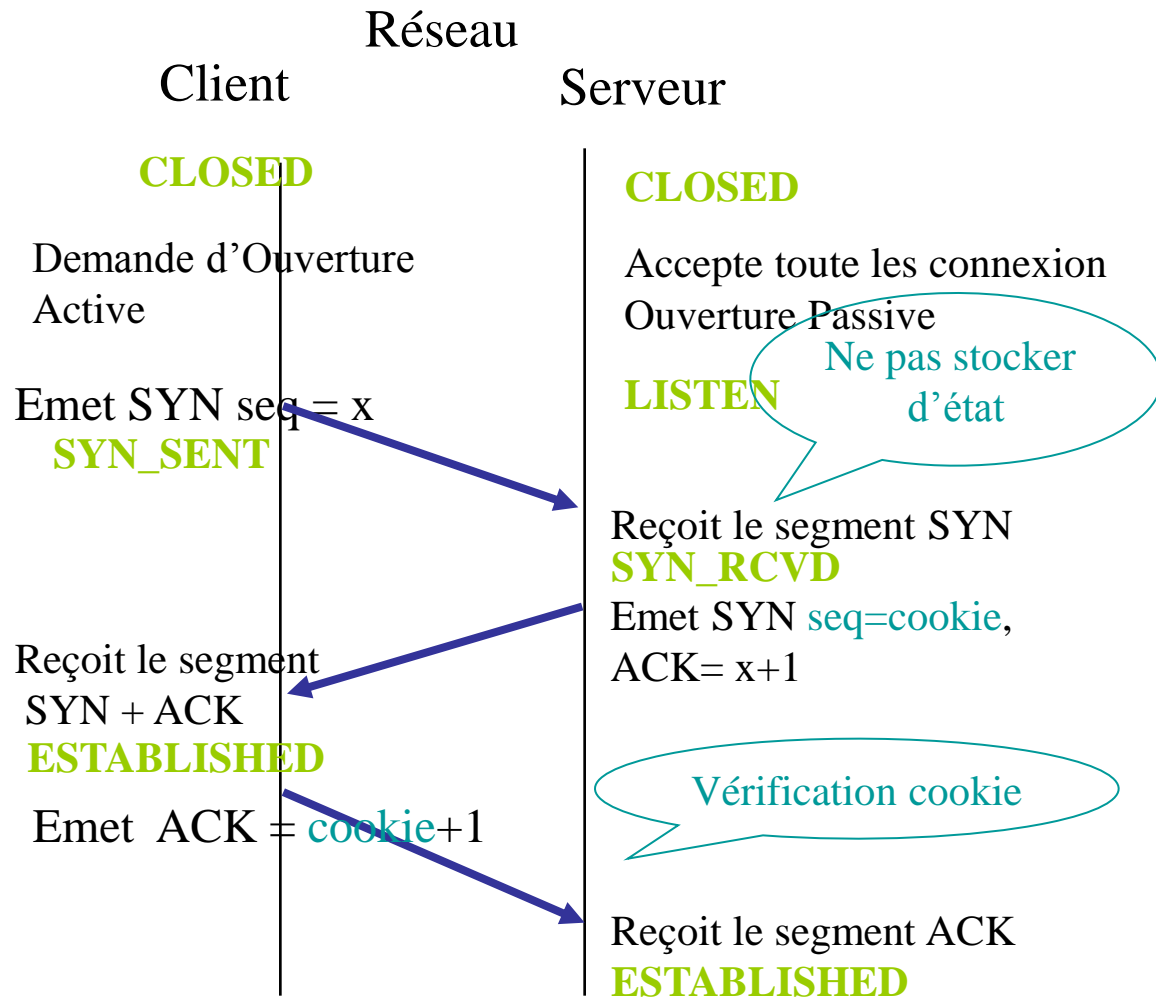


# TCP Etablissement d'une connexion



# TCP syn cookies

- Choix particulier du numéro de séquence côté serveur :  
( cf <http://cr.yp.to/syncookies.html> )
- 5 bits =  $t \bmod 32$  (t compteur de temps sur 32 bits)
- 3 bits : encodage de la valeur du MSS du serveur
- 24 bits  $F(\text{adr. src, port src, adr. dest, port dest, t, secret})$   
avec  $F$  = fonction crypto



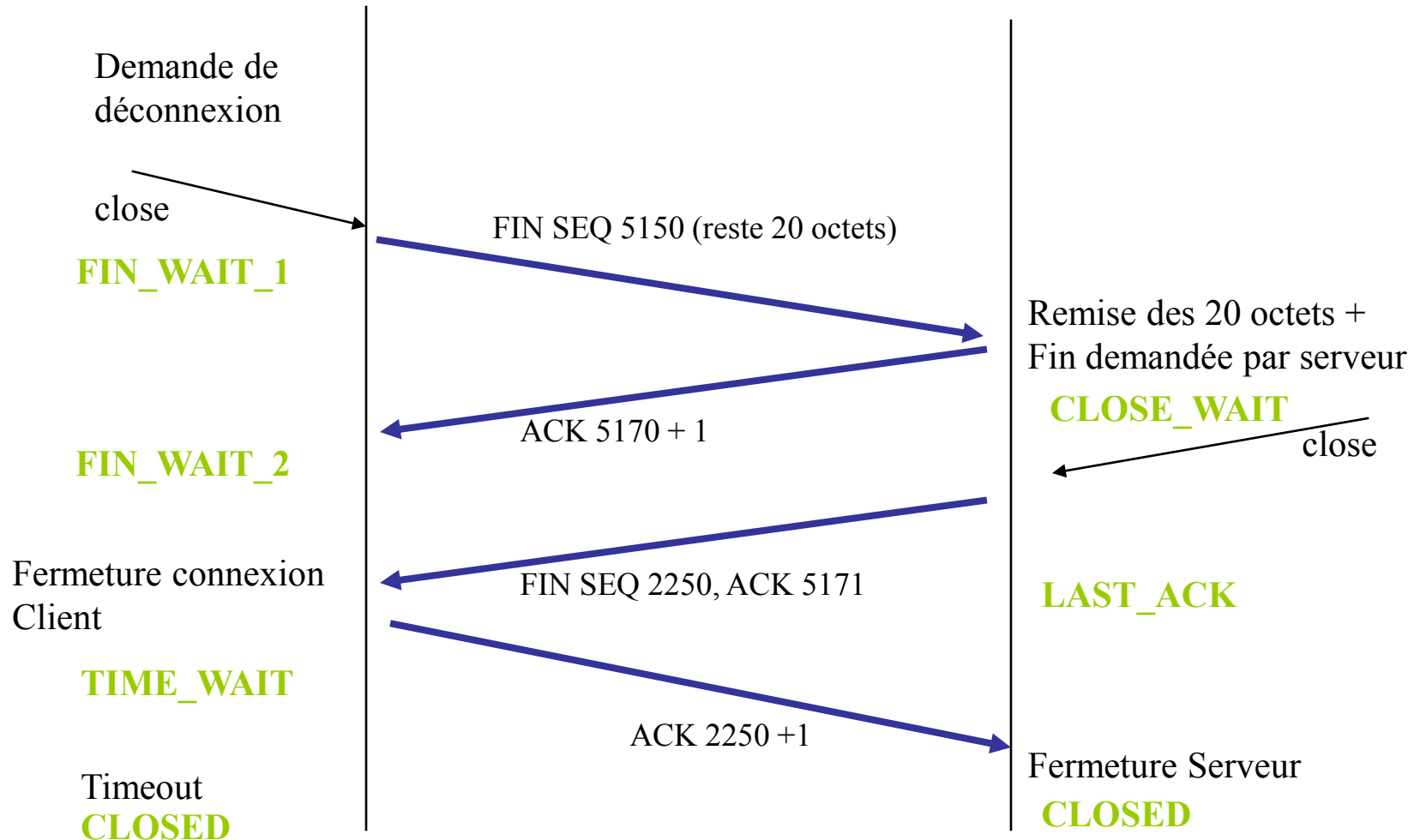


# TCP fermeture connexion

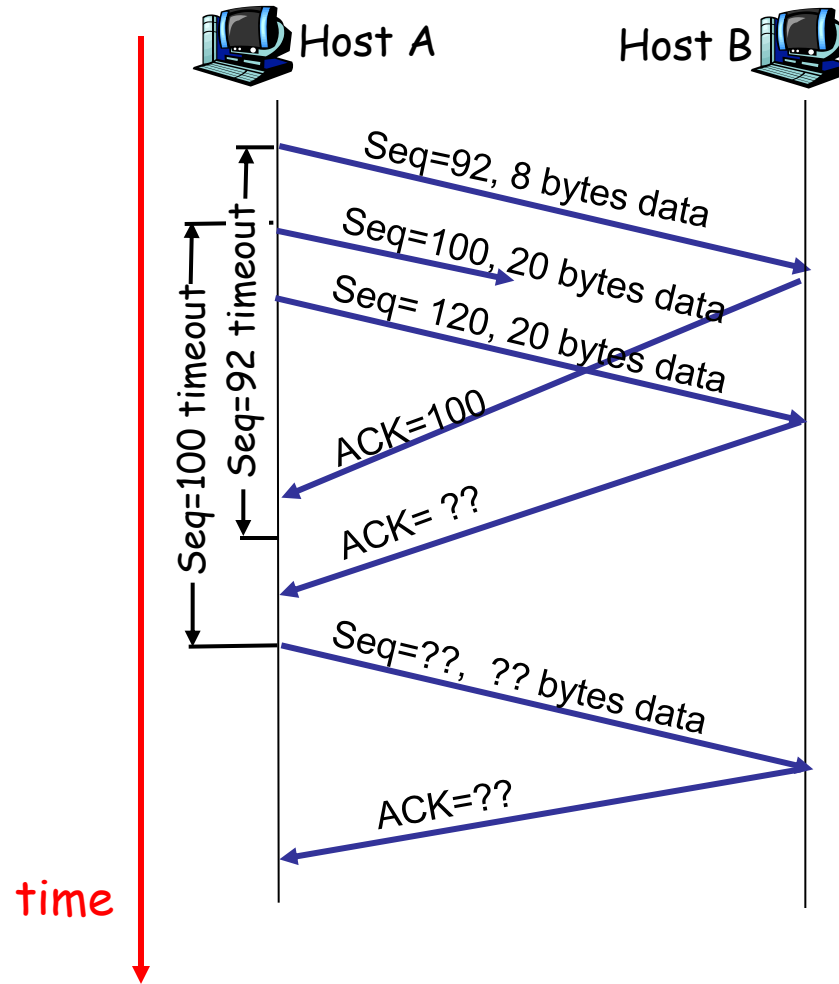
Client

Réseau

Serveur



# TCP: scénario de retransmission

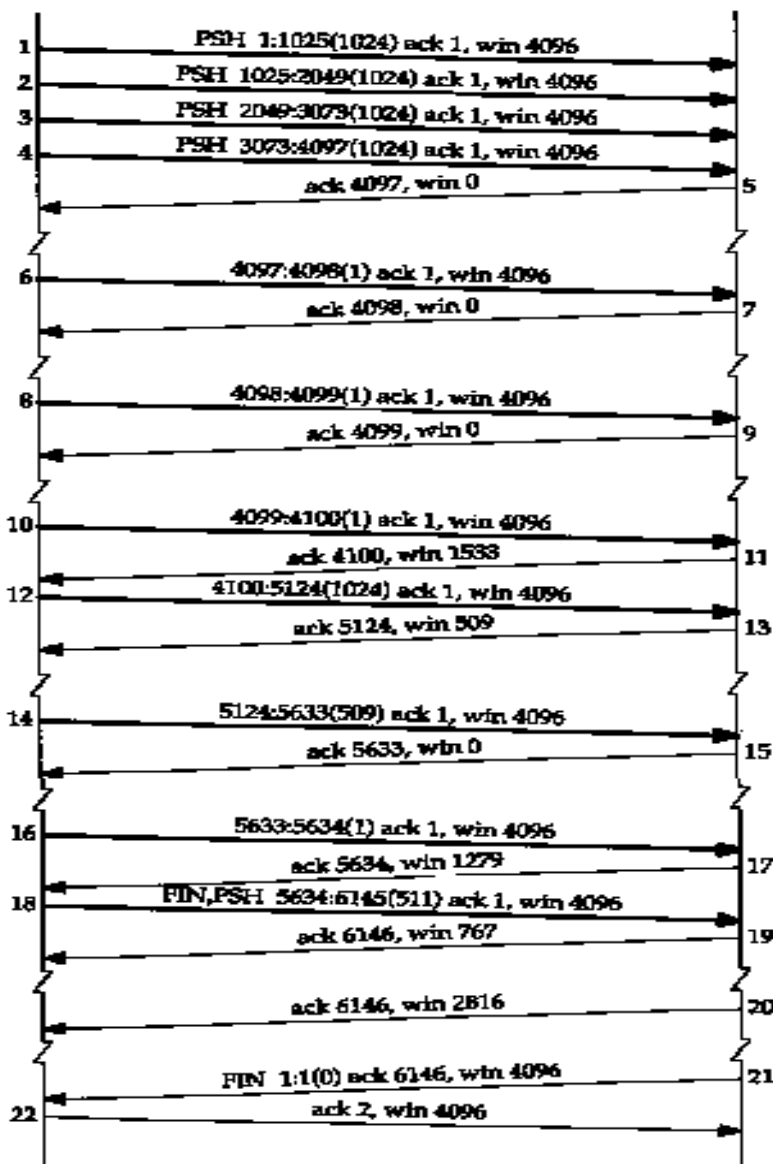


sun.1069

bsd1.7777

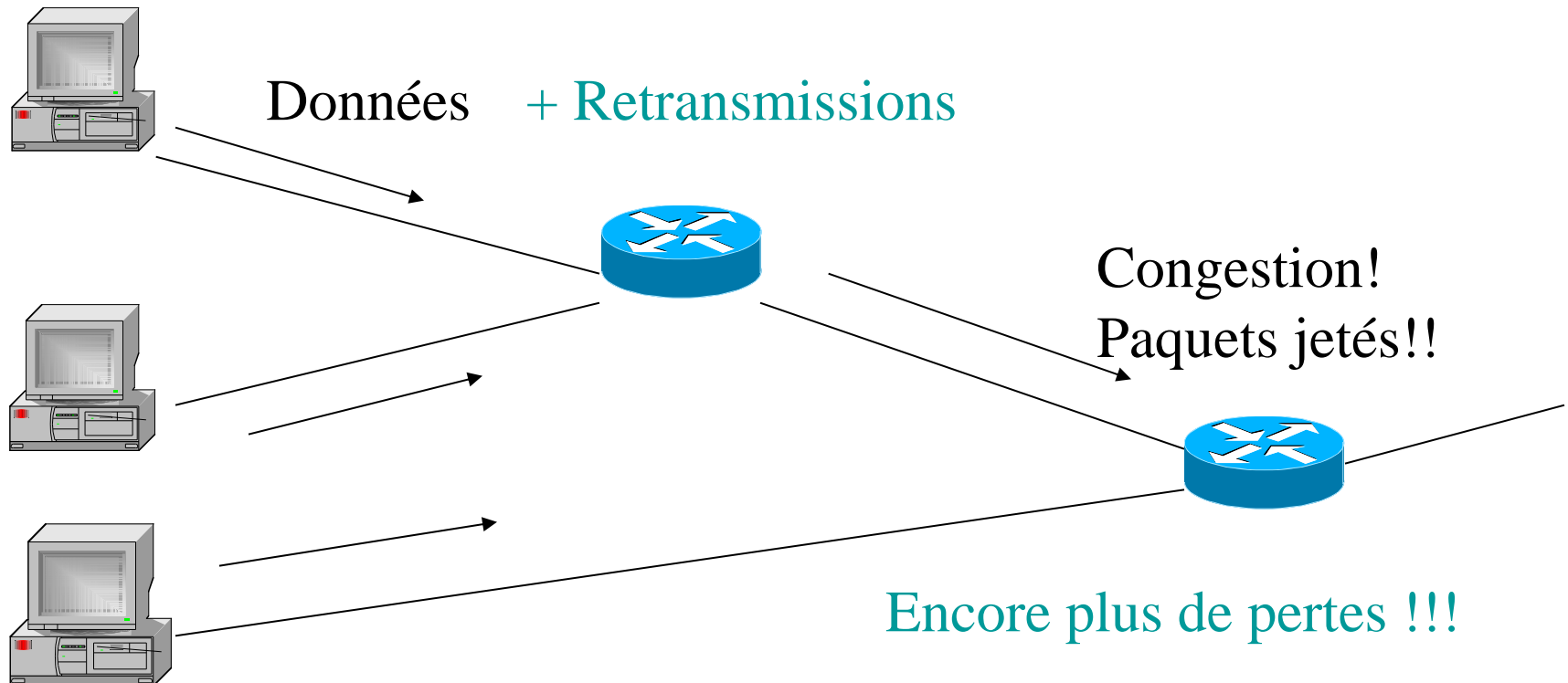
# Silly window avoidance TCP Illustrated Stevens

0.0  
0.002026 ( 0.0020)  
0.003737 ( 0.0017)  
0.005361 ( 0.0016)  
  
0.170306 ( 0.1649)  
  
5.151768 ( 4.9615)  
5.170308 ( 0.0185)  
  
10.151592 ( 4.9813)  
10.170299 ( 0.0187)  
  
15.151466 ( 4.9812)  
15.170296 ( 0.0188)  
15.172006 ( 0.0017)  
15.370307 ( 0.1983)  
  
20.151782 ( 4.7815)  
20.170297 ( 0.0185)  
  
25.151162 ( 4.9809)  
25.170302 ( 0.0191)  
25.171801 ( 0.0015)  
25.174401 ( 0.0026)  
  
39.991658 (14.8173)  
  
51.991775 (12.0001)  
51.992665 ( 0.0009)

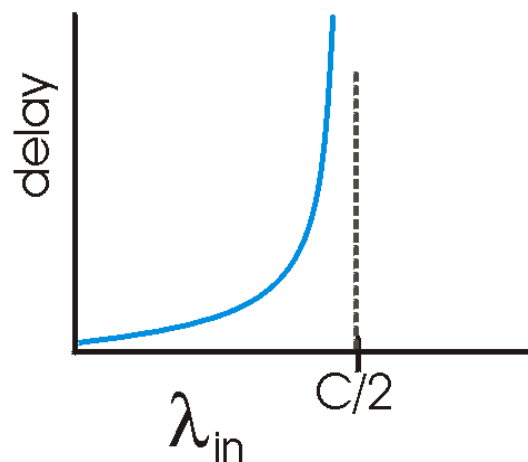
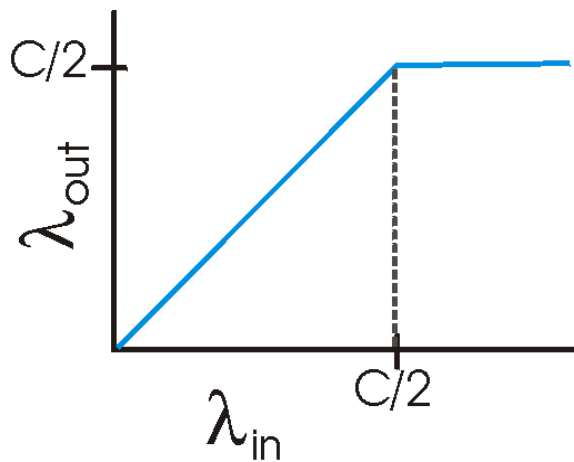
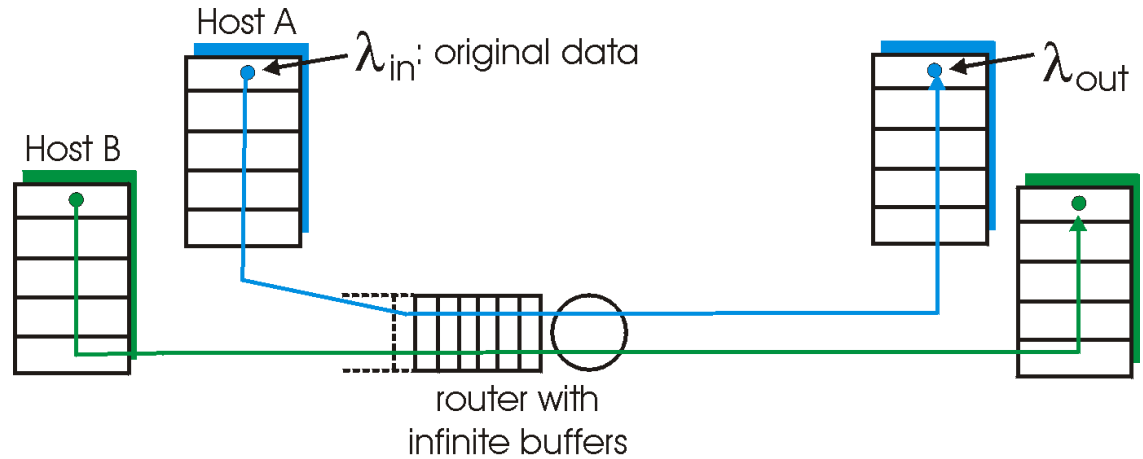


# Congestion Internet - Problématique

- Dans les années 80, avenir de TCP/IP en question
  - Trop de congestion

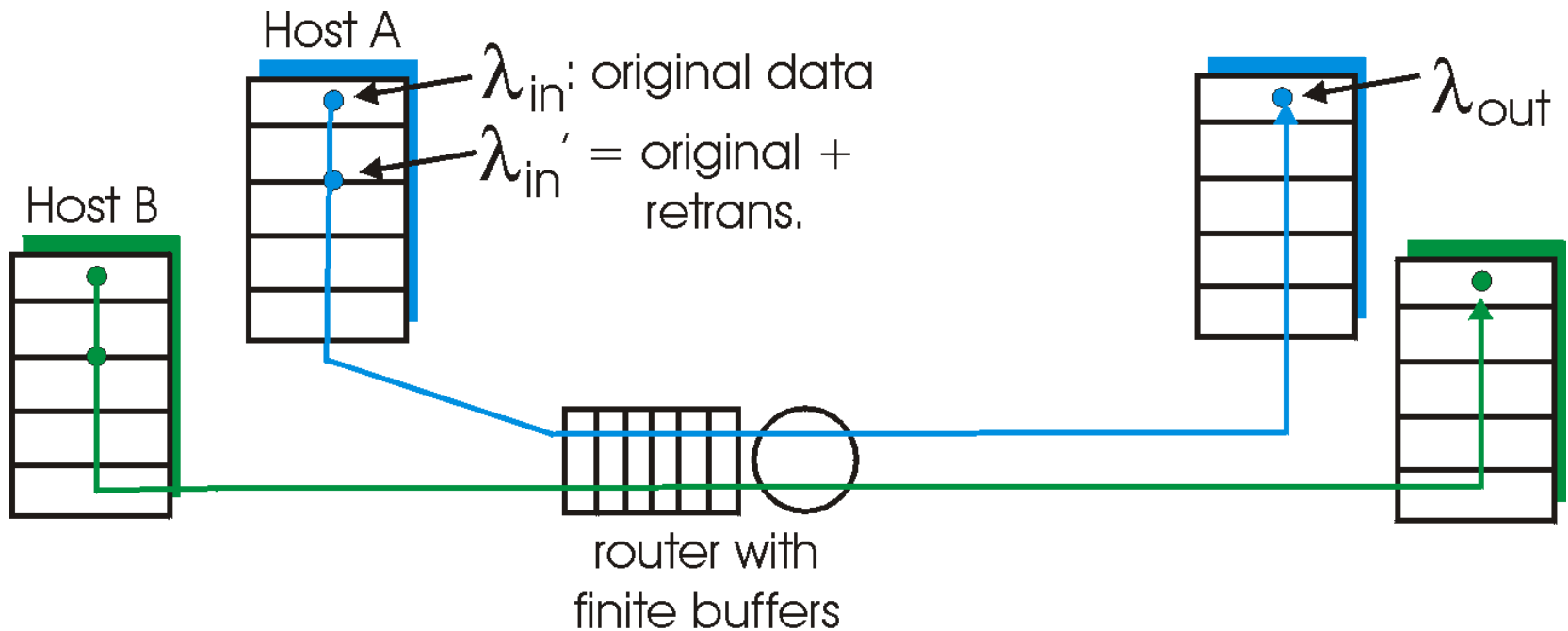


# Causes/coûts de la congestion : scénario 1



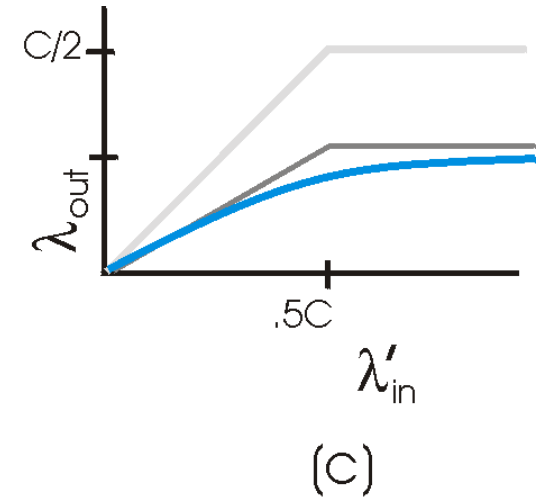
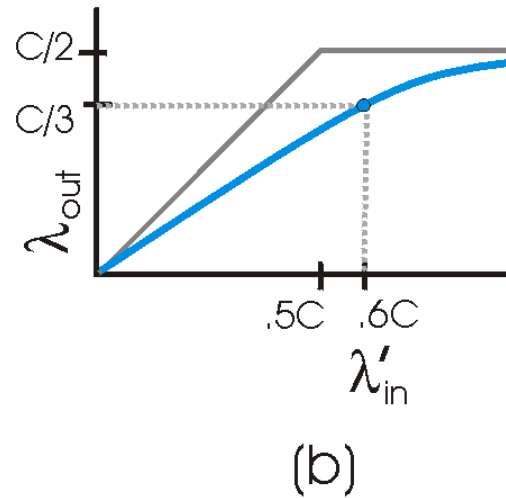
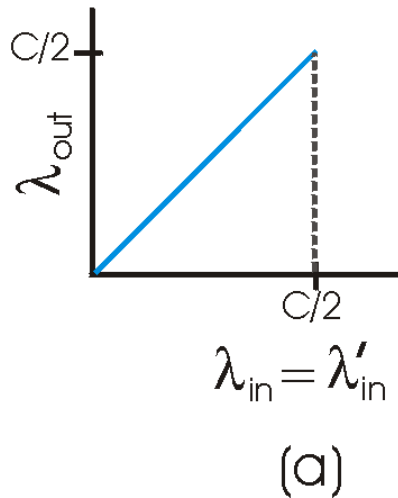
Kurose &amp; Ross

## Causes/coûts de la congestion : scénario 2

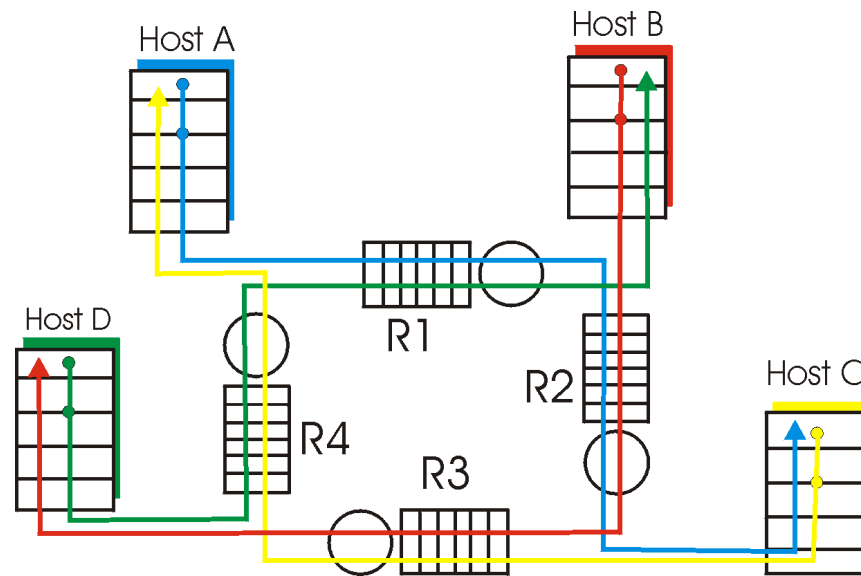
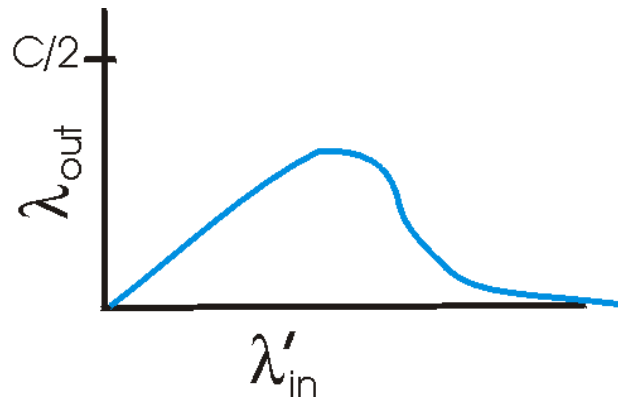


Kurose &amp; Ross

# Causes/coûts de la congestion : scénario 2



# Causes/coûts de la congestion: scenario 3

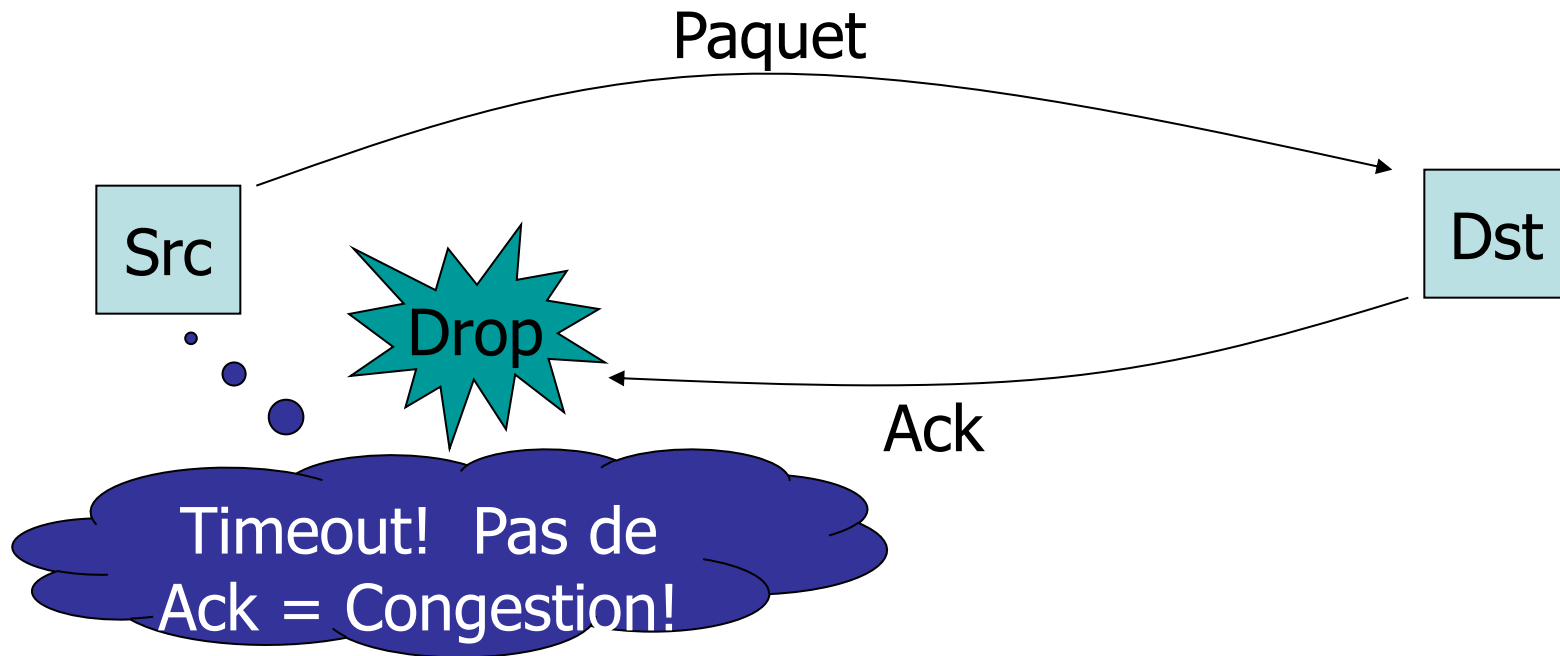




- Comment améliorer TCP afin qu'il envoie moins de trafic quand le réseau est congestionné ?
  - Comment détecter la congestion ?
  - Comment rendre le flux de données sensible au niveau de congestion ?

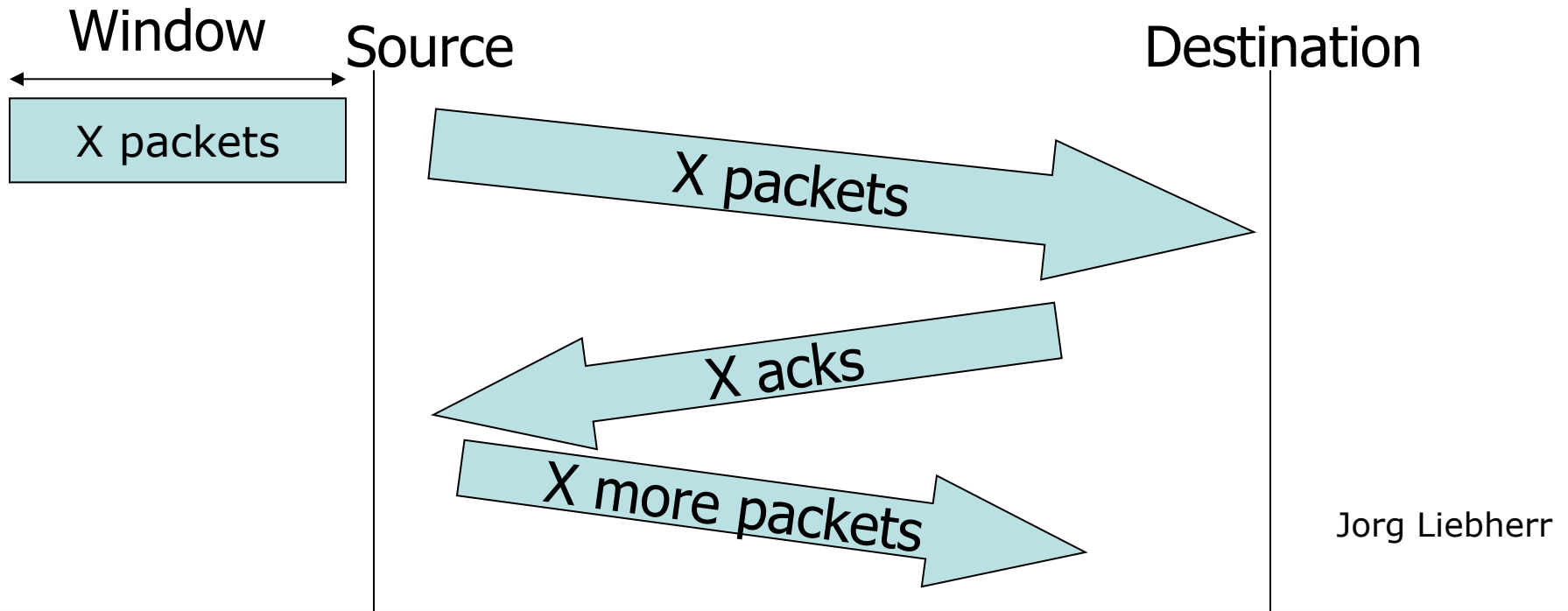
# Détection de la congestion

- Elimination de paquets = indicateur de congestion ?



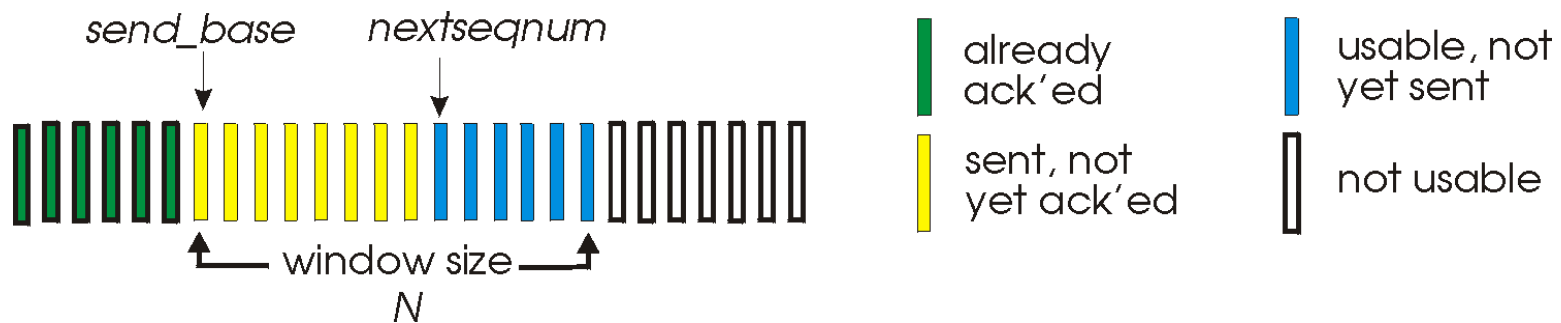
# Contrôle de Congestion – l'effet de la taille de la fenêtre

- Noter que la fenêtre de l'émetteur est égale au nombre de paquets en transit dans le réseau



# Contrôle de congestion TCP

- Contrôle de bout-en-bout (pas d'assistance du réseau)
- Débit de transmission limité par taille de fenêtre



Kurose &amp; Ross

# Contrôle de congestion TCP

- $w$  segments, chacun avec MSS octets émis en un RTT:

$$\text{throughput} = \frac{w * \text{MSS}}{\text{RTT}} \quad \text{Bytes/sec}$$

# Contrôle de congestion TCP

- “sonder” pour la bande passante utilisable:
  - idéalement: transmettre aussi vite que possible (fenêtre la + grande possible) sans perte
  - *augmenter* la fenêtre (→ plus de paquets dans le réseau ) jusqu’à l’apparition de pertes (congestion)
  - pertes: *diminuer* la fenêtre (→ moins de paquets dans le réseau) et recommencer à sonder (augmenter) à nouveau

# Contrôle de Congestion

- Idée: Concept de la *congestion window*
  - fenêtre qui est petite quand la congestion est importante
  - fenêtre qui est grande quand la congestion est petite
- Fenêtre autorisée = minimum (fenêtre récepteur, fenêtre de congestion)

# Contrôle de congestion de Van Jacobson

- Van Jacobson a introduit le contrôle de congestion dans TCP en 1988-1989
- La version d'origine de TCP qui implante le contrôle de congestion de Van Jacobson est connue sous le nom de TCP Tahoe



# Éléments de contrôle de congestion dans TCP Tahoe

- AIMD
  - Additive increase, multiplicative decrease
- Slow start
- Congestion Avoidance

# Additive Increase, Multiplicative Decrease

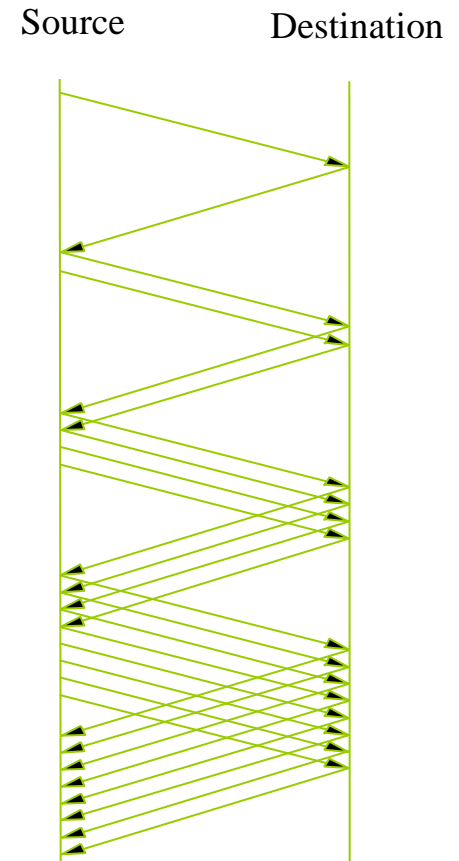
- Chaque fois qu'un paquet est détruit (perdu), réduire la taille de la fenêtre de manière significative (multiplicative decrease)
  - Multiplicative decrease est nécessaire pour éviter la congestion
- Si aucune perte n'est observée, augmenter de manière graduelle la taille de la fenêtre (additive increase)

# Problèmes

- Quelle doit être la taille de la fenêtre
  - Initialement ?
  - Quand il y a perte de paquet ?
- Taille de fenêtre pessimiste? (e.g., 1)
  - Additive increase est trop lent
  - Les connexions courtes n'utiliseront jamais la bande passante disponible
- Taille de fenêtre optimiste?
  - Un flux trop important de données peut impliquer un overflow dans les files des routeurs

# Slow Start

- Objectif: déterminer la capacité disponible rapidement
- Idée:
  - Utiliser **CongestionThreshold** comme une estimation optimiste de **CongestionWindow**
  - Commencer avec **CongestionWindow** = 1 segment
  - doubler **CongestionWindow** à chaque RTT (incrémenter de 1 paquet pour chaque ACK)
  - Quand **CongestionThreshold** est atteint utiliser additive increase

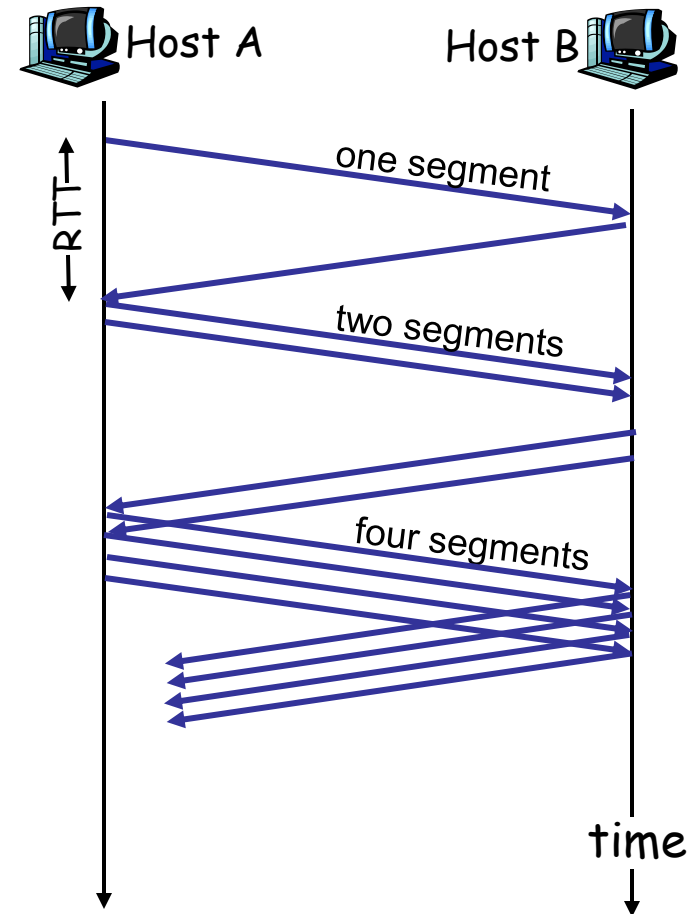


# Slow-start TCP

## Algorithme du slow start

```
initialise: Congwin = 1
Pour (chq segment ACKed)
    Congwin++
Jusqu' à (evt de perte OR
    CongWin >= threshold)
```

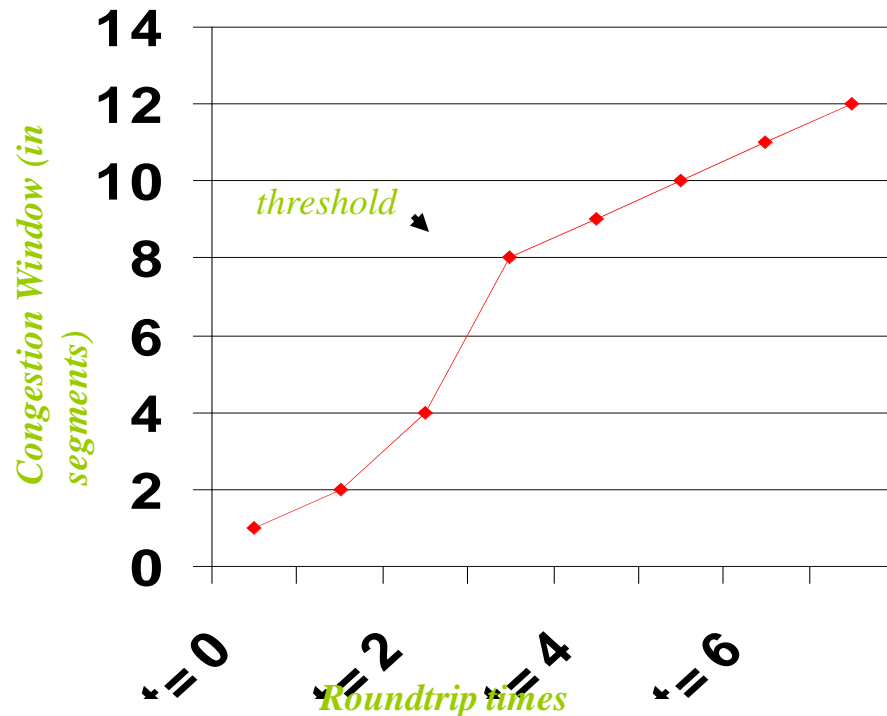
- Croissance exponentielle (par RTT) de la taille de fenêtre (pas si lent!)
- Evt de perte: timeout et/ou trois ACKs dupliqués (Tahoe TCP)



# Illustration du Slow Start

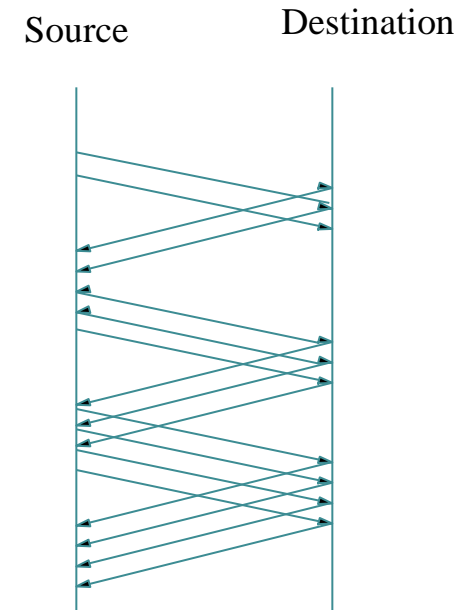
## Hypothèse

**CongestionThreshold = 8**



# AIMD + congestion avoidance

- Algorithme :
  - Incrémenter **CongestionWindow** d'un paquet par RTT (*linear increase*)
  - Réduire **CongestionThreshold** et **CongestionWindow** (*multiplicative decrease*) quand perte



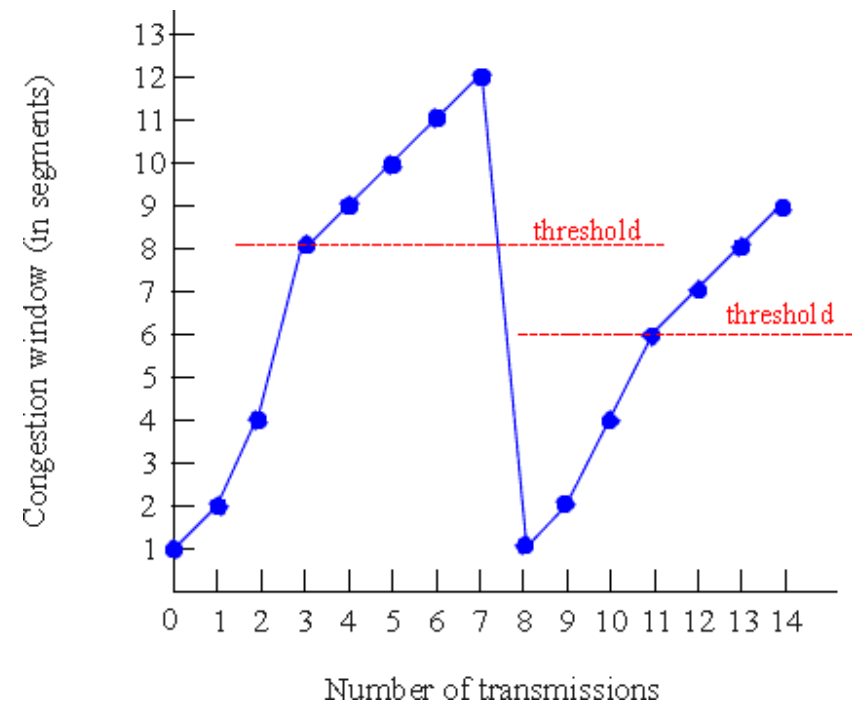
# Evitement de congestion TCP

## Congestion avoidance

### Evitement de congestion

```
/* slowstart est terminé */  
/* Congwin >= threshold */  
Until (evt de perte) {  
    chq w segments ACKed:  
        Congwin++  
}  
threshold = Congwin/2  
Congwin = 1  
effectuer slowstart1
```

1: TCP Reno évite slowstart (fast recovery) après 3 ACKs dupliqués



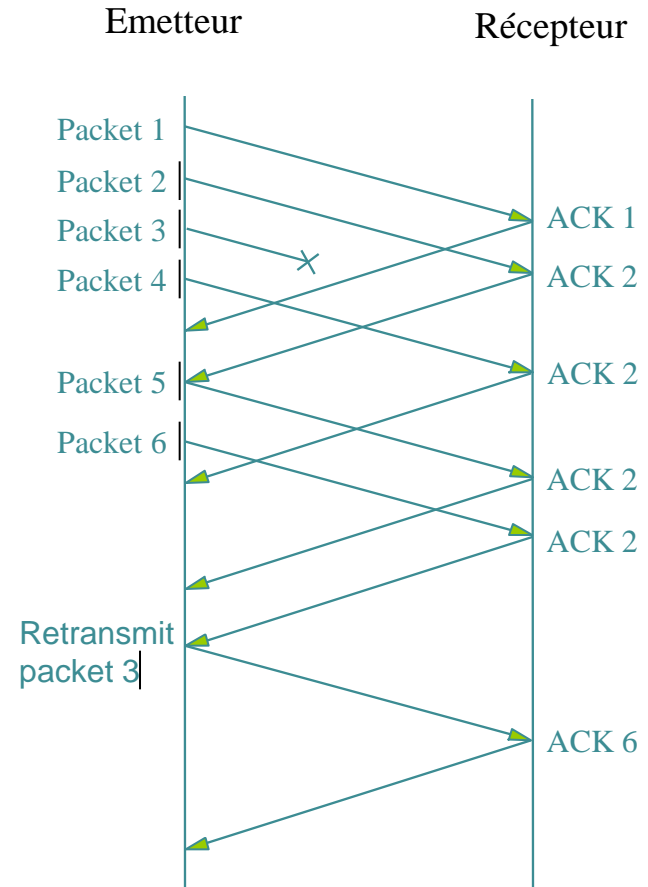


# TCP Reno

- TCP Reno (1990) et ses dérivés sont les plus utilisés aujourd'hui
- TCP Reno implémente :
  - Fast Retransmit
  - Fast Recovery
- RFC 2581

# Fast Retransmit

- Problème : les timers TCP peuvent conduire à des périodes d'inactivité



# Fast Retransmit

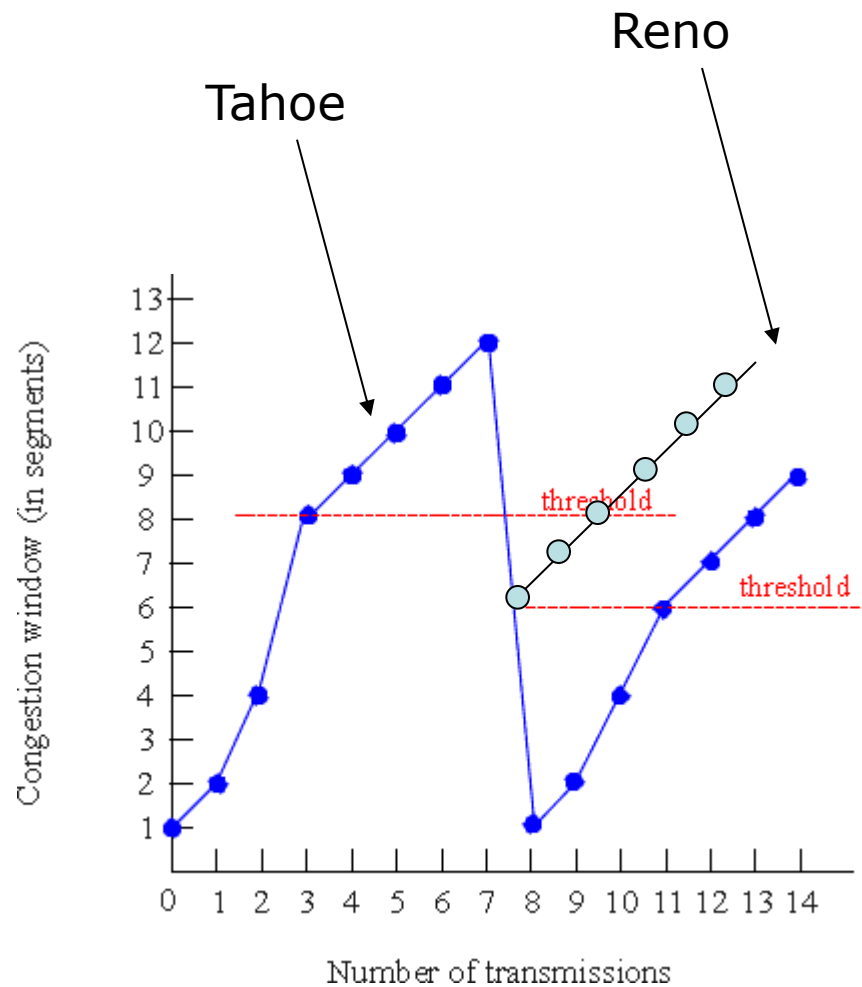
- Envoi d'un acquittement sur réception d'un paquet
- Envoi d'une duplication du dernier acquittement quand un paquet est reçu en désordre
- Utilisation des acquittements dupliqués pour déclencher des retransmissions
  - Si 3 (ou plus) d'acquittements dupliqués reçus, le segment est considéré comme perdu
  - TCP réalise une retransmission du segment manquant sans attendre l'expiration du timer

# Fast Recovery

- Fast Recovery est implémenté avec Fast Retransmit
- Fast Recovery supprime la phase de slow start après retransmission due à un Fast Retransmit
- Fast Recovery gère la transmission des données jusqu'à l'arrivée d'un ACK non dupliqué

# Fast Recovery

- Algorithme :
  - Quand le 3ème ACK dupliqué est reçu,
    - positionner le seuil (sssthresh) à la moitié de la congestion window (cwnd) courante
    - retransmettre le segment manquant
    - Positionner cwnd à la valeur du sssthresh + 3\*MSS
  - Chaque fois qu'un autre ACK dupliqué arrive
    - Incrémenter cwnd
    - Transmettre un paquet si possible
  - Quand ACK arrive qui acquitte la nouvelle donnée
    - Positionner cwnd à la valeur de sssthresh



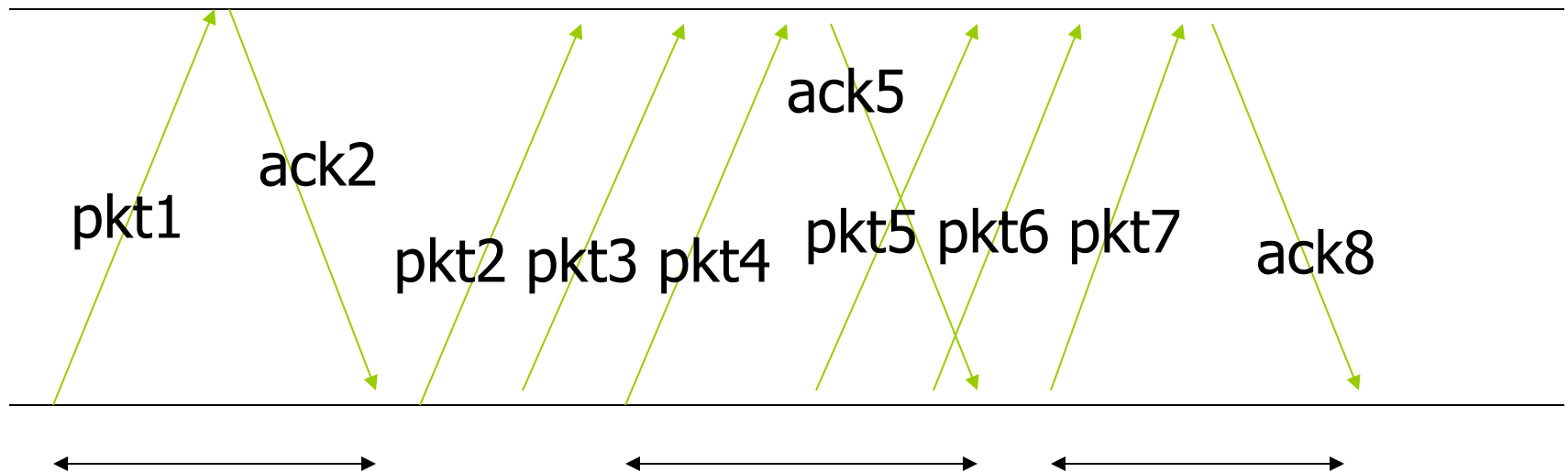
# Résumé Congestion Control

- Essayer d'éviter que la source applique le slow-start
  - Source utilise slow-start au début et sur timeout uniquement
  - Source divise la fenêtre de congestion window de la moitié suite à un fast retransmit
- La perte d'un seul paquet peut être détectée par les mécanismes de fast retransmit/fast recovery algorithm
- La perte de plusieurs paquets consécutifs forcera la source à appliquer le slow start
  - TCP New Reno (RFC3782)

# Tahoe/Reno

## Timers de retransmission

- Les timers de retransmission sont basés sur les calculs du round-trip time (RTT) que TCP effectue





# TCP Round Trip Time et Timeout

- Comment positionner le Timeout TCP ?
- Plus long que RTT
  - note: RTT va varier
- Trop court: timeout prématuré
  - Retransmissions inutiles
- Trop long: réaction lente à la perte de segments

# TCP Round Trip Time et Timeout

Comment estimer RTT ?

- **SampleRTT**: temps mesuré entre l'envoi d'un segment et la réception d'un ACK
- **SampleRTT** va varier, besoin d'un RTT moyen, plus lissé
  - Utilisation de plusieurs mesures récentes, pas uniquement le **SampleRTT courant**

# TCP Round Trip Time et Timeout

- Moyenne fluctuante
- L'influence d'un échantillon diminue de façon exponentielle
- Valeur typique de  $\alpha$ : 0.125 (1/8)
- Extrait du rfc 2988

$$\text{EstimatedRTT} = (1-\alpha).\text{EstimatedRTT} + \alpha.\text{SampleRTT}$$

# TCP Round Trip Time et Timeout

- Valeur du timeout
  - **EstimatedRTT** plus “marge de sécurité”
  - large variation dans **EstimatedRTT** → marge de sécurité accrue

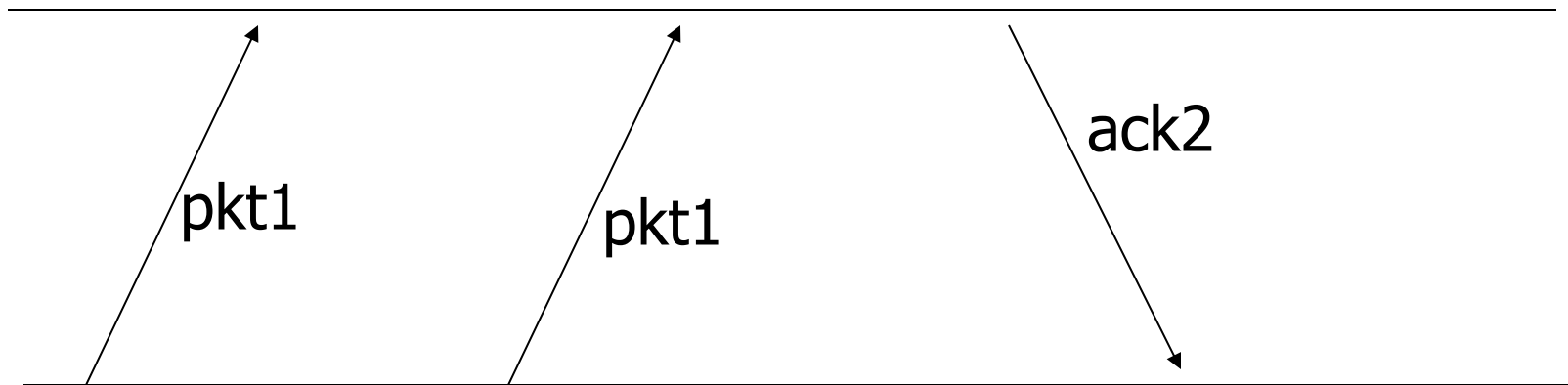
**RetransmissionTimeout ou RTO = EstimatedRTT + 4\*Deviation**

$$\text{Deviation} = (1-\beta).\text{Deviation} + \beta.|\text{SampleRTT}-\text{EstimatedRTT}|$$

Avec  $\beta = 0.25$

# Problème

- Si un segment est retransmis, TCP ne peut pas calculer le RTT
  - Pourquoi ?



# Algorithme de Karn

- Ne pas mettre à jour le RTT sur des segments qui ont été retransmis.
- Quand TCP retransmet, positionner :  
 **$RTO = \min (2 RTO, 64)$**

# Stratégies de contrôle de congestion

- Stratégie TCP
  - Contrôle de congestion une fois détectée
  - De manière répétitive, augmente la charge pour trouver le point de congestion puis diminution de la charge quand congestion
- Stratégie alternative
  - Prédire quand la congestion va se produire
  - Réduire le débit avant que le paquet soit détruit
  - Deux possibilités :
    - Orientée routeur : mécanisme RED
    - Orientée hôte : TCP Vegas

# TCP Vegas

- **Vegas:** Prédit et évite la congestion avant qu'elle se produise
- **Tahoe, Reno:** Réagit à la congestion une fois qu'elle s'est produite
- Question: comment prédire la congestion ?

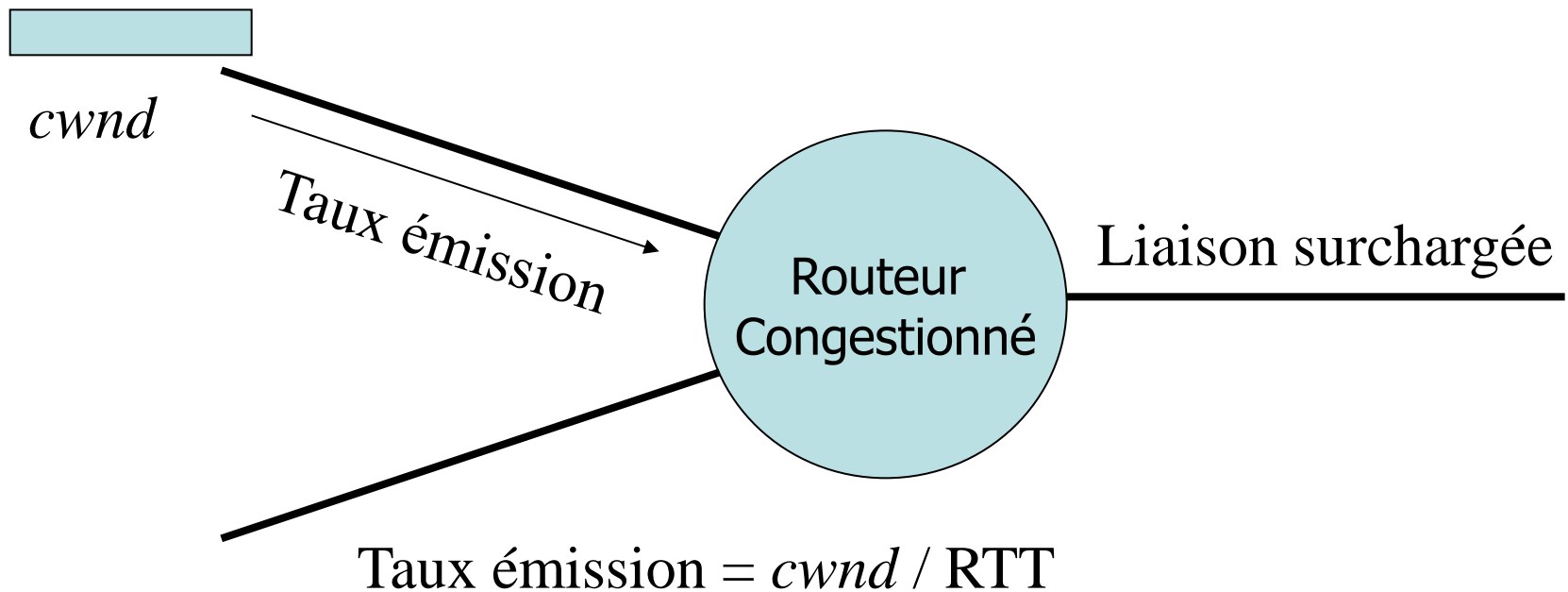


# Principe

- L. Brakmo, S.O'Malley et L. Peterson (SIGCOMM'94)
- Idée :
  - La source observe si des signes indiquent que les files des routeurs se remplissent et qu'une congestion risque de se produire
    - RTT augmente
    - Le débit d'émission fluctue

# Observation

- L'accumulation de paquets dans le réseau peut se déduire en observant le RTT et débit en émission



# Algorithme

- Soit **BaseRTT** le minimum de tous les RTT mesurés (généralement RTT du premier paquet)

$$\text{ExpectRate} = \text{CongestionWindow} / \text{BaseRTT}$$

- Source calcule **ActualRate** une fois par RTT

# Algorithme

- Source compare **ActualRate** avec **ExpectRate**

**Diff = ExpectedRate - ActualRate**

**if Diff <  $a$**

**augmente CongestionWindow linéairement**

**else if Diff >  $b$**

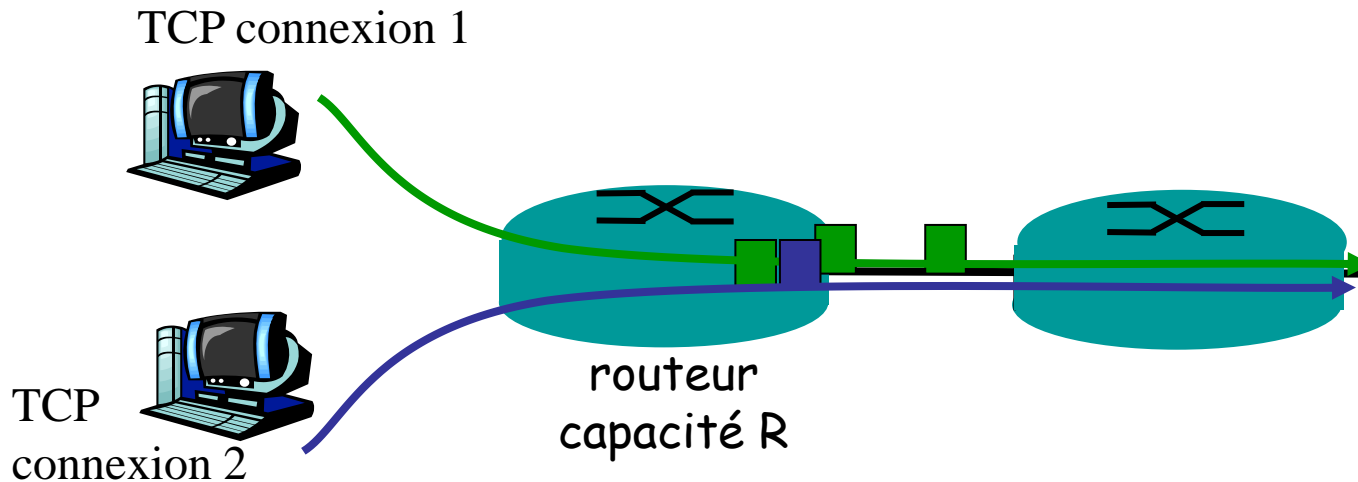
**décroître CongestionWindow linéairement**

**else**

**laisser CongestionWindow inchangée**

# Equité TCP

Si  $N$  sessions TCP partagent le même lien goulot d'étranglement, chacun doit avoir  $1/N$  de la capacité du lien

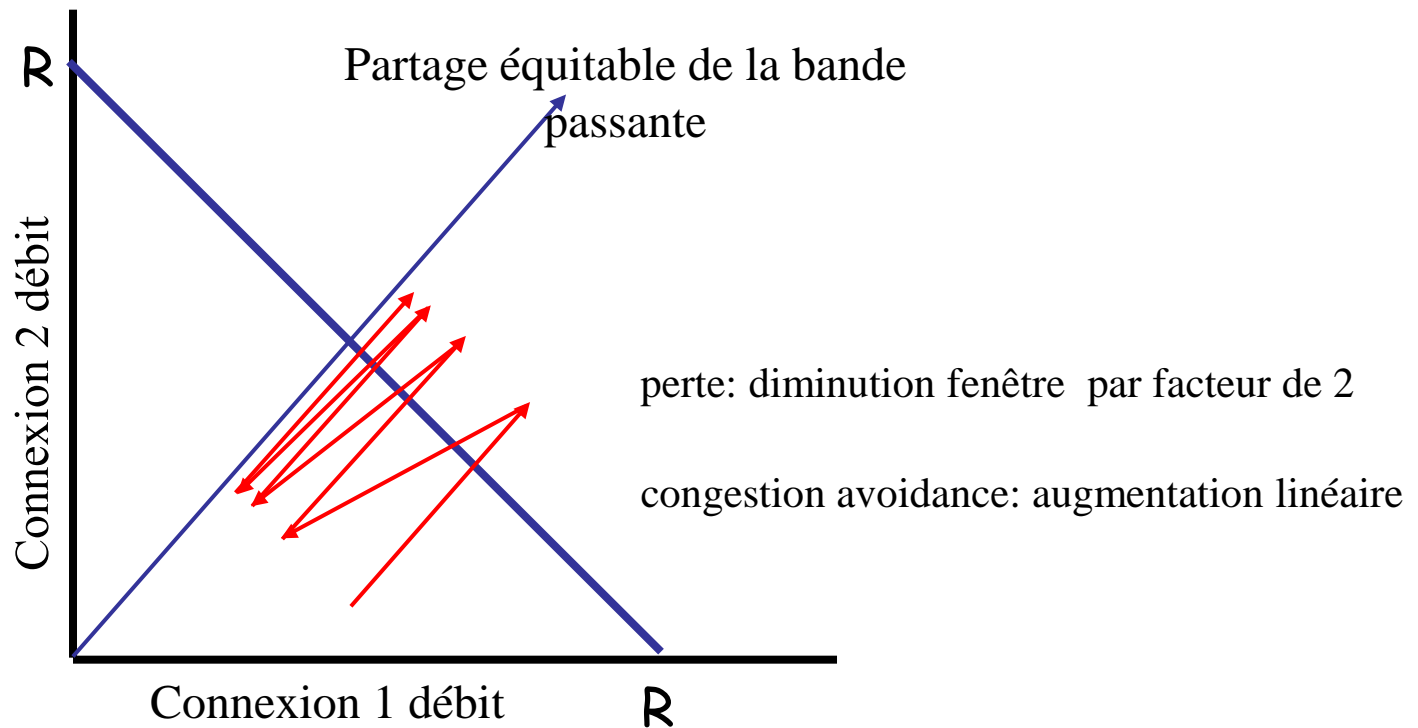


# Pourquoi TCP est équitable ?

Deux sessions en compétition:

- Augmentation additive donne une pente de 1, lorsque le débit augmente
- Diminution multiplicative diminue le débit proportionnellement

# Pourquoi TCP est équitable ?



Kurose &amp; Ross

# Plan

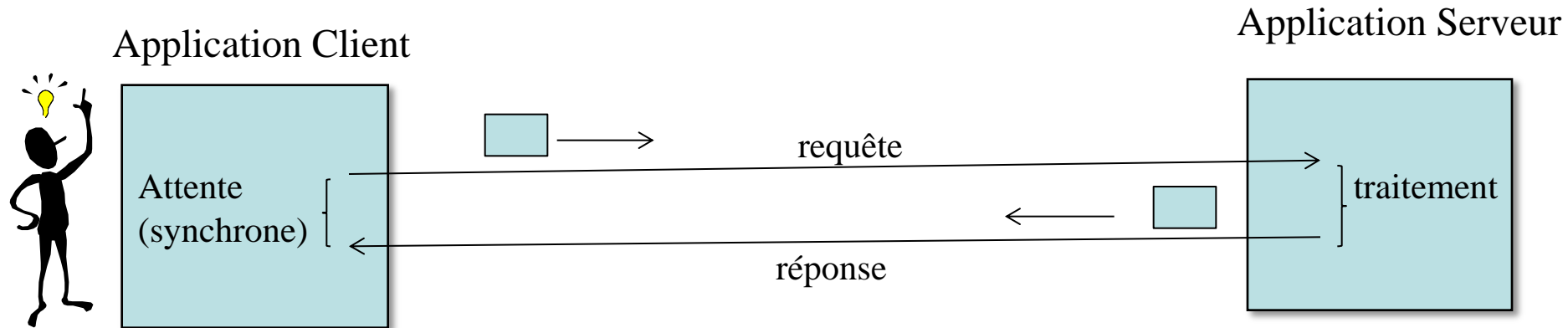
- ① Contrôle de congestion et TCP
- ② Programmation réseaux
  - socket, select(), options de socket, socket raw
- ③ Communication de groupe (multicast)
- ④ Introduction IPv6



## ② Programmation réseaux

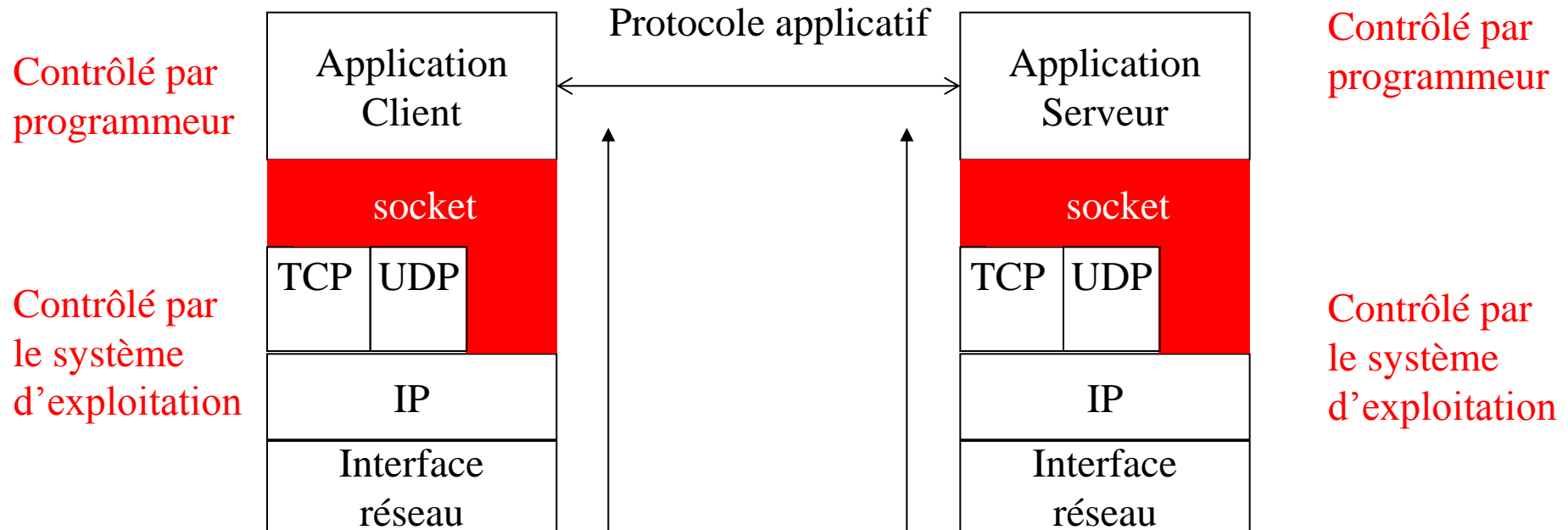
- **Modèle Client Serveur**
- Introduction socket
- Client-Serveur TCP
- Client-Serveur UDP
- Entrée/sortie multiplexée : select
- Entrée/sortie asynchrone
- Options de socket
- Socket raw

# Le modèle Client Serveur



- Modèle asymétrique :
  - Le client :
    - demande l'exécution d'un service
    - est bloqué en attente de la réponse (mode synchrone)
  - Le serveur :
    - est en attente de requêtes (processus en arrière plan)
    - réalise le service demandé d'un client et envoie la réponse (traitement séquentiel ou concurrent)
    - peut gérer plusieurs clients en même temps (serveur multi-clients)

# Le modèle Client Serveur

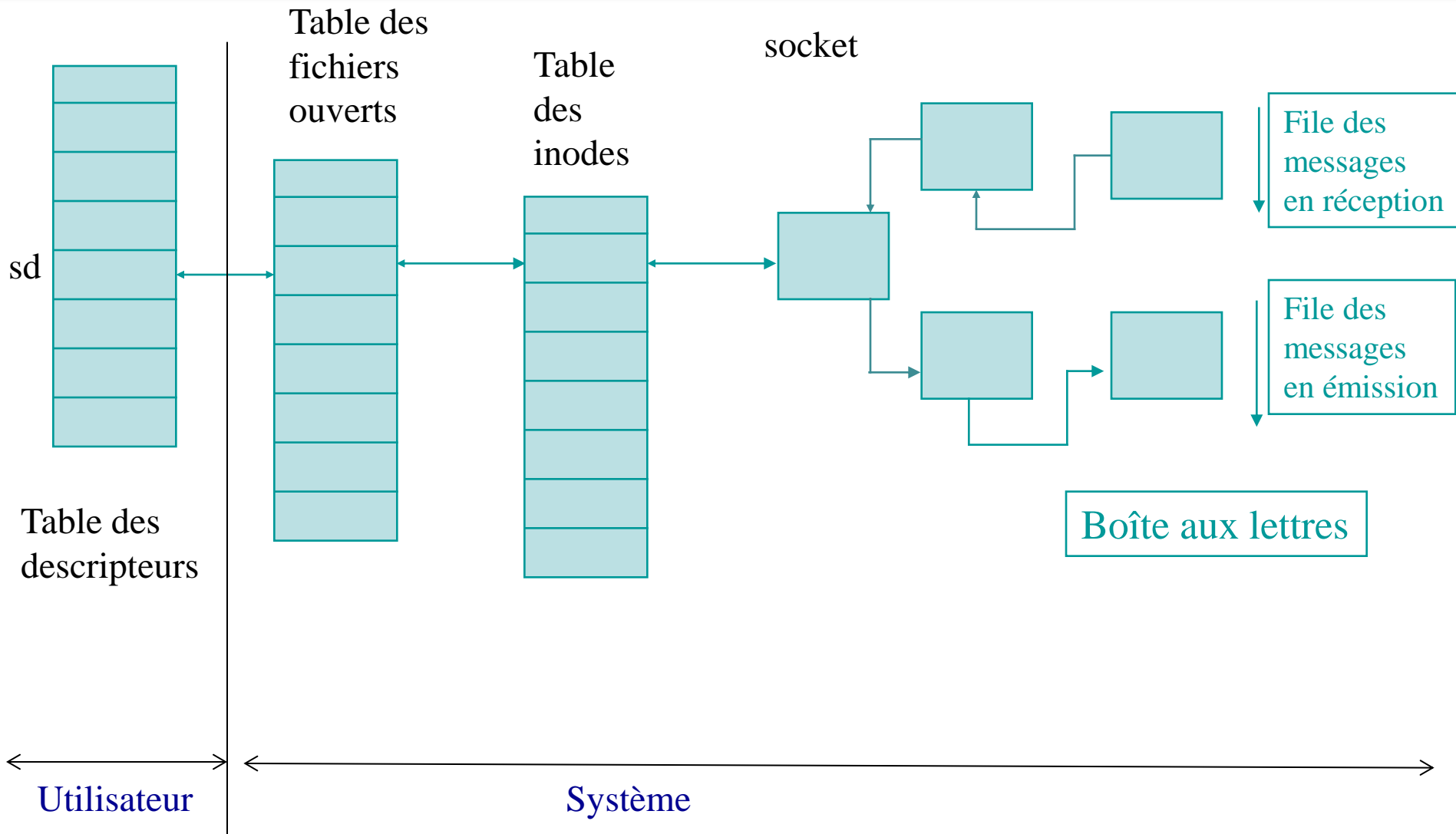


## ② Programmation réseaux

- Modèle client serveur
- Introduction socket
- Client-Serveur TCP
- Client-Serveur UDP
- Entrée/sortie multiplexée : select
- Entrée/sortie asynchrone
- Options de socket
- Socket raw

# Définition d'un(e) socket

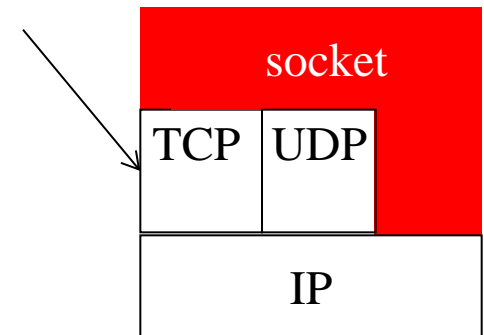
- Interface de programmation (API) permettant d'envoyer et recevoir des données
- Point de communication bidirectionnel
  - à l'intérieur d'une même machine → domaine Unix
  - à travers les réseaux → domaine Internet
- Initialement liée à la version BSD (Berkeley) de Unix
- Descripteur de socket → similaire descripteur fichier (ressources système)
- Accessible par des appels systèmes



# Les types de sockets

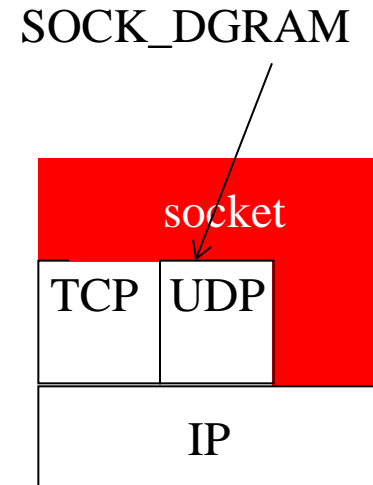
- Précise le type de communication utilisé
  - SOCK\_STREAM
    - transmission fiable, ordonnée, sans duplication
    - en mode connecté
    - autorise messages urgents
    - pas de préservation des limites de messages
    - TCP
  - SOCK\_DGRAM
  - SOCK\_RAW

SOCK\_STREAM



# Les types de sockets

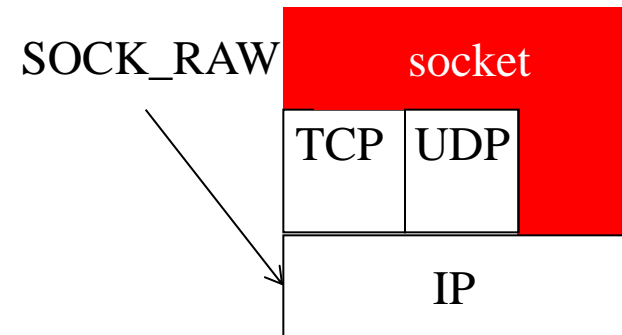
- Précise le type de communication utilisé
  - SOCK\_STREAM
  - SOCK\_DGRAM
    - transmission non fiable, non ordonnée, avec possibilité de duplication
    - en mode non connecté
    - préservation des limites de messages
    - UDP
  - SOCK\_RAW





# Les types de sockets

- Précise le type de communication utilisé
  - SOCK\_STREAM
  - SOCK\_DGRAM
  - SOCK\_RAW
    - protocoles de bas niveau (super-user)
    - permet de ne pas passer par la couche transport et d'utiliser IP directement
    - ICMP



# Les familles de sockets

- Une famille ou un domaine définit :
  - La convention d'adressage ou format des adresses à utiliser
  - Le ou les protocoles réseaux supportés
- AF\_UNIX (AF\_LOCAL)
  - Local à la machine : domaine UNIX
- AF\_INET
  - Domaine Internet et adresses IPv4
- AF\_INET6
  - Domaine Internet et adresses IPv6

# Structure générique d'une socket

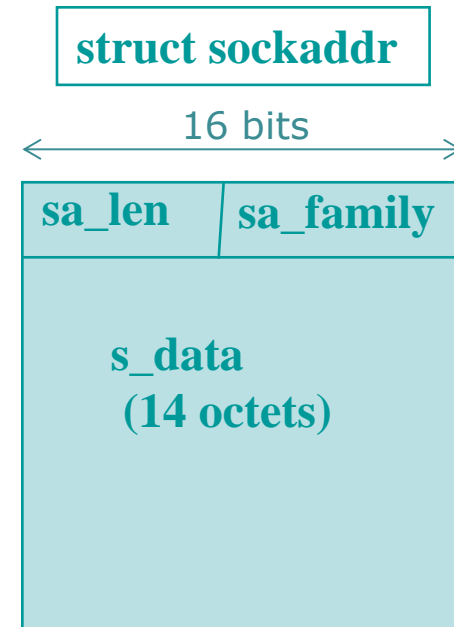
Format général décrit dans  
<sys/socket.h>

```
struct sockaddr {
    uint8_t sa_len;
    sa_family_t sa_family;
    char s_data[14];
}
```

**sa\_len** → longueur de la structure

**sa\_family** → AF\_UNIX ou  
AF\_INET,....

**s\_data** → jusqu'à 14 octets pour  
décrire l'adresse spécifique au  
protocole



# Structure spécifique d'une socket

- Format spécifique au protocole de communication

```
struct sockaddr_xx {  
    }
```

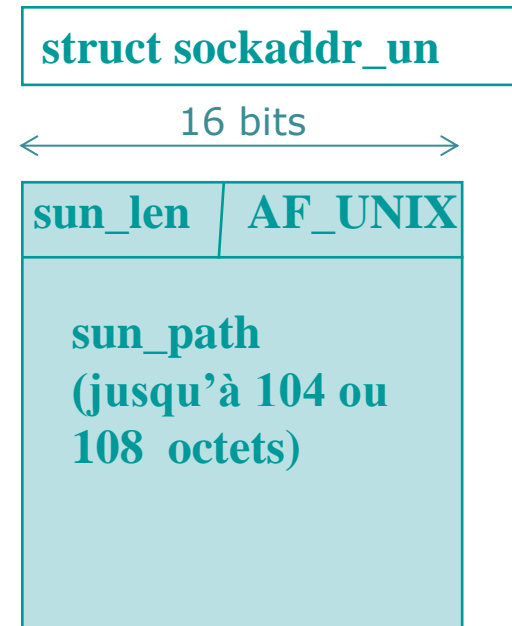
- xx est un suffixe unique fonction du protocole  
xx = **in** pour Internet (IPv4)  
**in6** pour Internet (IPv6)  
**un** pour Unix

# Structure dans le domaine UNIX

Format décrit dans `<sys/un.h>`

```
struct sockaddr_un {
    uint8_t sun_len;
    sa_family_t sun_family;
    char sun_path[104];
}

sun_family → AF_UNIX
sun_path → /* chemin = nom de fichier */
```



# Structure dans le domaine INET

Format décrit dans `<netinet/in.h>`

```
struct sockaddr_in {
    uint8_t sin_len;
    sa_family_t sin_family;
    in_port_t sin_port;
    struct in_addr sin_addr;
    char sin_zero[8];
}
```

```
struct in_addr {
    in_addr_t s_addr;
};
```

`typedef uint32_t in_addr_t;`

**sin\_family** → AF\_INET

**sin\_port** → numéro sur 16 bits dans l'ordre du réseau (network byte ordered)

**sin\_addr** → adresse Internet sur 32 bits dans l'ordre du réseau

**sin\_zero** → inutilisé (doit être mis à zéro)

**struct sockaddr\_in**

16 bits

sin_len	AF_INET
#port	
@IPv4	
inutilisé	

# Structure dans le domaine INET

## Numéro de port

- Identifie de manière unique, sur une machine, un processus exécutant un service donné
  - Il existe des ports prédéfinis (cf `/etc/services`)
    - serveur web : port 80
    - serveur mail : port 25
    - appel de la fonction **getservbyname()**
  - Le port peut être choisi par le programmeur d 'application  
 $\geq$  `IPPORT_RESERVED` (1024)
  - Le port peut être attribué par le système (**sin\_port**  $\leftarrow$  0)

# Structure dans le domaine INET

## Numéro de port – fonction getservbyname()

```
#include <netdb.h>
```

```
struct servent * getservbyname (const char *name, const char  
    *proto)
```

**name** → pointeur sur la chaîne décrivant le nom du service

**proto** → pointeur sur la chaîne où se trouve le nom du protocole (tcp)

**valeur de retour** -1 si erreur

```
struct servent{  
    char *s_name; /* nom du service*/  
    char **s_aliases; /* liste alias */  
    int s_port; /* numéro port (ordre réseau)*/  
    int s_proto ; /* protocole utilisé */  
}
```



# Structure dans le domaine INET

## Adresse IP

- Identifie de manière unique une machine sur un réseau
  - soit obtenue par les fonctions :
    - `gethostbyname()`, `gethostbyname2()` `getipnodebyname()`, `getaddrinfo()`
  - dans le cas d'un programme serveur la valeur `INADDR_ANY` est choisie : les messages peuvent être reçus sur n'importe quelle interface réseau
- Exprimée sur la forme `ddd.ddd.ddd.ddd`
  - exemple : `193.54.8.4`
  - Dans la structure ➔ entier format réseau

# Structure dans le domaine INET

## Adresse IP – fonction gethostbyname()

```
#include <netdb.h>
```

```
struct hostent *gethostbyname (const char *name)
```

**name** ➔ pointeur sur la chaîne où se trouve le nom du système

**valeur de retour** -1 si erreur

```
struct hostent{  
    char *h_name; /* nom machine */  
    char **h_aliases; /* liste alias */  
    int h_addrtype; /* type famille */  
    int h_length ; /* longueur adresse */  
    char **h_addr_list; /* liste adresses */  
#define h_addr h_addr_list[0]  
}
```

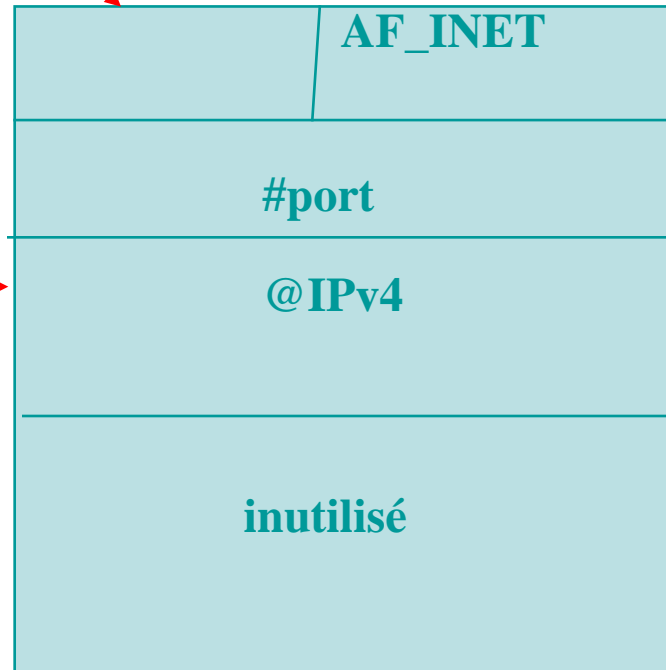
# struct sockaddr\_in

sin\_len (1 octet)

gethostbyname()  
->h\_addrtype  
sin\_family (1 octet)

getservbyname()  
>s\_port  
sin\_port (2 octets)

bzero() ou memset()  
sin\_zero (8 octets)



peut ne pas être  
renseigné  
sizeof (struct  
sockaddr\_in)

gethostbyname()  
->h\_addr  
sin\_addr(4 octets)  
adresse Internet (IP)

# Structure dans le domaine INET

## Format de représentation des données

- Les machines hôtes stockent les données (entiers) dans un format
  - Little endian
  - Big endian
- Pour communiquer il faut un format de transfert des données
  - *network byte order* : **big endian**

# Structure dans le domaine INET

## Fonctions d'ordonnancement des octets (Byte Ordering Functions)

```
#include <arpa/inet.h>
```

```
uint16_t htons (uint16_t host16bitvalue);
```

```
uint32_t htonl (uint32_t host32bitvalue);
```

retournent une valeur dans l'ordre du réseau

```
uint16_t ntohs (uint16_t net16bitvalue);
```

```
uint32_t ntohl (uint32_t net32bitvalue);
```

retournent une valeur dans l'ordre de la machine hôte

# Structure dans le domaine INET

## Fonctions de manipulation d'octets

```
#include <strings.h>
```

```
void bzero (void *dest, size_t nbytes) ;
```

```
void bcopy (const void *src, void * dest, size_t nbytes) ;
```

```
int bcmp (const void *ptr1, const void *ptr2, size_t nbytes) ;
```

retourne 0 si égal

Fonctions  
obsolètes

```
#include <string.h>
```

```
void *memset (void *dest, int c, size_t len) ;
```

```
void *memcpy (void *dest, const void *src, size_t nbytes) ;
```

```
int memcmp (const void *ptr1, const void *ptr2, size_t nbytes) ;
```

retourne 0 si égal

# Structure dans le domaine INET

## Fonctions de manipulation d'adresses

```
#include <arpa/inet.h>
```

```
int inet_aton (const char *strptr, struct in_addr *addrptr) ;
```

*retourne 1 si string valide 0 sinon*

```
in_addr_t inet_addr (const char *srcptr) ;
```

*retourne une valeur de 32 bits ordonnée réseau sinon INADDR\_NONE*

```
char *inet_ntoa (struct in_addr inaddr) ;
```

*retourne un pointeur vers une chaîne contenant une adresse IP « aaa.bbb.ccc.ddd »*

```
int inet_pton(af, src, dst)
```

```
int af;          /* AF_INET ou AF_INET6 */
```

```
const char *src; /* l'adresse (chaîne de caract.) à traiter */
```

```
void *dst;       /* le tampon où est rangé le résultat */
```

```
const char * inet_ntop(af, src, dst, size)
```

```
int af;          /* AF_INET ou AF_INET6 */
```

```
const void *src; /* l'adresse binaire à traiter */
```

```
char *dst;       /* le tampon où est rangé le résultat */
```

```
socklen_t size;  /* la taille de ce tampon */
```

```

struct sockaddr_in serv_addr;
Struct hostent *hp;

{
bzero( (&serv_addr, sizeof(serv_addr) );
memset (&serv_addr, 0, sizeof(serv_addr) );
serv_addr.sin_port = htons ((ushort) atoi(argv[2]));
hp = (struct hostent *)gethostbyname (argv[1]);
if (hp == NULL) {
    fprintf(stderr, "%s: %s non trouve dans in /etc/hosts ou dans le DNS\n",
        argv[0], argv[1]);
    exit(1);
}
serv_addr.sin_family = hp->h_addrtype ; /* AF_INET */
serv_addr.sin_addr = * ((struct in_addr *) (hp->h_addr));
printf ("IP address: %s\n", inet_ntoa (serv_addr.sin_addr));

```



## ② Programmation réseaux

- Modèle client serveur
- Introduction socket
- **Client-Serveur TCP**
- Client-Serveur UDP
- Entrée/sortie multiplexée : select
- Entrée/sortie asynchrone
- Options de socket
- Socket raw

# Dialogue Client-Serveur TCP

Création de socket

Ouverture de dialogue

Echange de données

Fermeture de socket

# Type de Dialogue TCP/IP

- transmission fiable , ordonnée, sans duplication
- en mode connecté
- autorise messages urgents
- pas de préservations de limites de messages

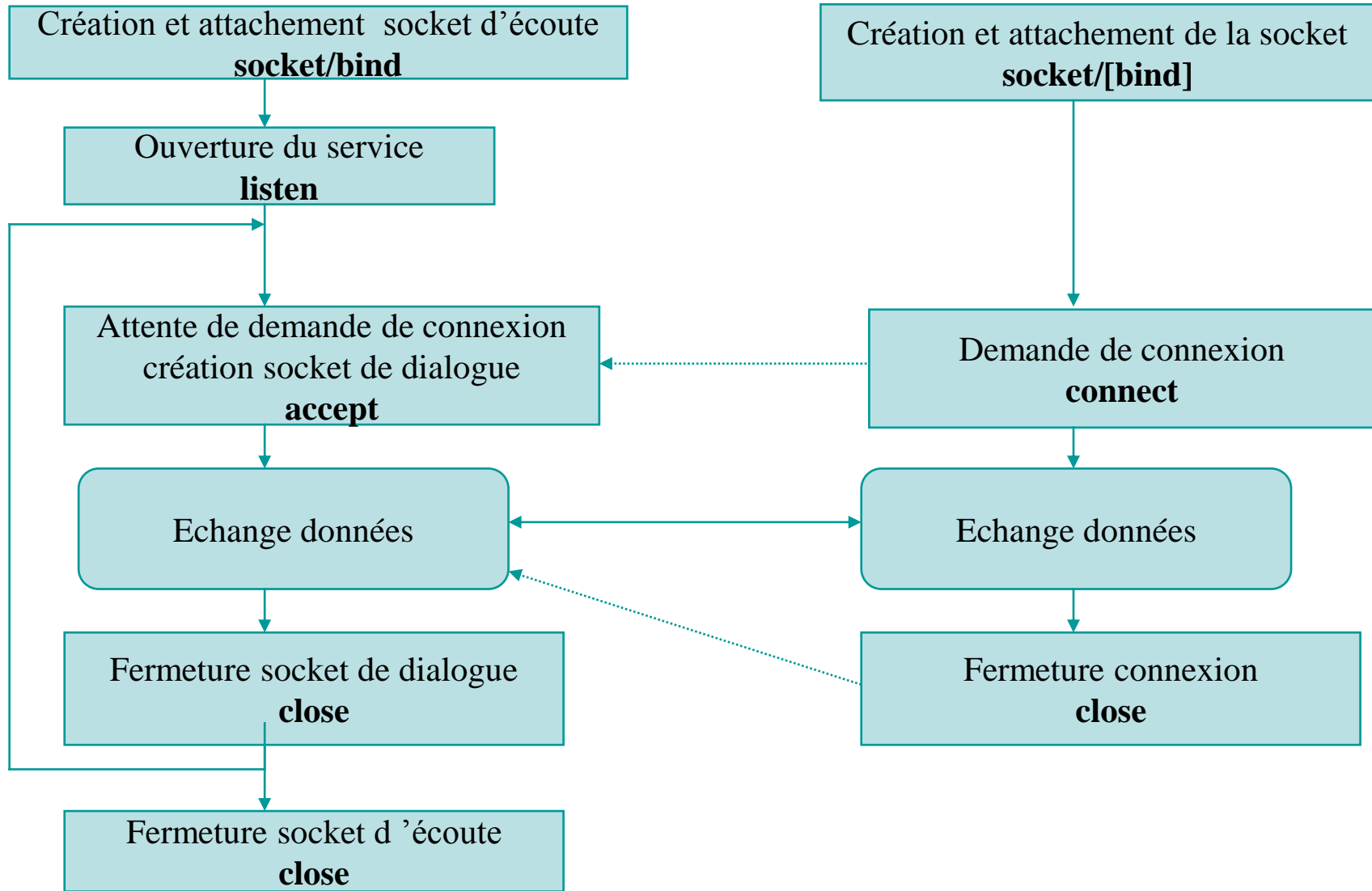


Mode de communication TCP/IP (SOCK\_STREAM)

Transmission Control Protocol/Internet Protocol

## Serveur

## Client



# Création d'une socket (C/S)

```
#include <sys/types.h>
#include <sys/socket.h>
int socket (int domain, int type, int protocol)
```

**domain** → [AF\_UNIX ou AF\_INET]  
→ PF\_UNIX ou PF\_INET

**type** → SOCK\_STREAM

**protocol** → 0 par défaut

**valeur de retour** → -1 si erreur sinon  
descripteur de socket

```
int serverSocket;
/*
 * Ouvrir socket (socket STREAM)
 */
if ((serverSocket = socket(PF_INET,
SOCK_STREAM, 0)) < 0) {
    perror ("erreur socket");
    exit (1);
}
```

# Attachement/nommage d'une socket (C/S)

- Association entre une socket et une adresse

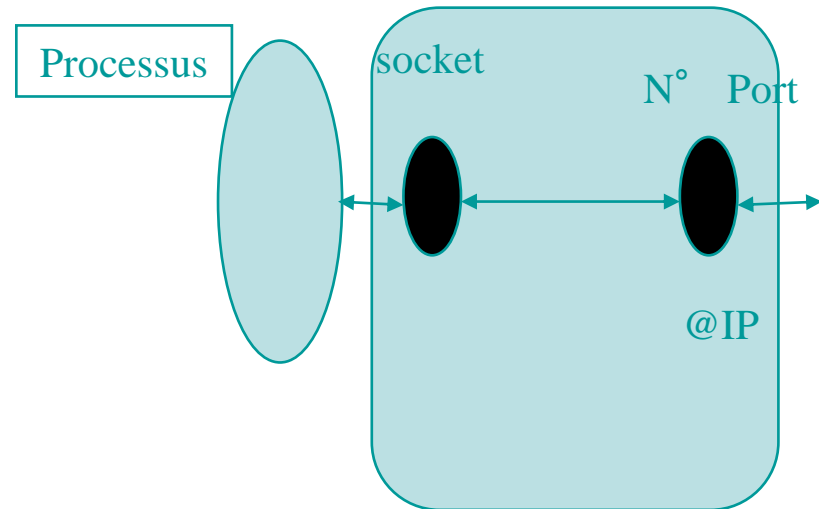
```
#include <sys/types.h>
#include <sys/socket.h>
int bind (int s, const struct sockaddr
*name, socklen_t addrlen)
```

**s** → descripteur retourné par socket()

**name** → pointeur sur une structure contenant l'adresse Internet

**addrlen** → longueur de l'adresse

**valeur de retour** → -1 si erreur



Bind : ceci est mon adresse et tous les messages reçus sur cette adresse doivent m'être délivrés

Si la machine a plusieurs adresses IP, lors du bind on pourra utiliser INADDR\_ANY

```
#define SERV_PORT 2222
struct sockaddr_in serv_addr;

/*
 * Lier l'adresse locale à la socket
 */;
memset (&serv_addr, 0, sizeof(serv_addr) );
serv_addr.sin_family = AF_INET ;
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
serv_addr.sin_port = htons(SERV_PORT);

if (bind(serverSocket, (struct sockaddr *)&serv_addr, sizeof (serv_addr) ) < 0) {
    perror ("servecho: erreur bind\n");
    exit (1);
}
```

# Ouverture du service (S)

- Utilisé par le **serveur** en mode connecté pour indiquer qu'il est prêt à recevoir des demandes de connexion
- Après cet appel, seuls des paquets d'ouverture de connexion seront reçus sur ce socket

```
#include <sys/socket.h>
```

```
int listen (int s, int backlog)
```

**s** → descripteur retourné par socket()

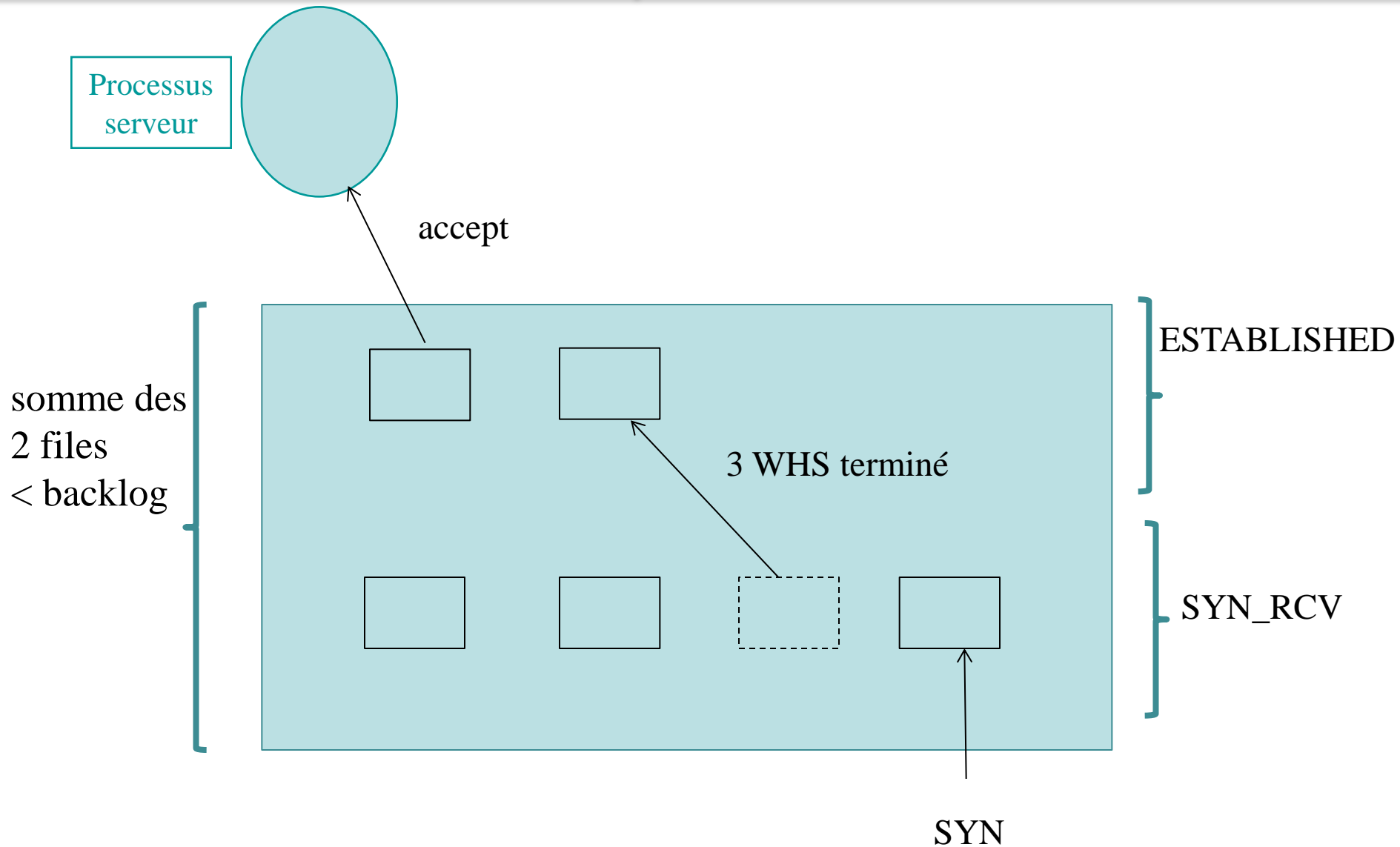
**backlog** → nombre de connexions

simultanément en attente de traitement

**valeur de retour** → -1 si erreur

```
/* Paramétrer le nombre de connexion
"pending" */
if (listen(serverSocket, SOMAXCONN)
<0) {
    perror ("servecho: erreur listen\n");
    exit (1);
}
```





W.R Stevens et al.

# Création d'une socket de dialogue (S)

- Crée une nouvelle socket qui servira au dialogue entre l'appelant et l'entité en attente de traitement de la connexion

```
#include <sys/types.h>
```

```
#include <sys/socket.h>
```

```
int accept (int s, struct sockaddr *peer,  
            socklen_t *addrlen)
```

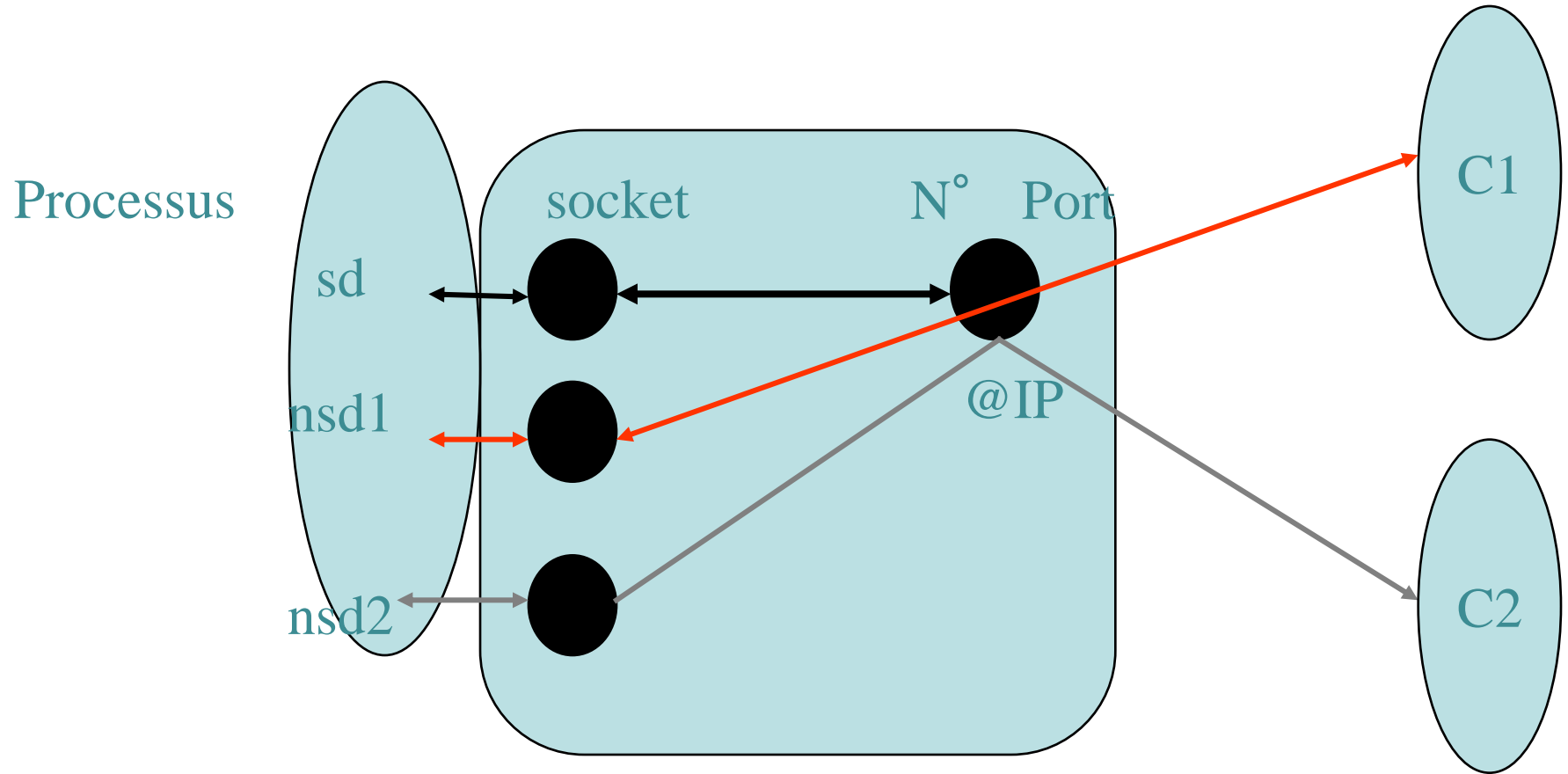
**s** → descripteur retourné par socket()

**peer** → pointeur sur l'adresse du client qui a initié la connexion

**addrlen** → pointeur sur la taille allouée pour peer

**valeur de retour** → -1 si erreur

```
int dialogSocket;  
int clilen,;  
struct sockaddr_in cli_addr;  
  
clilen = sizeof(cli_addr);  
  
dialogSocket = accept(serverSocket,  
                      (struct sockaddr *)&cli_addr,  
                      (socklen_t *)&clilen);  
  
if (dialogSocket < 0) {  
    perror("servecho : erreur accep\n");  
    exit (1);  
}
```



## Demande de connexion (C)

- Utilisée par un **client** (en mode connecté en général) pour établir une connexion avec un serveur distant
- Pendant cette phase : négociation des paramètres du protocole TCP/IP
- Bloque l'appelant jusqu'à l'établissement de la connexion ou la génération d'une erreur
- L'établissement de connexion permet de débloquer le serveur

# Demande de connexion (C)

```
#include <sys/types.h>
```

```
#include <sys/socket.h>
```

```
int connect (int s, struct sockaddr
             *servaddr, socklen_t addrlen)
```

**s** → descripteur retourné par  
socket()

**servaddr** → pointeur sur l'adresse  
du serveur vers lequel initié la  
connexion

**addrlen** → longueur de l'adresse

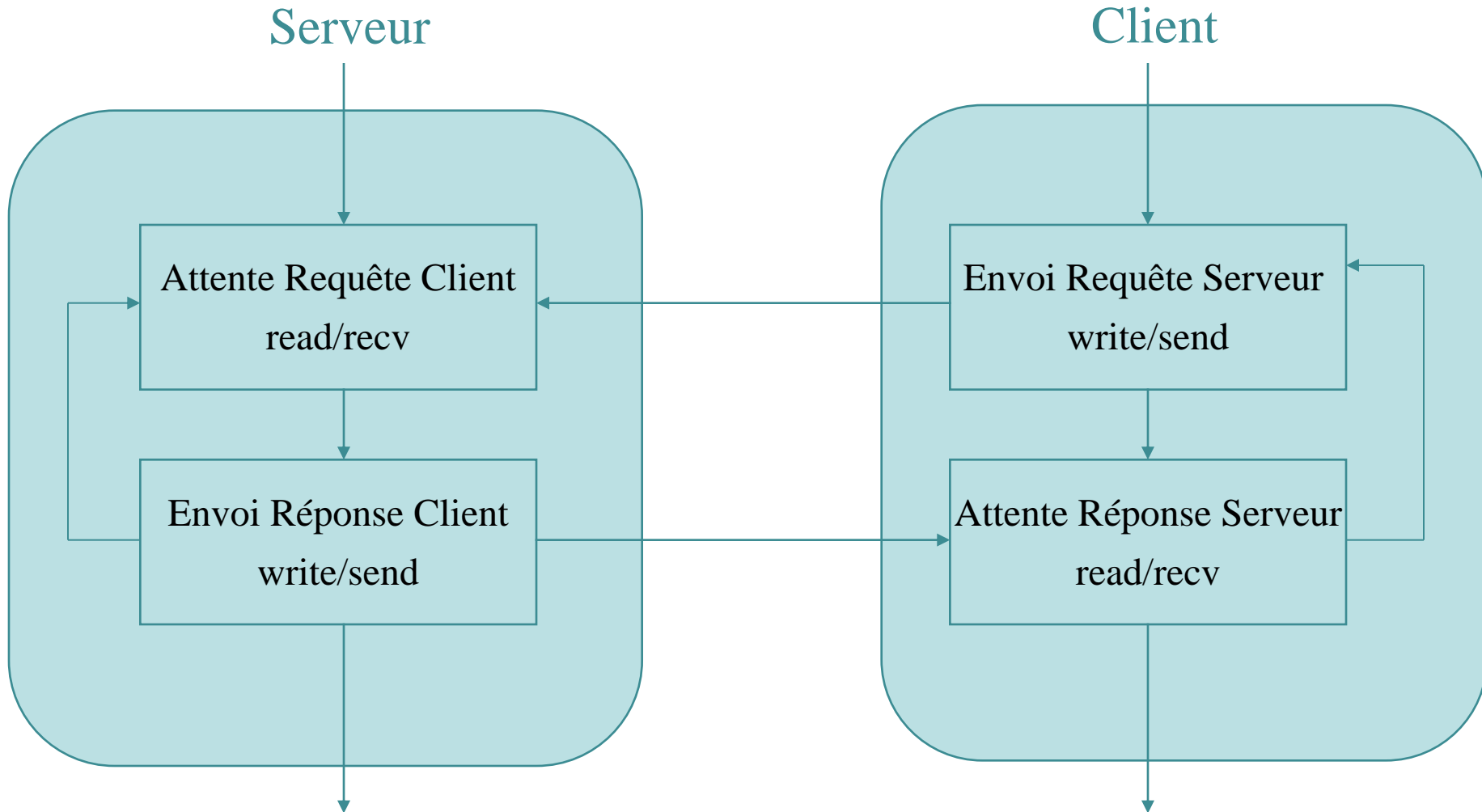
**valeur de retour** → -1 si erreur

```
struct sockaddr_in serv_addr;

/*
 * Remplir la structure serv_addr avec l'adresse du serveur
 */
bzero( (char *) &serv_addr, sizeof(serv_addr) );
serv_addr.sin_family = AF_INET;
serv_addr.sin_port = htons((ushort) atoi(argv[2]));
serv_addr.sin_addr.s_addr = inet_addr(argv[1]);

if (connect (sockfd, (struct sockaddr *) &serv_addr,
             sizeof(serv_addr) ) < 0){
    perror ("cliecho : erreur connect");
    exit (1);
}
```

# Dialogue - échange données (C/S)



# Dialogue - échange de données (C/S)

- send/recv : appels système proches du read() et write() qui peuvent aussi être utilisés pour les sockets

```
#include <sys/types.h>
```

```
#include <sys/socket.h>
```

```
ssize_t send (int s, const void *buff, size_t nbytes, int flags)
```

```
ssize_t recv (int s, void *buff, size_t nbytes, int flags)
```

**s** → descripteur retourné par accept()

**buff** → pointeur sur le buffer où doit être stocké le message à émettre/recevoir

**nbytes** → longueur du message

**flags** → combinaison logique des trois valeurs suivantes :

- MSG\_OOB                      données urgentes
- MSG\_PEEK                    lecture (sans suppression) de données entrantes
- MSG\_DONTROUTE            transmission sans routage

**valeur de retour** → longueur effectivement envoyée/reçue ou -1 si erreur

# Fermeture d'une socket (C/S)

- Libération des ressources système

```
#include <unistd.h>
```

```
int close (int s)
```

**s** → descripteur de socket

**valeur de retour** → -1 si erreur sinon  
descripteur de socket

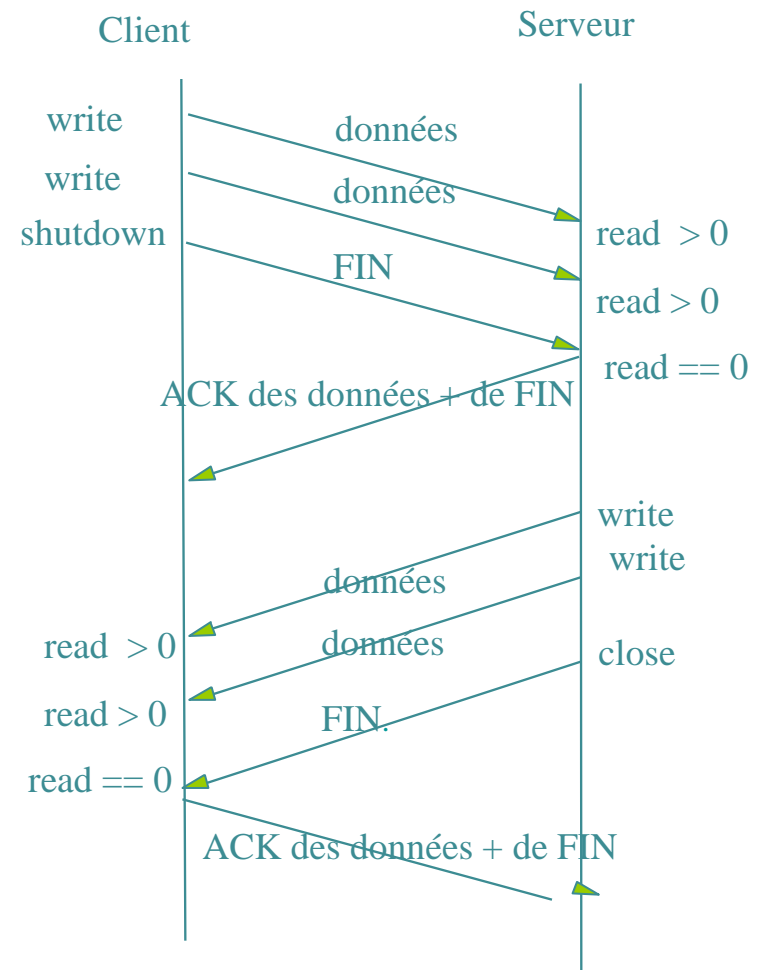
```
#include <unistd.h>
```

```
int shutdown (int s, int how)
```

**s** → descripteur de socket

**how** → 0 fermeture en réception  
1 fermeture en émission  
2 fermeture en émission/réception

**valeur de retour** → 1 si erreur sinon  
descripteur de socket

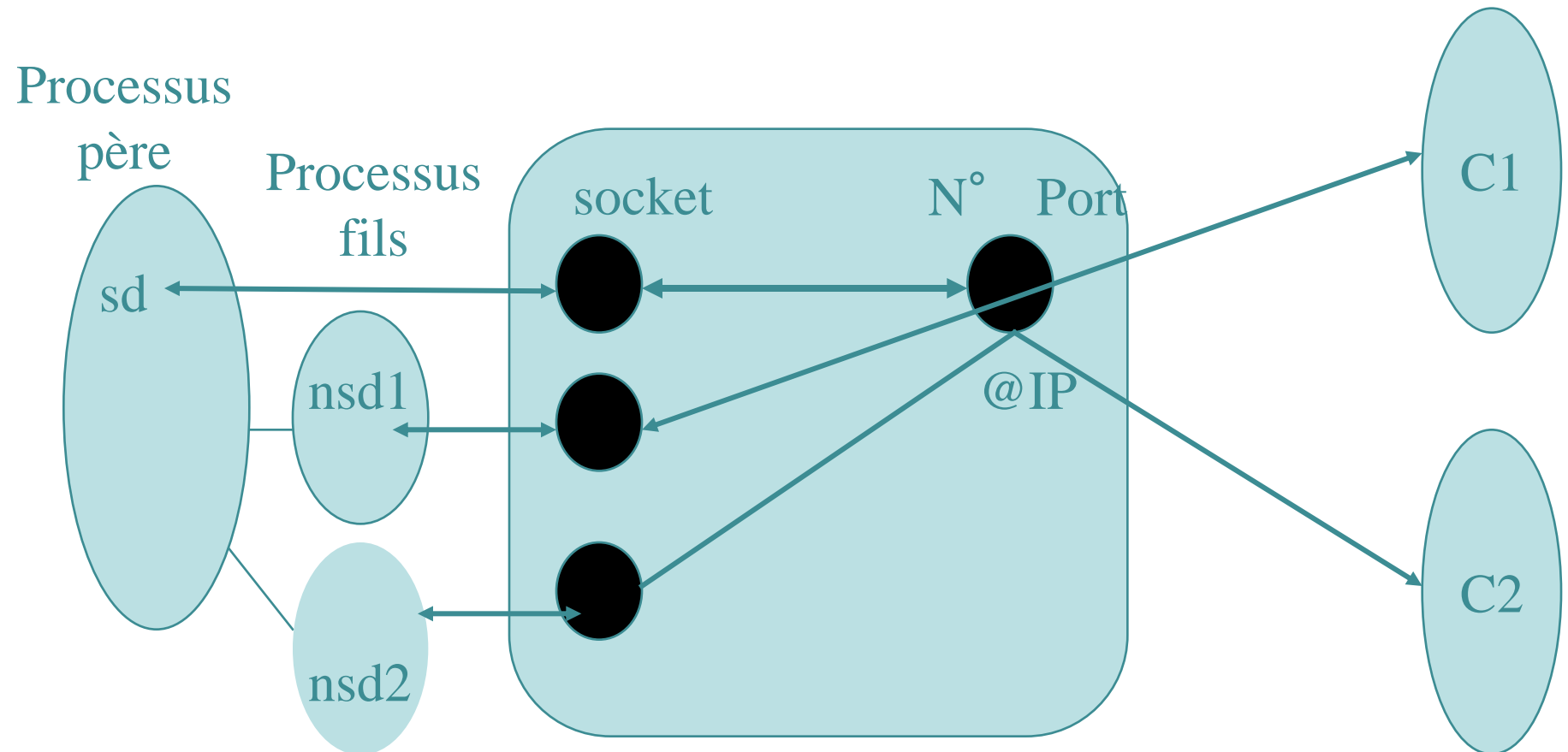


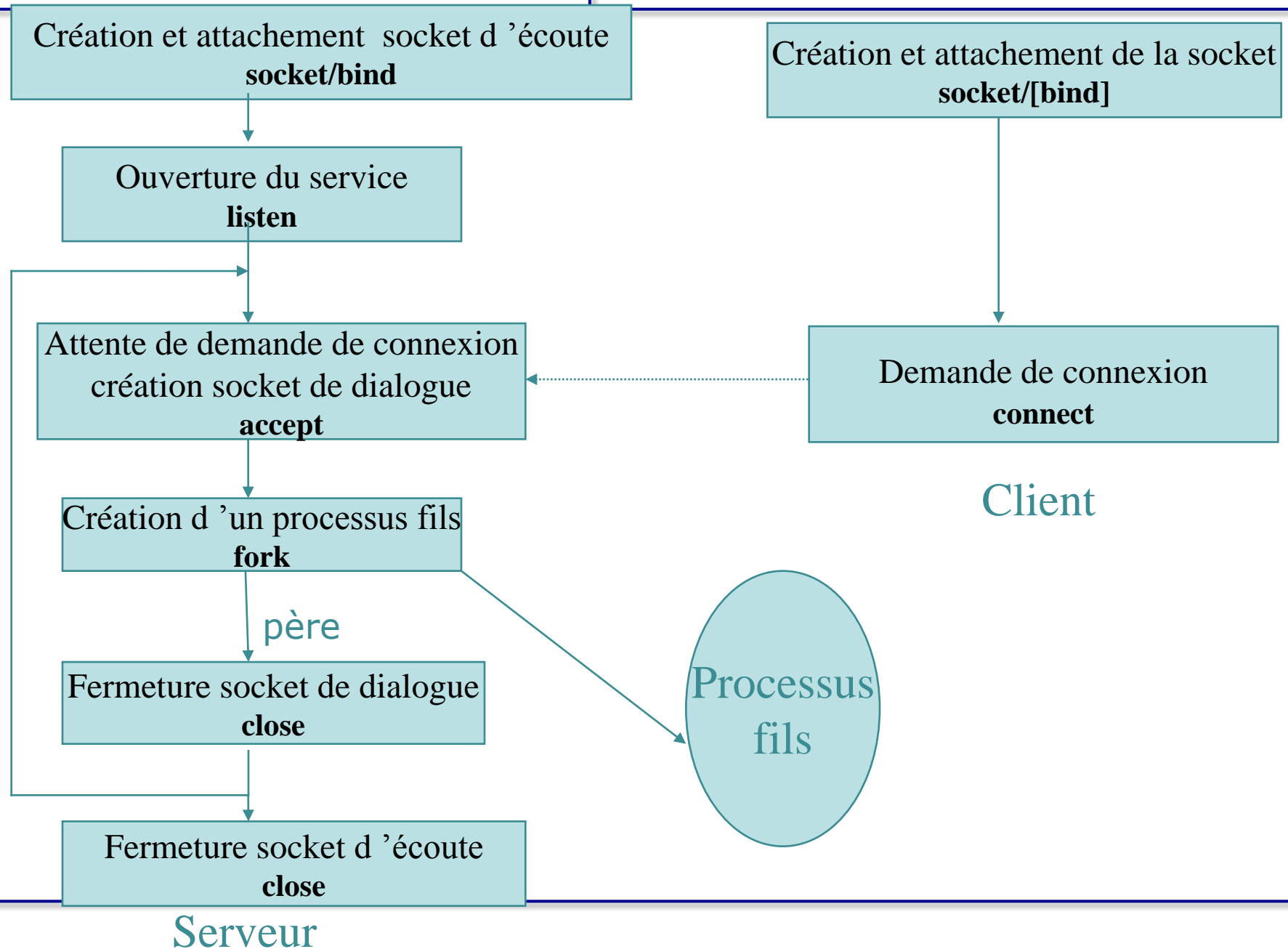
W.R Stevens et al.



# Serveur multi-clients ?

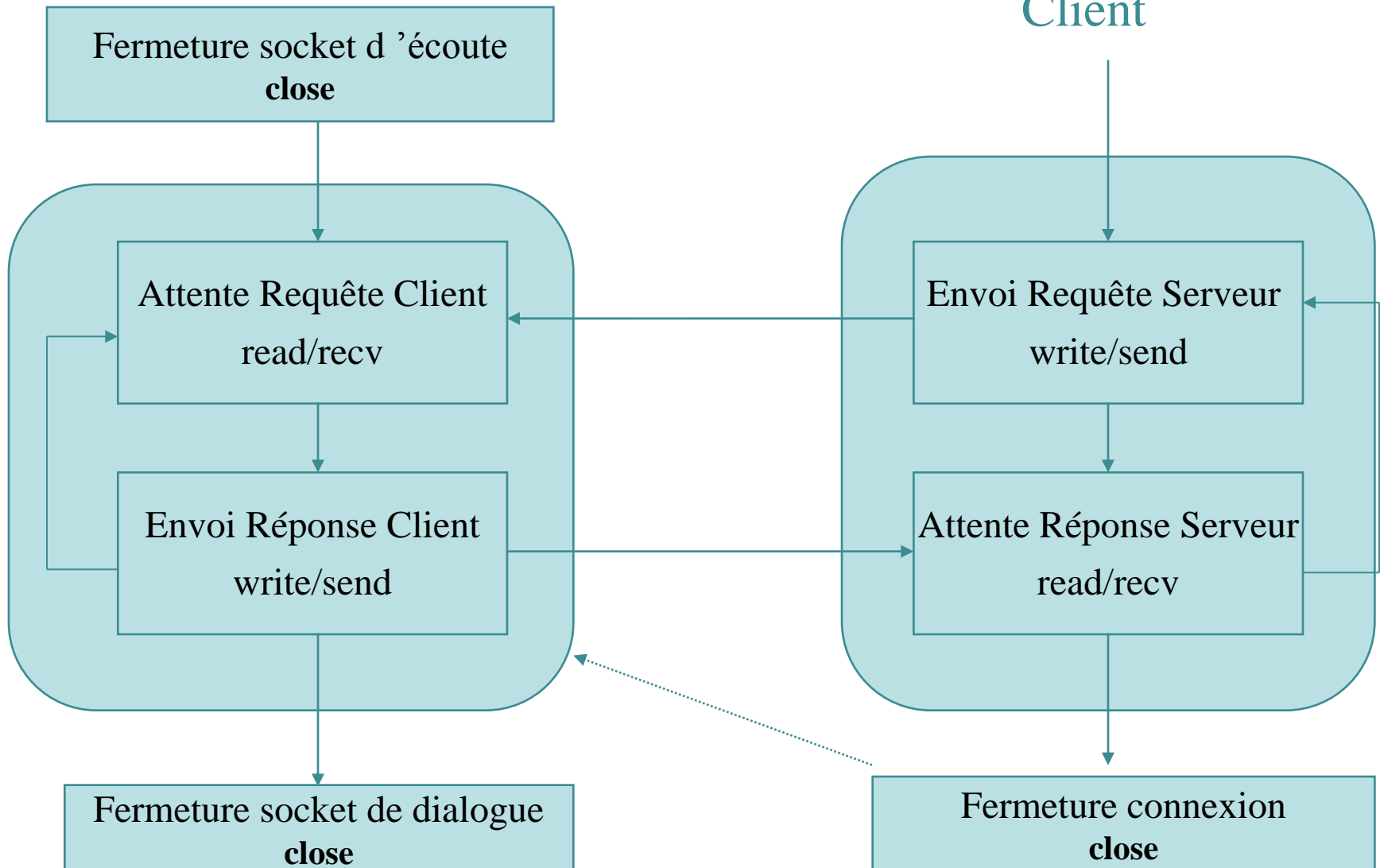
- Un serveur capable de traiter plusieurs clients à la fois





## Serveur/fils

## Client



# Scénario serveur concurrent

```

int sockfd, newsockfd;

if ( (sockfd = socket(..)) < 0) {
    perror("probleme ouverture socket");
    exit(1);
}
if bind (sockfd, ...) <0) {
    perror("probleme bind");
    exit(1);
}
if listen (sockfd, 5) <0) {
    perror("probleme listen");
    exit(1);
}
for(;;){
    if ( (newsockfd = accept (sockfd,...)) < 0) {
        perror("probleme accept");
        exit(1);
    }
    switch ( fork()) {
        case -1 : perror("probleme fork");
                exit(1);
        case 0 : close (sockfd);           /* fils */
                traiter_requête(newsockfd);
                close (newsockfd);
                exit(0);
        default : close(newsockfd);       /* pere*/
    } /* end of switch */
} /* end of for */

```

## ② Programmation réseaux

- Modèle client serveur
- Introduction socket
- Client-Serveur TCP
- **Client-Serveur UDP**
- Entrée/sortie multiplexée : select
- Entrée/sortie asynchrone
- Options de socket
- Socket raw

# Type de Dialogue – UDP/IP

- transmission non fiable
- en mode non connecté
- N'autorise pas les messages urgents
- préservations de limites de messages

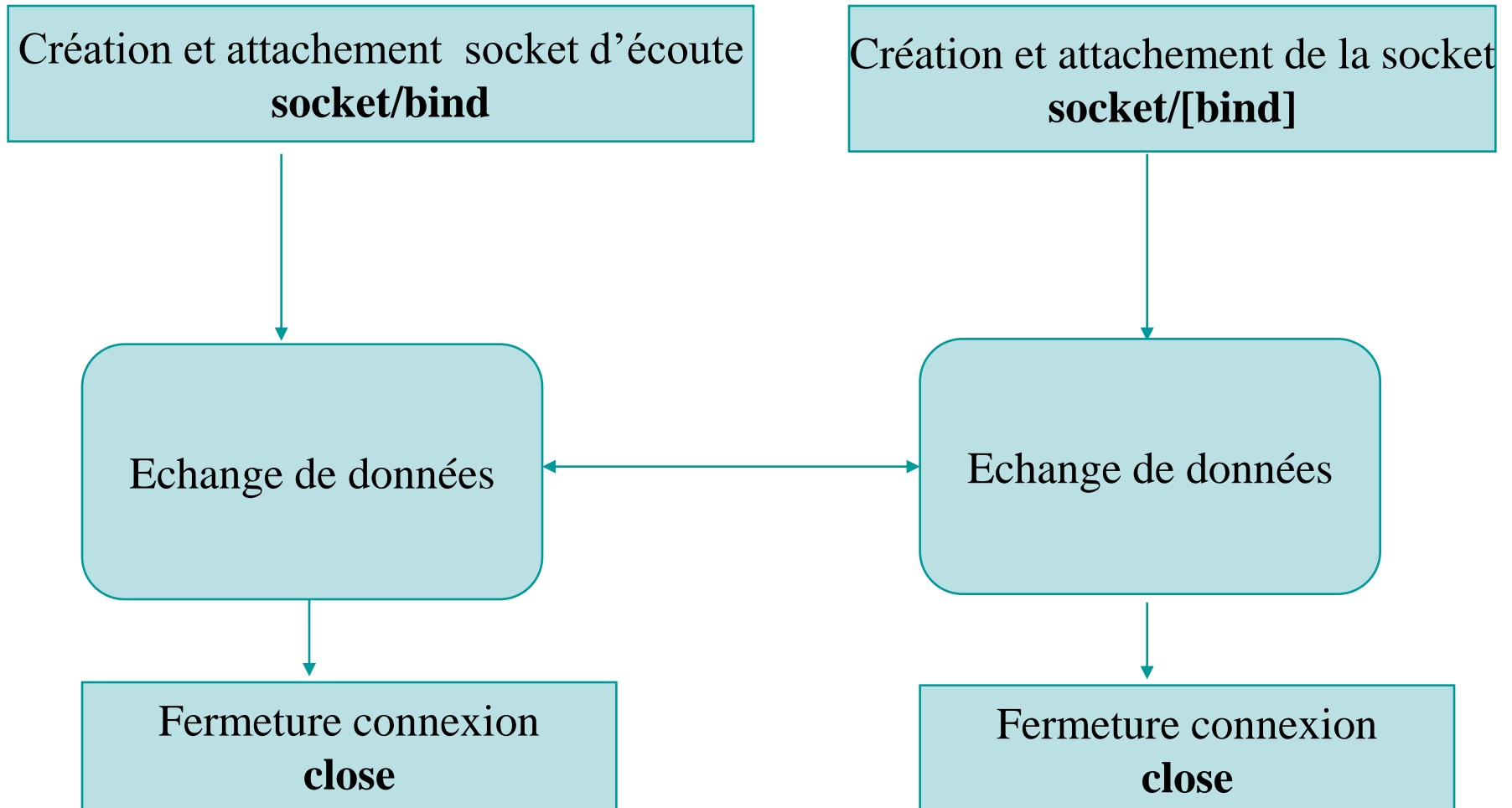


Mode de communication UDP/IP (SOCK\_DGRAM)

User Datagram Protocol/Internet Protocol

## Serveur

## Client



## Création d'une socket (C/S)

```
#include <sys/types.h>  
#include <sys/socket.h>
```

```
int socket (int domain, int type, int protocol)
```

**domain** → [AF\_INET]/PF\_INET

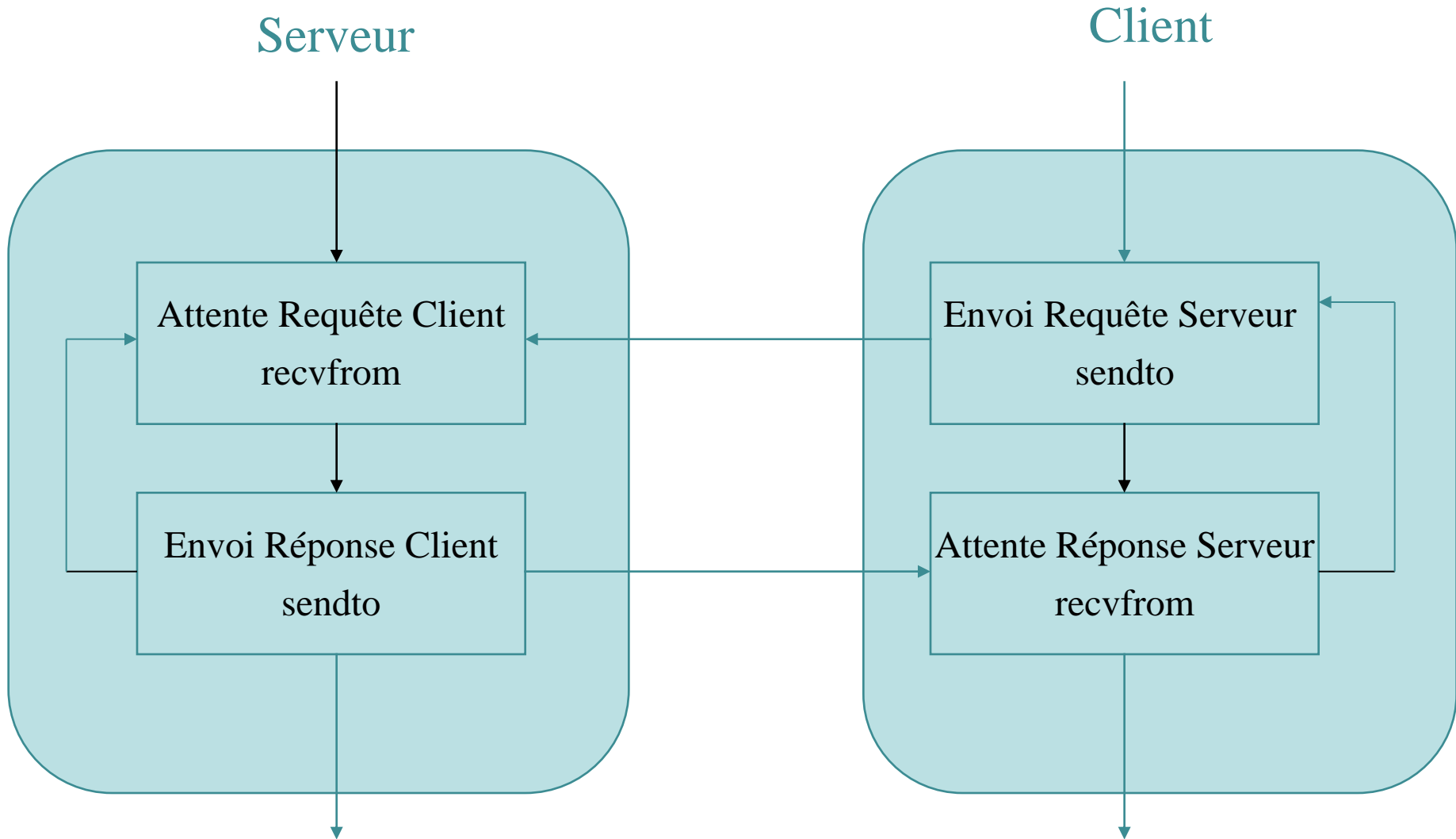
**type** → SOCK\_DGRAM

**protocol** → 0 par défaut

**valeur de retour** → -1 si erreur sinon descripteur de socket



# Dialogue - échange données (C/S)



# Dialogue - échange de données (C/S)

```
#include <sys/types.h>
```

```
#include <sys/socket.h>
```

```
ssize_t sendto (int s, const void *buff, size_t nbytes, int flags,  
               const struct sockaddr *to, socklen_t tolen)
```

```
ssize_t recvfrom (int s, void *buff, size_t nbytes, int flags,  
                 struct sockaddr *from, socklen_t *fromlen)
```

- Les 4 premiers arguments identiques au send/rcv
- to/from : adresse du destinataire/émetteur du datagramme

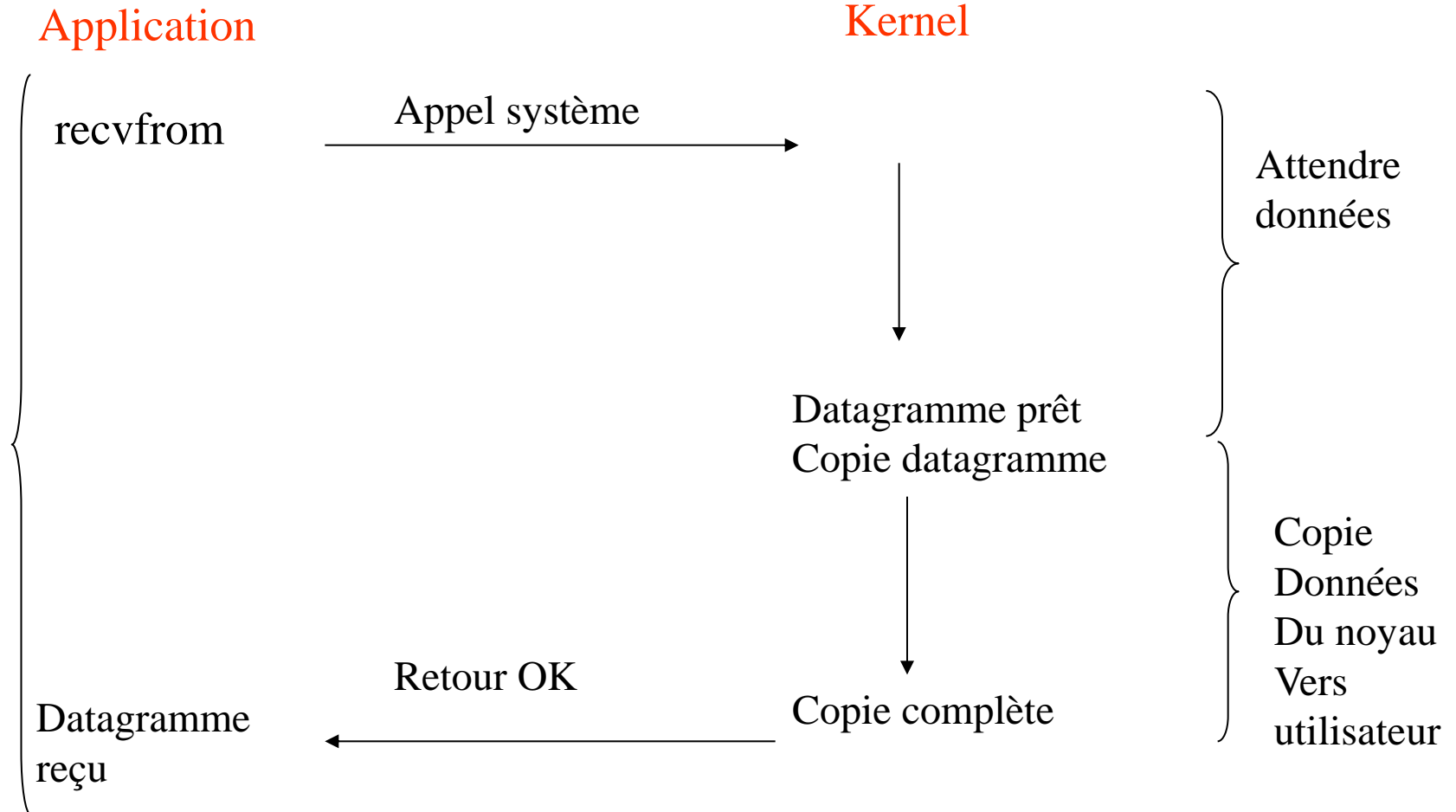
## ② Programmation réseaux

- Modèle client serveur
- Introduction socket
- Client-Serveur TCP
- Client-Serveur UDP
- **Entrée/sortie multiplexée : select**
- Entrée/sortie asynchrone
- Options de socket
- Socket raw

# Modèles Entrée/Sortie

- Quatre modèles d'entrée/sortie
  - E/S bloquante
  - E/S non bloquante
  - E/S multiplexée
  - E/S asynchrone (« signal-driven I/O »)

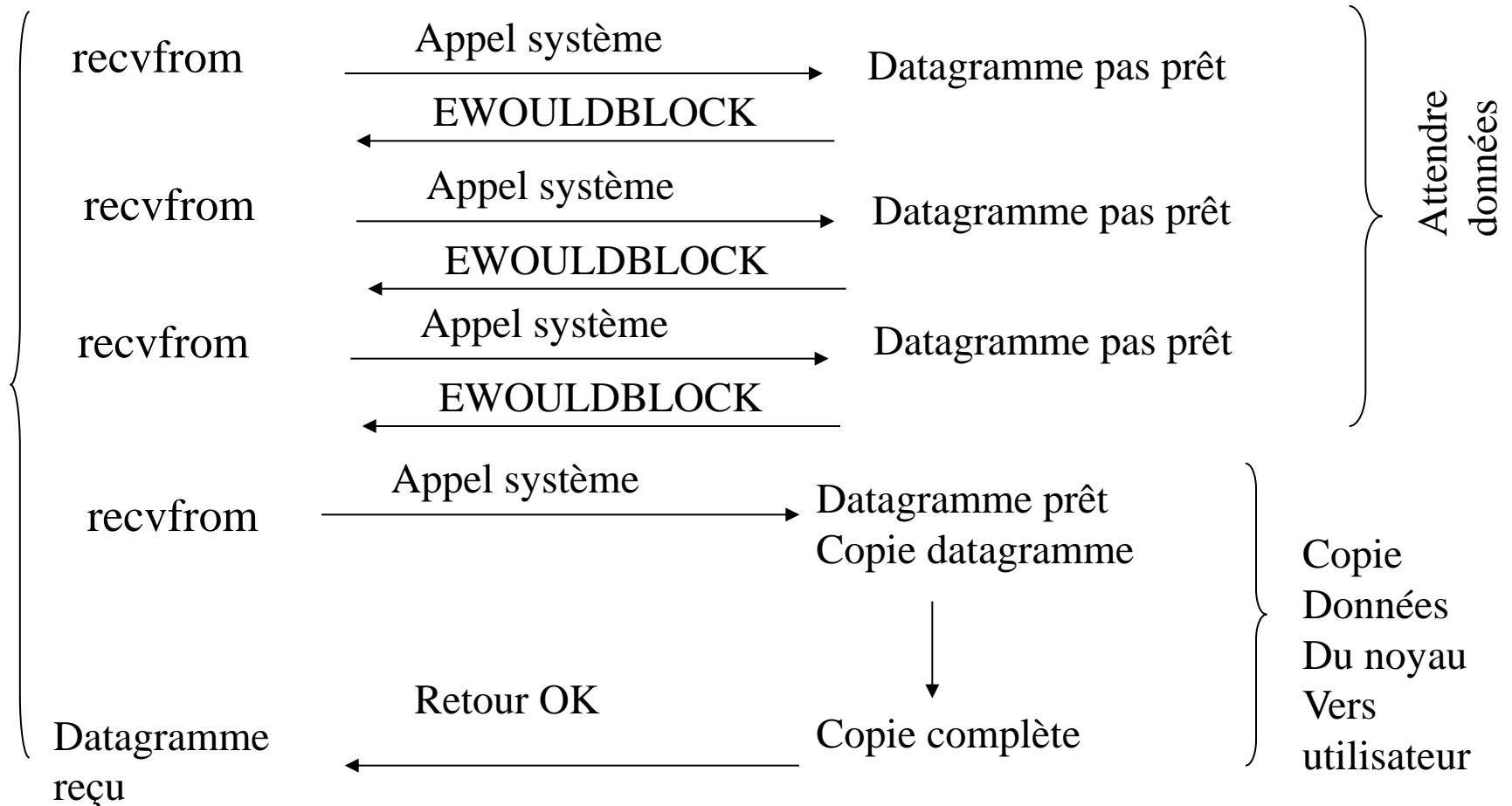
# Entrée/Sortie bloquante



# Entrée/Sortie non bloquante

Application

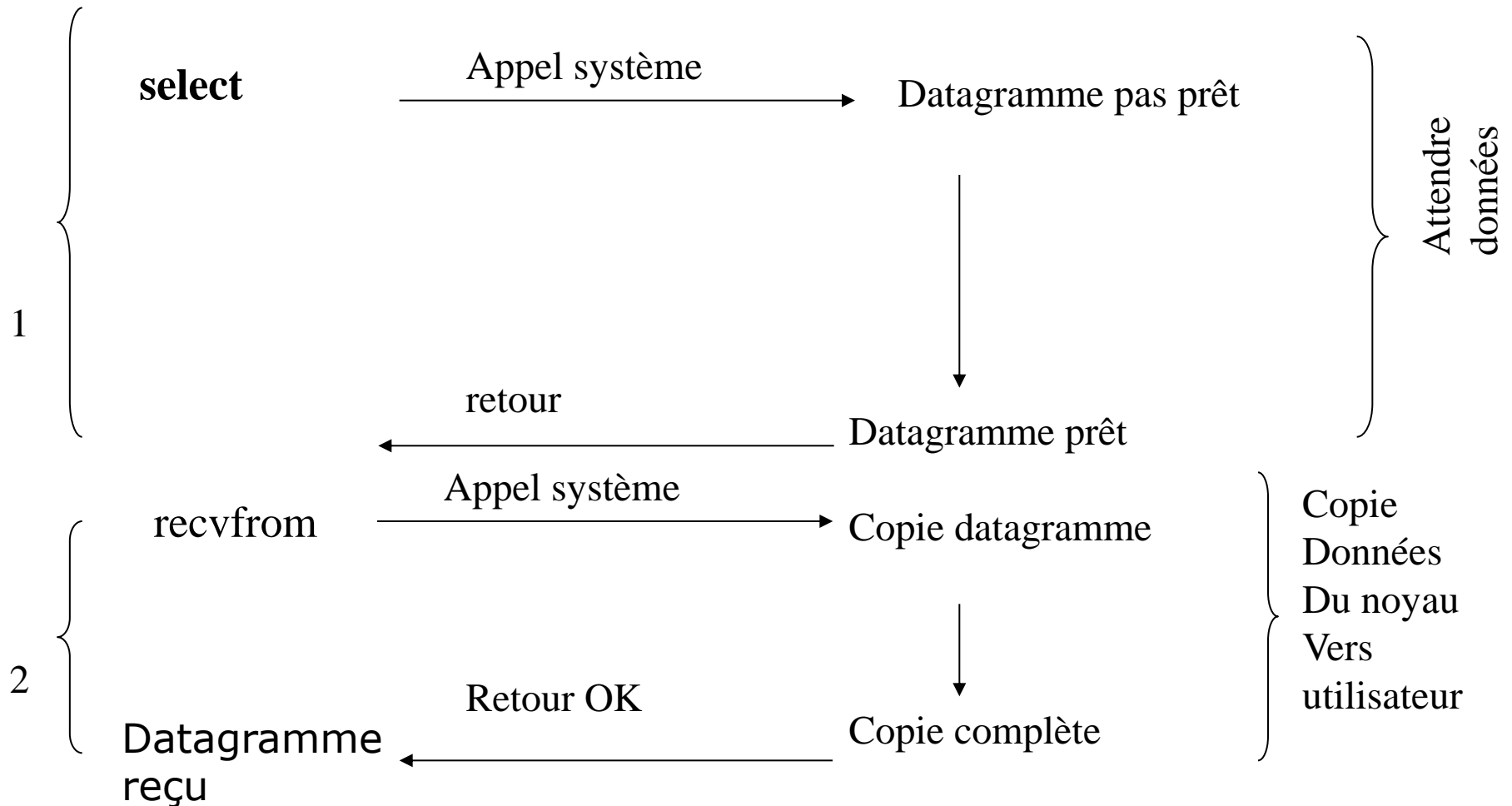
Kernel



# Entrée/Sortie multiplexée

Application

Kernel



# Fonction select()

- Permet l'attente simultanée sur plusieurs flux d'entrée-sortie
- Est utilisée :
  - Quand un client gère plusieurs descripteurs
  - Quand un client gère plusieurs sockets à la fois
  - Dans un serveur TCP, pour gérer à la fois le socket d'écoute et le socket de dialogue
  - Quand un serveur utilise à la fois UDP et TCP
  - Si un serveur gère plusieurs services et plusieurs protocoles
- Se termine :
  - Quand un évènement arrive sur un flux
  - Au bout d'un certain temps



# Fonction select()

```
#include <sys/select.h>
```

```
#include <sys/time.h>
```

```
int select (int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset, const  
            struct timeval *timeout)
```

**maxfdp1** → spécifie le nombre maximum de descripteurs à tester

Les descripteurs 0,1,2,... maxfdp1-1 sont testés

FD\_SETSIZE : nombre de descripteurs dans le type fd\_set

**readset, writeset, exceptset** → masques positionnés par l'appelant et indiquant les descripteurs pour lesquels on veut tester un prédicat en lecture (readset), en écriture (writeset), ou si une condition exceptionnelle se présente (exceptset)

**valeur de retour** → le nombre descripteurs prêts ou

0 si timeout

-1 si erreur

**Au retour les masques sont modifiés et indiquent les descripteurs pour lesquels le prédicat est vérifié**

# Fonction select()

## fd\_set

- fd\_set : tableau d'entiers où chaque bit dans chaque entier correspond à un descripteur
- Macros pour manipuler fd\_set
  - void FD\_ZERO (fd\_set \**fdset*);
  - void FD\_SET (int fd, fd\_set \**fdset*);
  - void FD\_CLR (int fd, fd\_set \**fdset*);
  - Int FD\_ISSET (int fd, fd\_set \**fdset*);

serveur



**fd\_set rset;**  
**FD\_ZERO (&rset);**

0	1	2	3	4	5	FD_SETSIZE-1	
0	0	0	0	0	0	....	0

**rset**

Listening



**socket () + bind () + listen ()**  
**FD\_SET(socketecoute, &rset);**

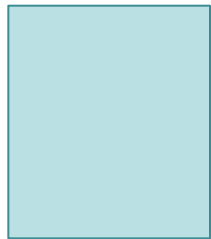
0	1	2	3	4	5	FD_SETSIZE-1	
0	0	0	1	0	0	....	0

**rset**

← maxfdp1 = 4 →

**select(maxfdp1, &rset, NULL, NULL, NULL);**

client1



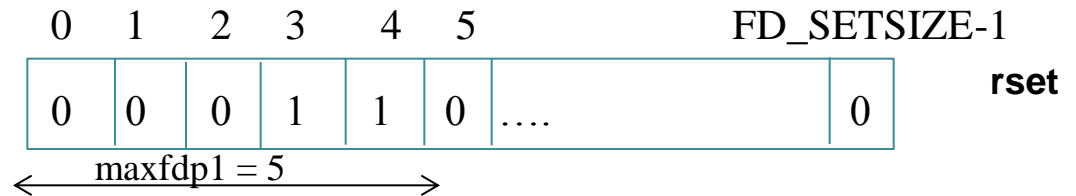
0	1	2	3	4	5	FD_SETSIZE-1	
0	0	0	1	0	0	....	0

**rset**

**3WHS**

**FD\_ISSET (socketecoute, &rset) → TRUE**  
**socketdialogueC1 = accept (socketecoute,..)**

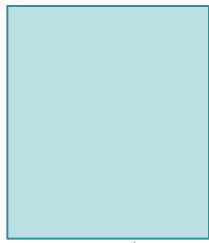
```
FD_SET (socketdialogueC1, &rset);
```



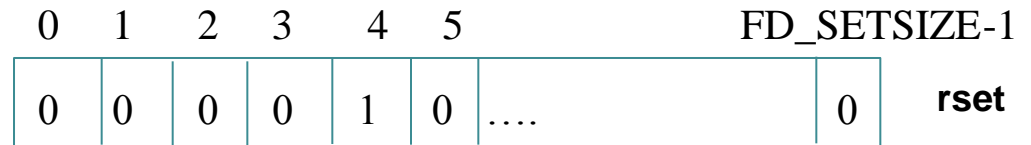
```
select(maxfdp1, &rset, NULL, NULL, NULL);
```

client1

serveur



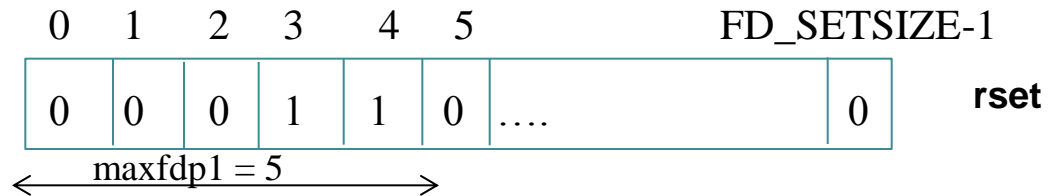
Données



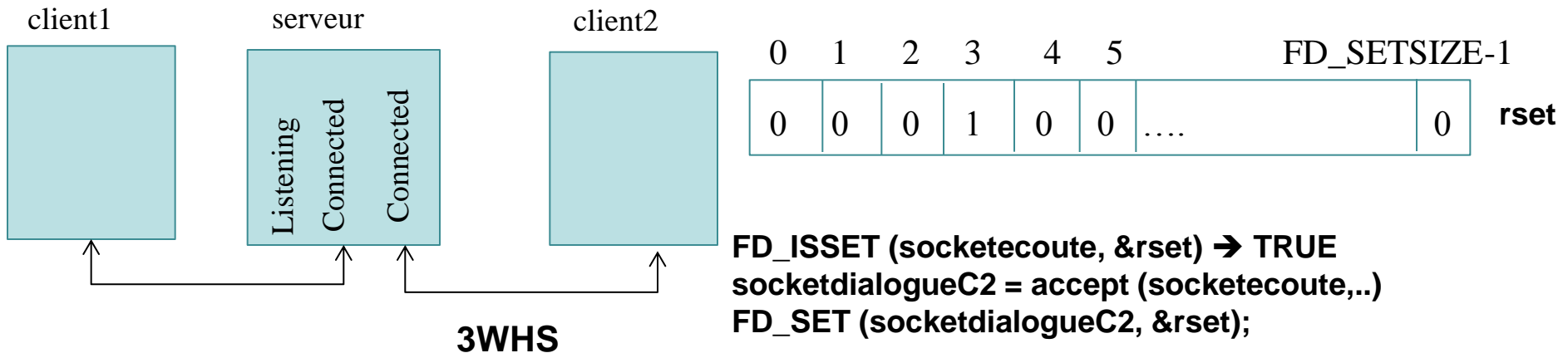
```
FD_ISSET (socketdialogueC1, &rset) → TRUE  
read (socketdialogueC1,...)
```

**Au retour les masques sont modifiés et indiquent les descripteurs pour lesquels le prédicat est vérifié**

**FD\_SET (socketecoute, &rset);**



**select(maxfdp1, &rset, NULL, NULL, NULL);**



**Sauvegarder rset avant appel à select ()**

**cprset = rset;**

ou

**FD\_COPY(&rset , &cprset ); // BSD**

# Fonction select()

## Contrôle de la durée d'attente

```
struct timeval {
    long tv_sec ; /* secondes */
    long tv_usec; /* microsecondes */
};
```

- **Attendre toujours** → pointeur NULL
- **Pas d'attente** : retour de l'appel immédiat → champs internes de timeout initialisés à 0
- **Attente limitée** : attente jusqu'à arrivée d'un évènement ou fin du timer → champs internes de timeout initialisés à une valeur différente de 0

```
struct timeval timeout ;
timeout.tv_sec=1;
timeout.tv_usec=0;
```

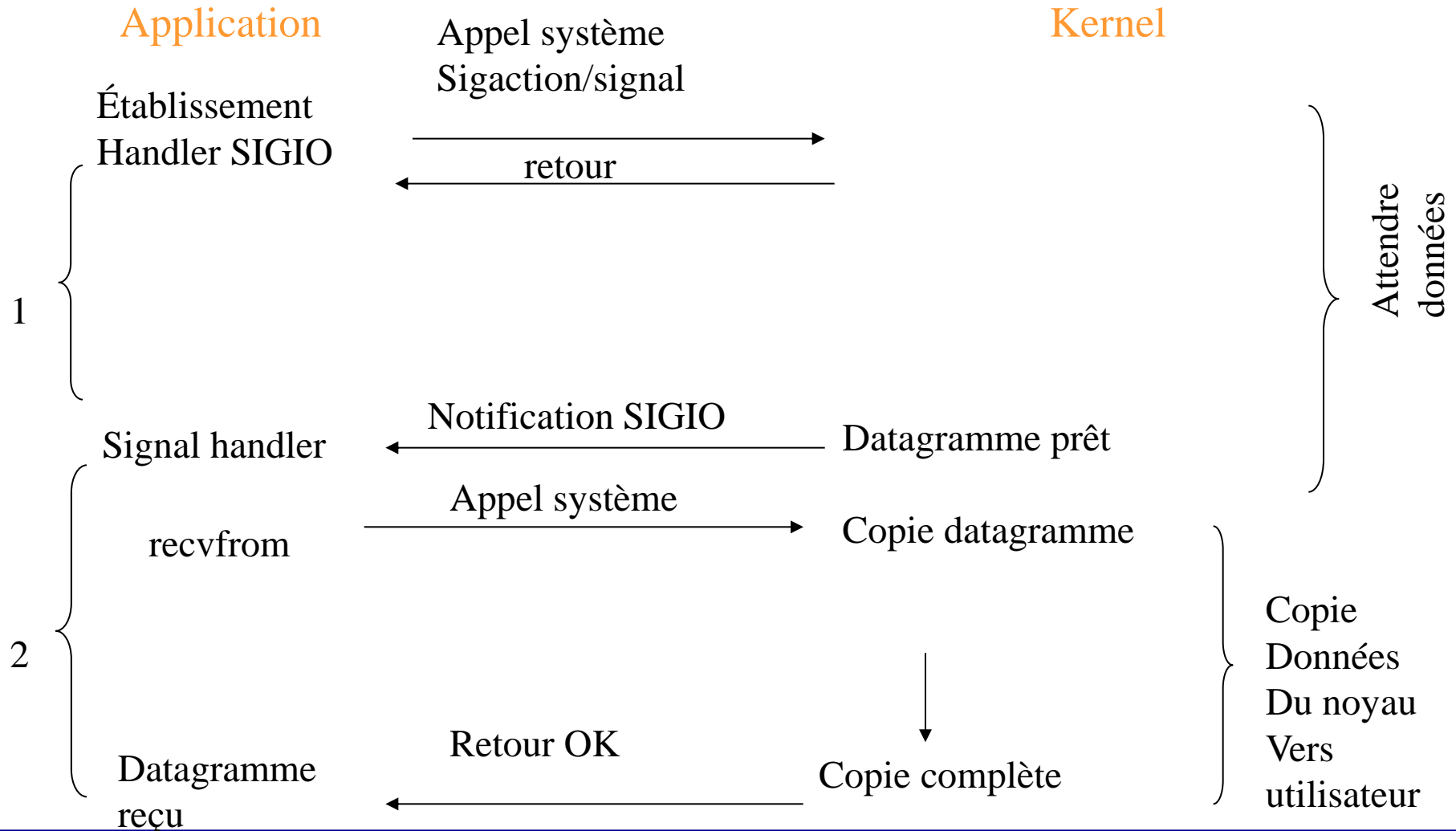
```
if (select(0, NULL,NULL,NULL, &timeout)
    < 0) {
    perror ("cliecho : erreur fgets \n");
    exit(1);
}
```

## ② Programmation réseaux

- Modèle client serveur
- Introduction socket
- Client-Serveur TCP
- Client-Serveur UDP
- Entrée/sorties multiplexée : select
- **Entrée/sortie asynchrone**
- Options de socket
- Socket raw

# Entrée/Sortie asynchrone

## Signal driven I/O





## Socket et signaux

- 3 signaux peuvent être générés pour les sockets
  - **SIGIO**
    - Indique que la socket est prête pour une entrée ou une sortie asynchrone
  - **SIGURG**
    - Indique que des données urgentes sont disponibles en lecture
  - **SIGPIPE**
    - Indique l'écriture n'est plus possible sur le socket concerné

## Socket et signaux

- Permet à un processus d'être interrompu quand un processus est disponible en lecture/écriture
  - Le noyau prévient le processus par le signal SIGIO
  - Le processus doit tout d'abord armer un « handler » pour le signal SIGIO (fonction `signal/sigaction`)
  - Le processus doit ensuite indiquer au noyau qu'il désire recevoir le signal SIGIO
    - fonction `fcntl`, commande `F_SETOWN` et argument `pid` du processus
  - Enfin le processus doit activer les entrées/sorties asynchrones sur le descripteur désiré
    - fonction `fcntl`, commande `F_SETFL` et argument `O_ASYNC`
- Applicable uniquement pour les terminaux et les descripteurs de sockets
- Problème ?

```
#include <signal.h>
#include <fcntl.h>
#define BUFSIZE 4096
int sigflag;
void sigio_func ()
{ sigflag = 1;}

main()
{
int n ;
char buff[BUFSIZE];
signal (SIGIO, sigio_func);
fcntl(0, F_SETOWN, getpid() );
fcntl(0, F_SETFL, O_ASYNC) ;

for (;;) {
    sigblock(sigmask(SIGIO));
    while (sigflag == 0) sigpause (0); /* wait for a signal */

    if ((n = read (0, buff, BUFSIZE)) > 0) {
        write (1, buff, n) ;
    else if (n < 0) perror ("read error");
    else if (n == 0) exit (0);      /* EOF */
    sigflag = 0;
    sigsetmask(0);
}}
}
```

## ② Programmation réseaux

- Modèle client serveur
- Introduction socket
- Client-Serveur TCP
- Client-Serveur UDP
- Entrée/sorties multiplexée : select
- Entrés/sortie asynchrone
- Options de socket
- Socket raw

# Options sur les sockets

- Plusieurs fonctions disponibles pour positionner et obtenir les valeurs des options sur les sockets
  - **getsockopt()**, **setsockopt()**, **fcntl()**, **ioctl ()**

```
#include <sys/types.h>
```

```
#include <sys/socket.h>
```

```
int getsockopt (int fd, int level, int optname, void *optval, socklen_t *optlen)
```

```
int setsockopt (int fd, int level, int optname, const void *optval,  
                socklen_t optlen)
```

**level** → permet d'indiquer la couche logicielle en charge de l'option (socket, TCP, IP) : SOL\_SOCKET, IPPROTO\_IP, IPPROTO\_TCP

**optname** → nom de l'option

**optval** → pointeur générique contenant la valeur de l'option

**optlen** → contient la taille de la valeur

## (level) SOL\_SOCKET

- Optname **SO\_BROADCAST**
  - permet d'envoyer des datagrammes en broadcast au niveau des réseaux locaux
- Optname **SO\_RCVBUF** ou **SO\_SNDBUF**
  - permet de préciser la taille du buffer de réception ou émission des sockets TCP/UDP
    - Appel avant ou après connect(côté client) ?
    - Appel avant ou après listen(côté serveur) ?

```
int on=1;
setsockopt (sockfd, SOL_SOCKET,
SO_BROADCAST,&on, sizeof(on));
```

```
int size = 4096;
setsockopt (sockfd, SOL_SOCKET,
SO_RCVBUF, &size,sizeof(size));
```

## (level) SOL\_SOCKET

- Optname **SO\_REUSEADDR / SO\_REUSEPORT**
  - permet d'indiquer qu'une même adresse locale peut être réutilisée
- Optname **SO\_LINGER**
  - spécifie comment la fonction close se comporte

```
struct linger{  
    int      l_onoff; /* 0 → off */  
    int      l_linger; /* temps en sec */  
}
```

Comportement par défaut (`l_onoff == 0`) : la fonction close retourne immédiatement

## (level) IPPROTO\_IP

- Optname **IP\_OPTIONS**
  - permet d'utiliser les options de l'entête IP
- Optname **IP\_TTL**
  - permet de préciser TTL qui va être utilisé par le système
  - dans le cas du multicast → **IP\_MULTICAST\_TTL**
- Optname **IP\_HDRINCL**
  - permet d'utiliser les sockets RAW



## (level) IPPROTO\_TCP

- Optname **TCP\_MAXSEG**
  - spécifie la taille de segment maximum
- Optname **TCP\_NODELAY**
  - désactive l'algorithme de Nagle

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>

main()
{
    int sockfd, maxseg, sendbuf ;
    socklen_t optlen ;
    if ( (sockfd=socket(AF_INET, SOCK_STREAM,0)) <0){
        perror("probleme socket"); exit(1) ;
    }
    optlen = sizeof(maxseg);
    if (getsockopt(sockfd, IPPROTO_TCP, TCP_MAXSEG, &maxseg, &optlen) <0){
        perror("probleme getsockopt"); exit(1) ;
    }
    printf ("TCP maxseg = %d\n",maxseg);
    sendbuf = 16384;
    if (setsockopt(sockfd, SOL_SOCKET, SO_SNDBUF, &sendbuf, sizeof(sendbuf))<0){
        perror("probleme setsockopt"); exit(1) ;
    }
    optlen = sizeof(sendbuf);
    if (getsockopt(sockfd, SOL_SOCKET, SO_SNDBUF, &sendbuf, &optlen)) <0){
        perror("probleme getsockopt"); exit(1) ;
    }
    printf ("send buffer size = %d\n",sendbuf);
}
```

## ② Programmation réseaux

- Modèle client serveur
- Introduction socket
- Client-Serveur TCP
- Client-Serveur UDP
- Entrée/sorties multiplexée : select
- Entrées/sortie asynchrone
- Options de socket
- **Socket raw**

## Caractéristiques d'une socket raw

- Permet à un processus de lire et d'envoyer des paquets de contrôle (ICMP, IGMP)
- Permet à un processus de lire et d'envoyer des datagrammes IP avec un champ *protocol* non traité par le noyau
- Permet à un processus de construire sa propre entête IPv4 (options socket `IP_HDRINCL`)

# Création

```
int sockfd;  
  
if ( (sockfd=socket(AF_INET, SOCK_RAW, protocol)) <0){  
    perror("probleme socket");  
    exit(1) ;  
}
```

- **protocol** → constante définie dans `<netinet/in.h>` comme `IPPROTO_ICMP`
- Création uniquement en mode super utilisateur

# Création

```
const int on =1:

if ( setsockopt(sockfd, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on)) <0){
    perror("probleme setsockopt");
    exit(1) ;
}
```

- Positionnement de l'option IP\_HDRINCL si en-tête IP est inclus dans la donnée
- Fonction bind peut être appelée mais rare
  - Pas de concept de numéro de port, uniquement adresse locale
- Fonction connect peut être appelée mais rare

# Emission

- Utilisation de **sendto** en spécifiant l'adresse IP destination
- Si l'option IP\_HDRINCL n'est pas positionnée
  - L'adresse de début de la donnée = le 1er octet suivant l'en-tête IP
- Le noyau fragmente les paquets raw qui excèdent la taille du MTU de l'interface sortante

# Réception

- Utilisation de **recvfrom**
- Les paquets TCP et UDP reçus ne sont jamais passés à une socket RAW
- La plupart des paquets ICMP sont passés à une socket RAW une fois que le noyau a fini de traiter le message ICMP
  - Les implémentations dérivées de BSD passent tous les paquets ICMP sauf echo request, timestamp request et address mask request.
- Tous les paquets IGMP sont passés à une socket RAW une fois que le noyau a fini de traiter le message IGMP
- Tous les datagrammes IP avec un champ *protocol* non reconnu par le noyau sont passés à une socket RAW



# Réception

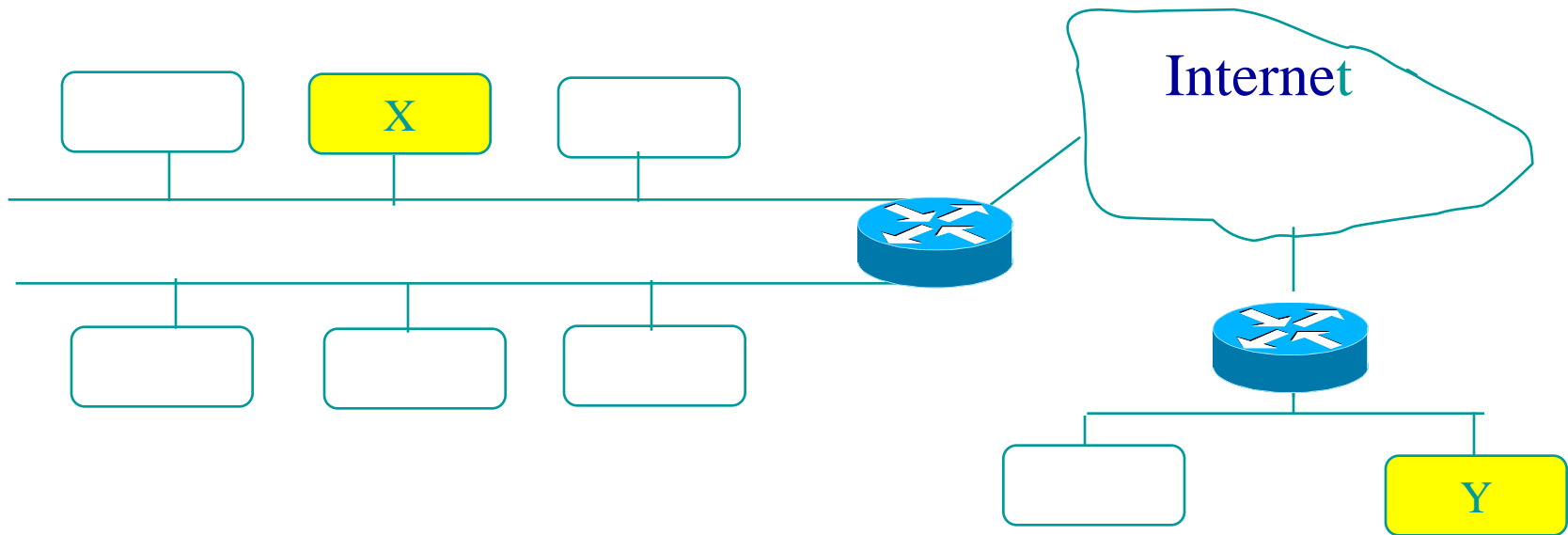
- Si un datagramme arrive en fragments :
  - Attente de tous les fragments
- Tests à vérifier pour obtenir un datagramme IP
  - Si *protocol*  $\neq 0$  spécifié à la création du socket RAW alors
    - Champ *protocol* du datagramme reçu == champ *protocol* spécifié
  - Si une adresse IP locale est liée au socket RAW par *bind* alors
    - Adresse destination du datagramme reçu == adresse locale
  - Si une adresse distante est spécifiée par *connect* alors
    - Adresse source du datagramme reçu == adresse distante

# Plan

- ① Contrôle de congestion et TCP
- ② Programmation réseaux
  - socket, select(), options de socket, socket raw
- ③ Communication de groupe (multicast)
- ④ Introduction IPv6

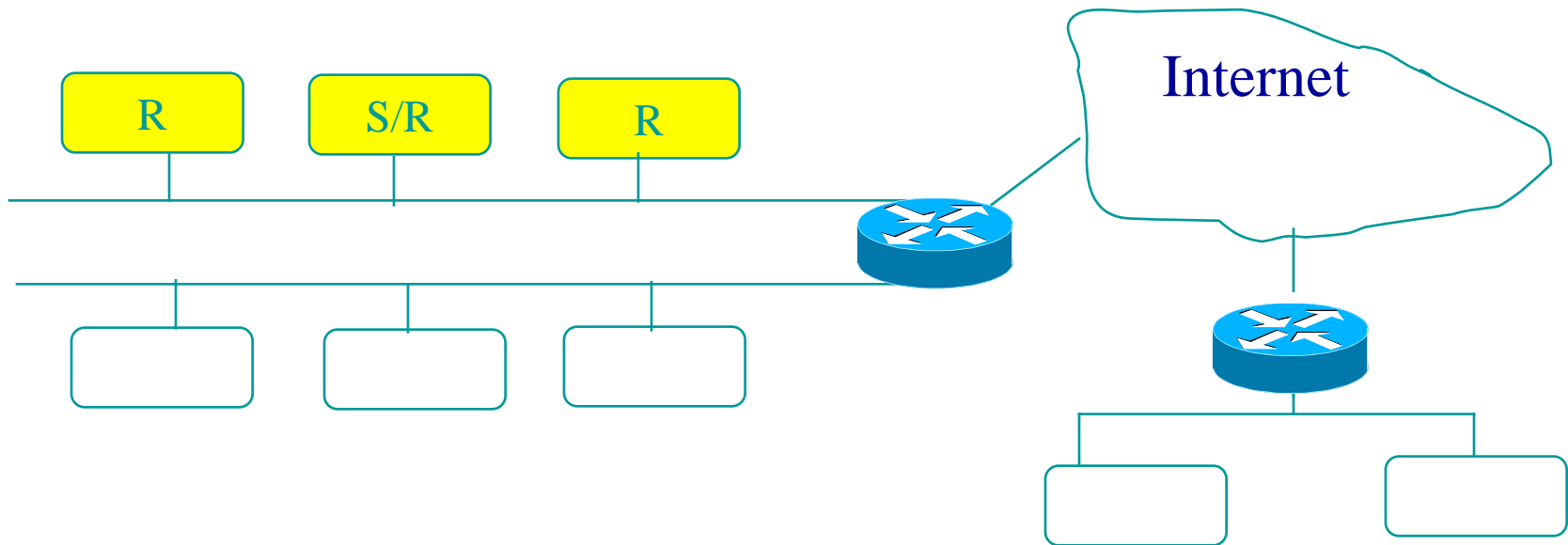
# Unicast, Broadcast et Multicast

- IP Unicast (diffusion point à point)  
Liaison point à point entre 2 machines : source - destination



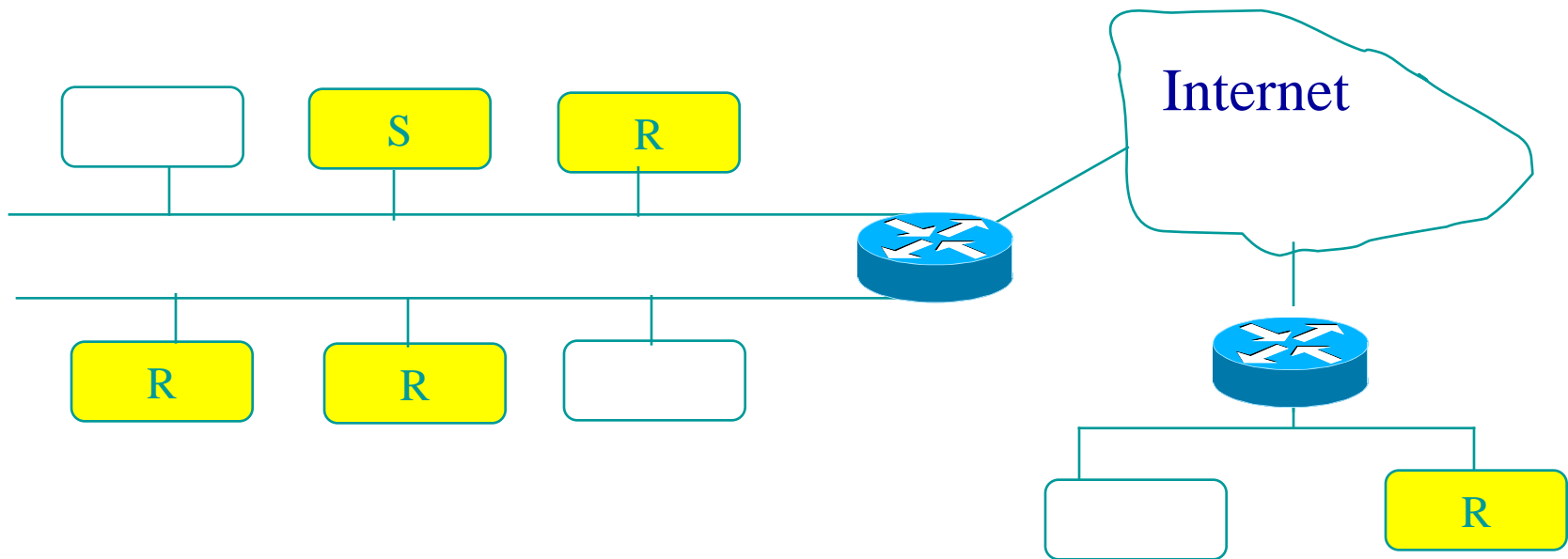
# Unicast, Broadcast et Multicast

- IP Broadcast (diffusion générale)  
Diffusion de paquets limitée au réseau local



# Unicast, Broadcast et Multicast

- IP Multicast (diffusion de groupes/multipoint)  
Le multicast atteint un groupe de machines qui peuvent être réparties sur des réseaux distincts



# Applications du Multicast

- Téléconférence
- Jeux distribués
- Distribution de logiciels
- Vidéo à la demande
- Mise à jour des bases de données répliquées

# Le modèle IP multicast

- Défini par Steve Deering
  - [Multicast Routing in a datagram internetwork, Ph.D Thesis, Stanford University, 1991]
- Présente
  - la notion de groupe,
  - le type d' adressage,
  - Extension de l'interface de service IP du module IP pour envoyer et recevoir des données
  - Le protocole d'adhésion au groupe sur un réseau local

# La notion de groupe/multicast

- Un groupe est un ensemble de 0 ou n machines
- Un groupe est entièrement dynamique
  - Une station peut rejoindre ou quitter le groupe à tout moment
- Un groupe est ouvert
  - Une station peut émettre un paquet dans un groupe sans en faire partie
- Un groupe est temporaire ou non
  - Adresses allouées dynamiquement ou non
- Un groupe est désigné par une adresse IP



# Adressage de groupe/multicast

- Adresse Internet de Classe D :

- 224.0.0.0 -> 239.255.255.255

1   1   1   0	Identification du groupe
---------------	--------------------------

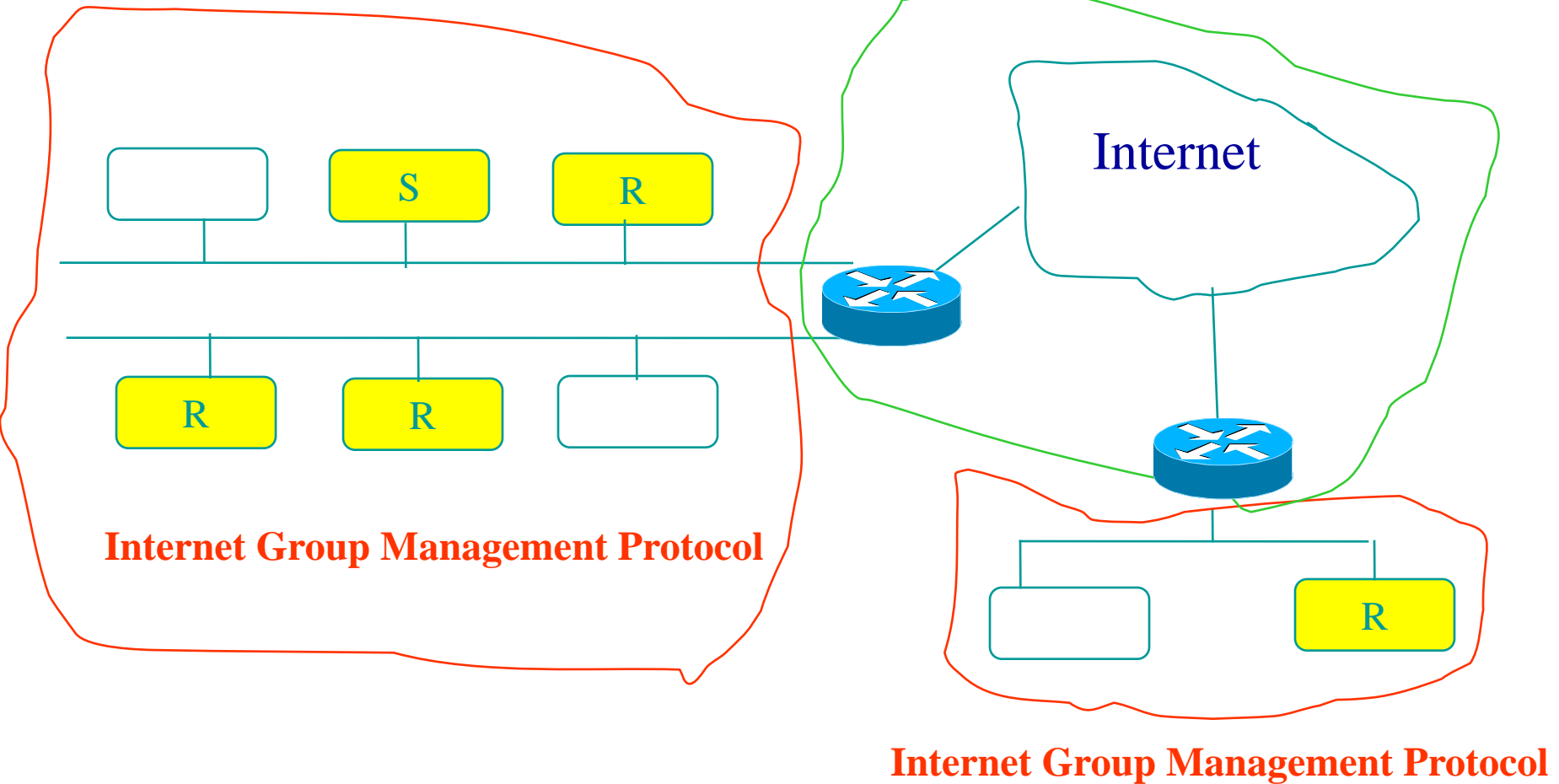
- Adresse utilisée uniquement en destination
- Réservation de plages d'adresses spécifiques
  - <http://www.iana.org/numbers.html>
  - 224.0.0.0 -> 224.0.0.255 (224.0.0.0/24)
    - Réservées pour la diffusion sur le LAN
    - 224.0.0.1 tous les hosts multicast sur un LAN
    - 224.0.0.2 tous les routeurs multicast sur un LAN

# Adressage de groupe/multicast

- Correspondance adresse IP multicast et adresse Ethernet multicast
  - Pour mettre en correspondance les adresses de diffusion Ethernet, placer les 23 bits de poids faibles des adresses de diffusion au niveau IP dans les 23 bits de l'adresse de diffusion de groupe au niveau Ethernet spécifique suivante :
    - 01:00:5E:00:00:00
  - Adresse IP : 224.255.26.9      Adresse MAC = ??????
  - Remarque ?

# Extension Interface de service IP et Module IP

- Émission de données
  - Emission d'un paquet multicast est identique à l'émission en point-à-point
  - Pour autoriser les transmissions multicast, la logique de routage doit être changée :
    - if IP-destination is on the same local network **or IP-destination is a host group**,  
send datagram locally to IP-destination  
else send datagram locally to Gateway
- Réception de données
  - Nécessite un abonnement préalable au groupe pour laisser passer vers le niveau supérieur les données relatives à une adresse de diffusion :
  - Ajout des nouvelles primitives à l'interface comme Join et Leave (=> IGMP )

**Multicast Routing Protocol**

# Le protocole IGMP

- Protocole d'interaction entre :
  - Les routeurs multicast du réseau local
  - Les machines hôtes multicast du réseau local
- Les hôtes informent les routeurs de leur appartenance au groupe (abonnement/désabonnement)
- Les routeurs sollicitent directement les hôtes connectés

# Le protocole IGMP

- Comme ICMP, les messages IGMP sont encapsulés dans des datagrammes IP
  - numéro de protocole 2
  - TTL à 1
  - l'option IP Router Alert dans l'entête IP
- Trois versions de IGMP :
  - v1 (rfc 1112)
  - v2 (rfc 2236)
  - v3 (rfc 3376)

# Le protocole IGMP

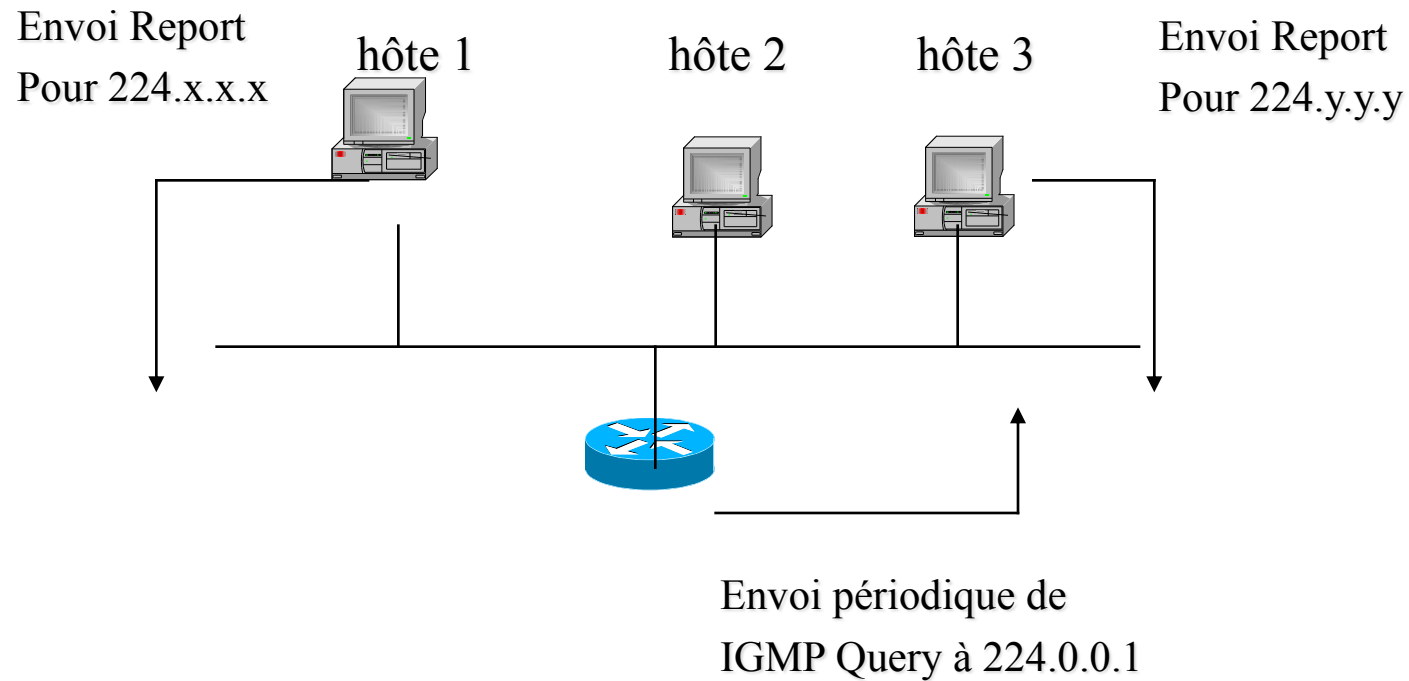
- Le routeur envoie de manière périodique (60s) un *IGMP Query* à l'adresse 224.0.0.1
  - Sollicitation des hôtes pour savoir à quel groupe ils sont abonnés
- Les hôtes répondent par un *IGMP Report*
  - Précisant l'adresse du ou des groupes auxquels ils appartiennent
- Si le routeur ne reçoit plus de réponses pour un groupe spécifique
  - Les paquets multicast pour ce groupe ne seront plus retransmis dans le réseau local
  - Le routeur prévient les autres routeurs multicast en amont via un protocole de routage multicast

# Le protocole IGMP

- Sur réception d'un *Query* :
  - L'hôte fixe un délai aléatoire avant de répondre
  - Si un autre hôte répond avant lui, il arrête le temporisateur et n'émet plus le *Report*
  - Diminution de la surcharge du réseau : une réponse par groupe multicast sur un réseau local
- Quand un hôte joint un nouveau groupe, il transmet un *Report* pour ce groupe



# Abonnement

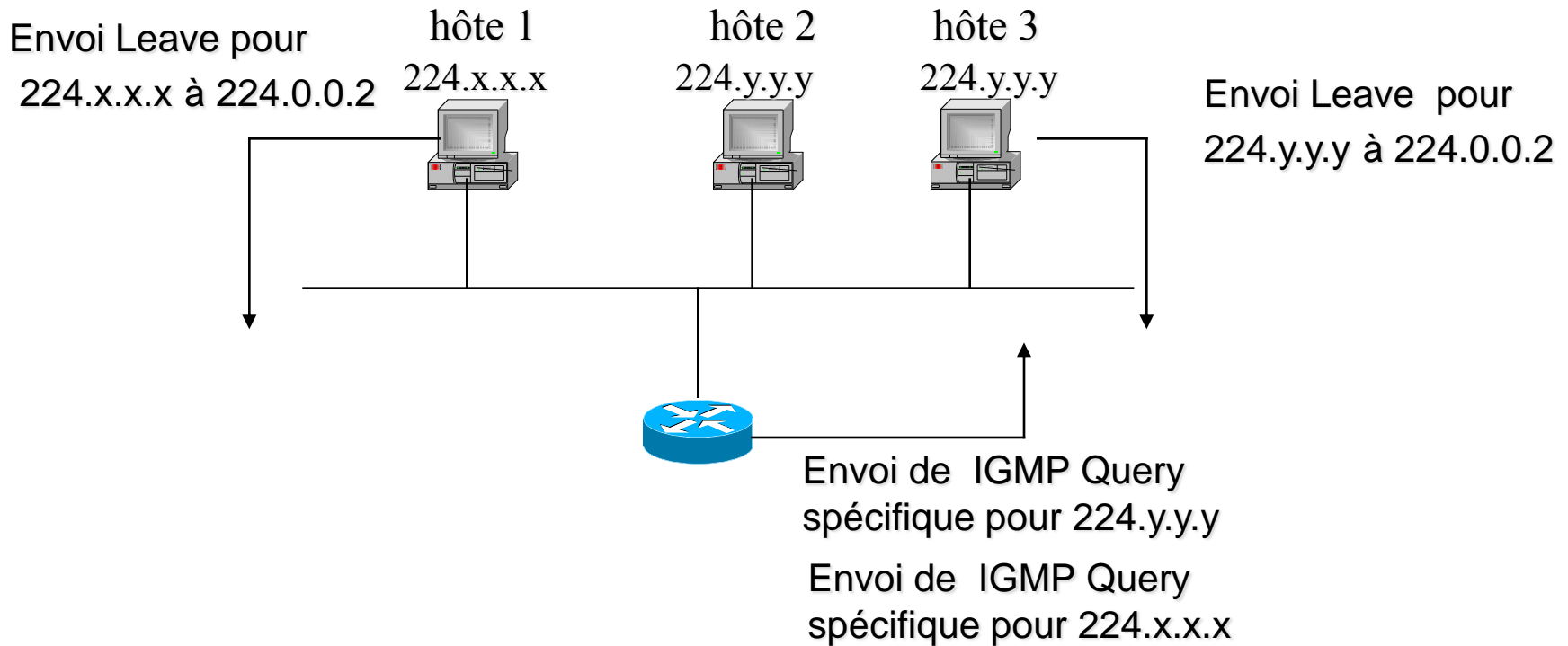


## Evolutions vers IGMPv2

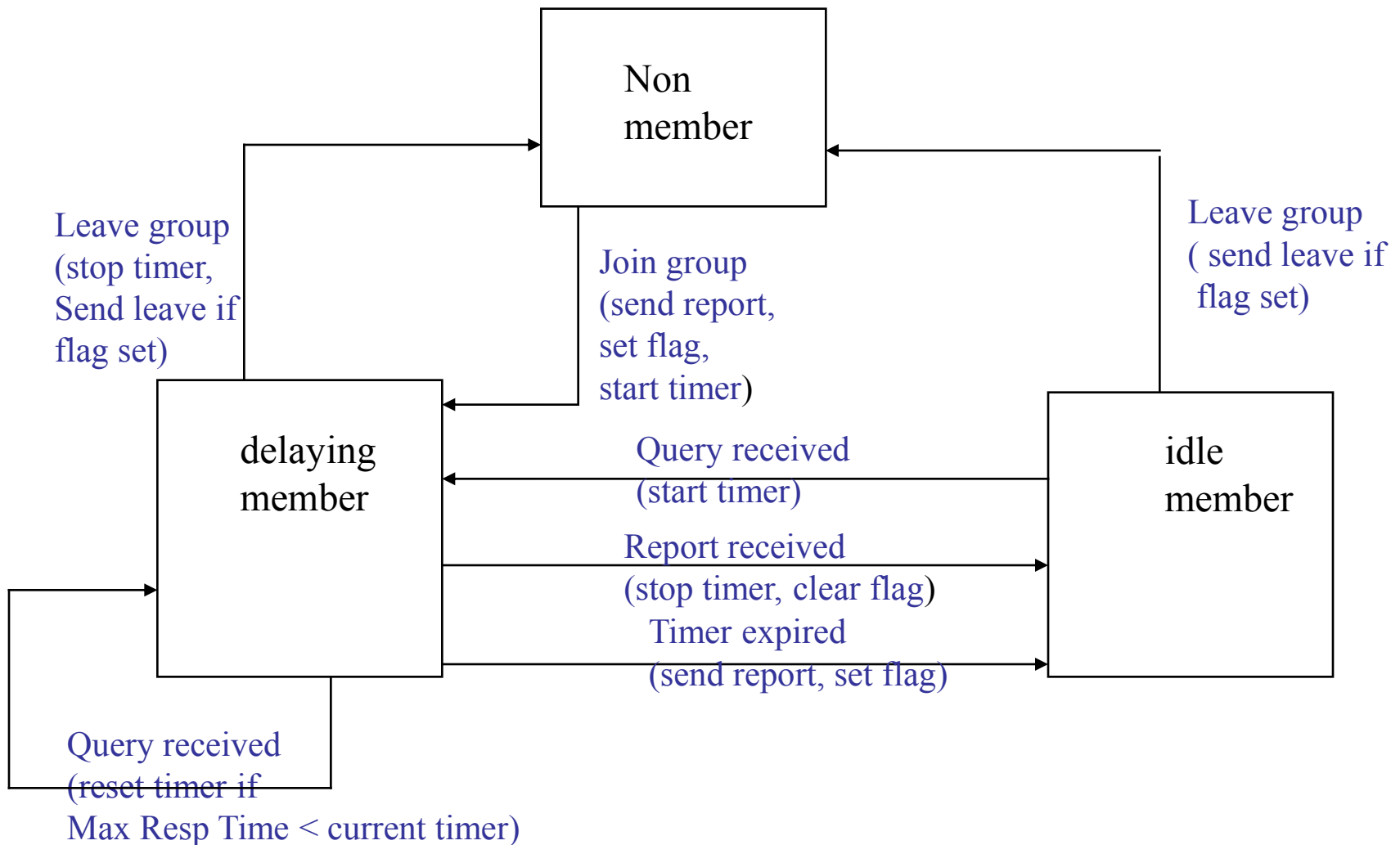
- Si plusieurs routeurs sur le même réseau local :
  - Un seul routeur est élu pour émettre des *Queries* : Designated Router (DR)
  - En IGMPv1, le mécanisme d'élection devait être réalisé par le routage multicast et non par IGMP
  - En IGMPv2, le DR sera celui avec l'adresse IP la plus petite
- Désabonnement:
  - Introduction d'un message *Leave Group*
    - Message envoyé à l'adresse 224.0.0.2 (all-routers multicast group) si le hôte est le dernier membre (pas obligatoire sinon)

# Désabonnement

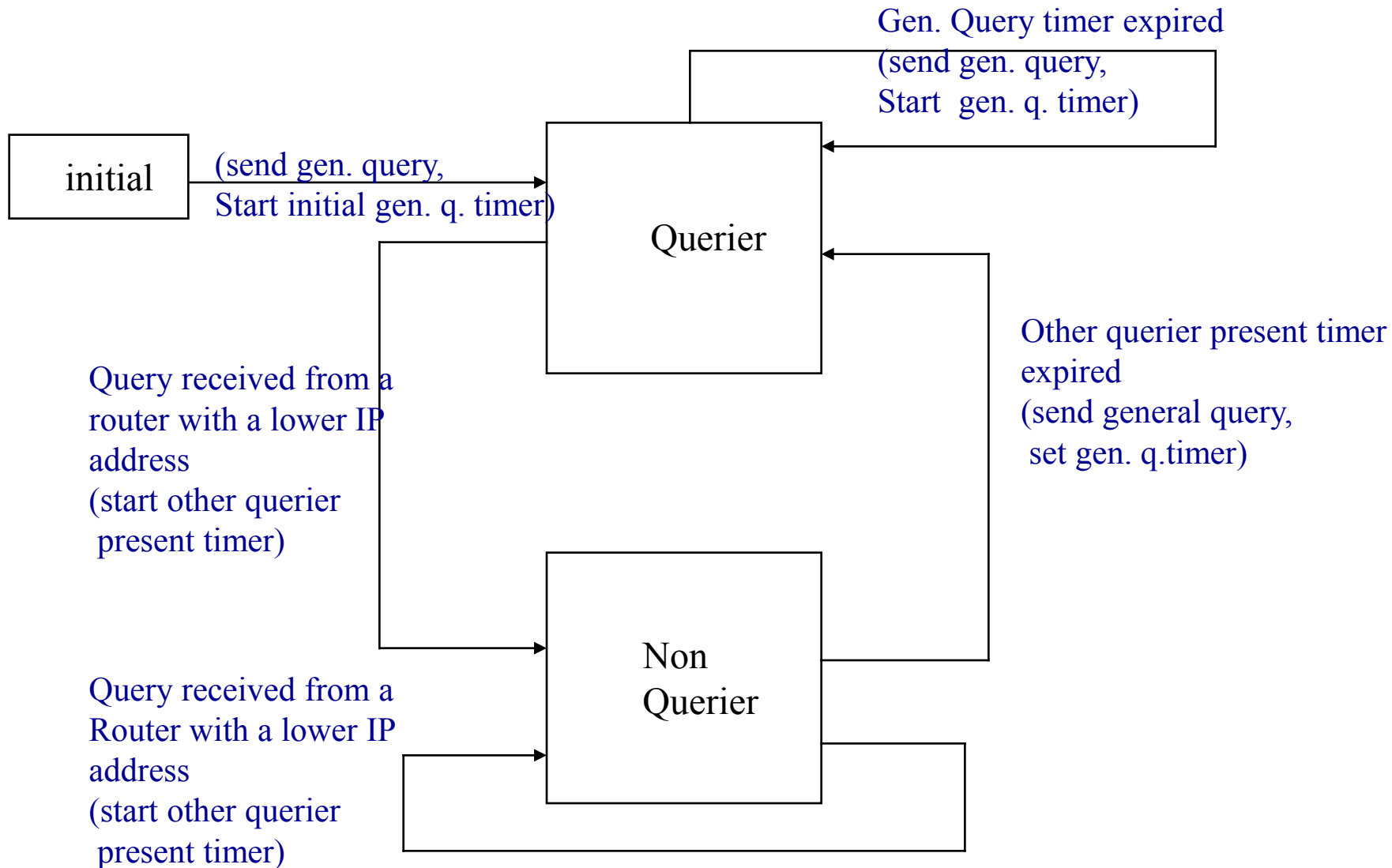
Envoi Report pour  
224.y.y.y à 224.0.0.2



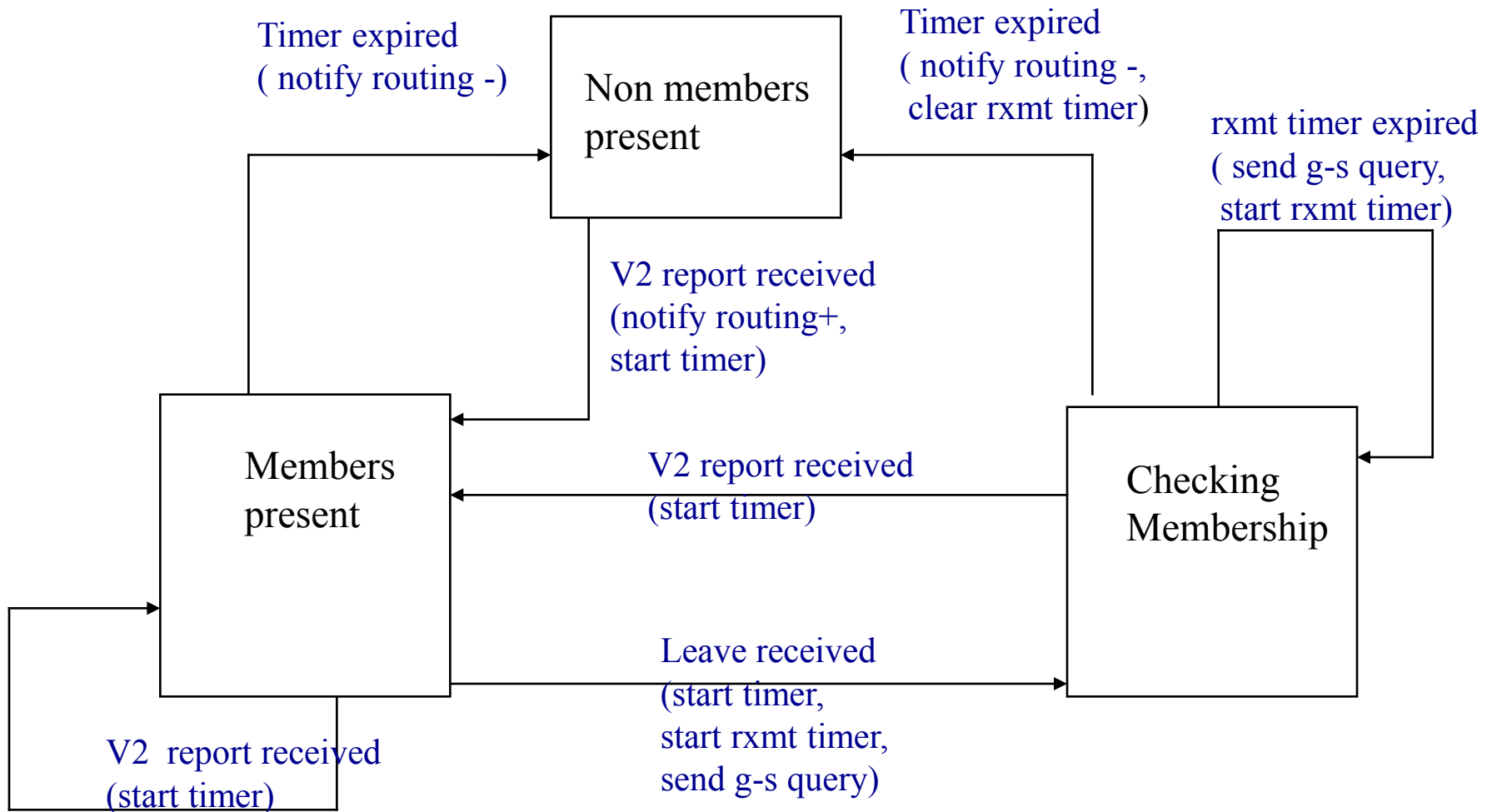
# Diagramme état : IGMP hôte



# Diagramme état : IGMP routeur

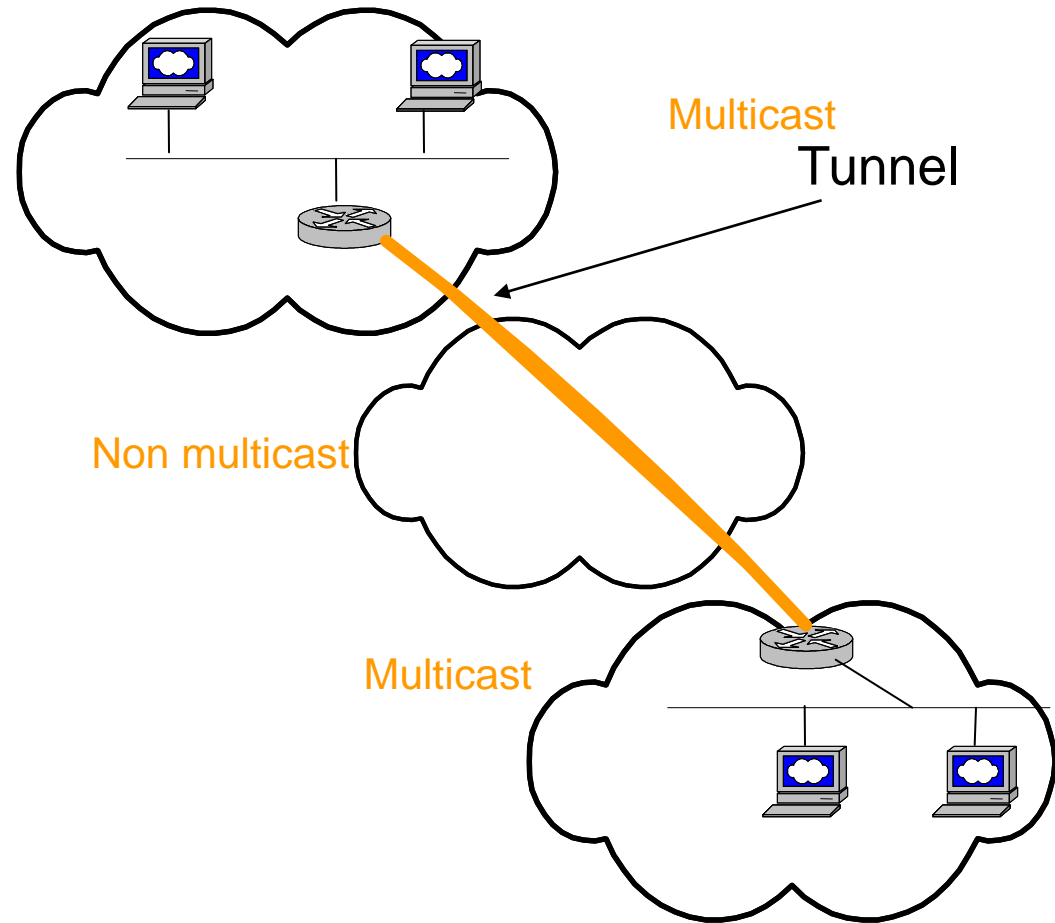


# Diagramme état : IGMP routeur



# IP Multicast et le Mbone

- Mbone = Virtual Internet Backbone for Multicast IP
- Relier des "îlots multipoints" (zone où le routage multipoint est supporté) par des tunnels.



# Initialisation d'un socket Multicast récepteur (R)

```
#define GROUP "239.137.194.111"
#define PORT 55555

int sdr;

/* création socket */
sdr = socket(AF_INET, SOCK_DGRAM, 0);
if (sdr < 0) {perror("problème création socket"); exit(1);}

struct sockaddr_in addr_r;
/* Initialisation structure adresse réception */
memset (&addr_r, 0, sizeof(addr_r));
addr_r.sin_family = AF_INET;
addr_r.sin_port = htons(PORT);
addr_r.sin_addr.s_addr = htonl(INADDR_ANY);
```



# Rejoindre un groupe Multicast (R)

```
struct ip_mreq imr;  
  
/* Initialisation structure imr */  
imr.imr_multiaddr.s_addr = inet_addr(GROUP);  
imr.imr_interface.s_addr = INADDR_ANY  
  
/* Demande de participation à un groupe */  
if (setsockopt (sdr,IPPROTO_IP,IP_ADD_MEMBERSHIP, &imr, sizeof(imr))<0){  
    perror("probleme setsockopt");  
    exit(1);  
}  
  
unsigned int on = 1;  
/* En cas de réutilisation d'un port */  
if (setsockopt (sdr,SOL_SOCKET, SO_REUSEADDR, &on,sizeof(on)) < 0)  
    {perror("probleme setsockopt ");  
    exit(1);  
}  
  
/* Nommer et lier le socket en reception */  
if (bind (sdr, (struct sockaddr *)&addr_r, sizeof(addr_r)) < 0) {  
    perror("probleme setsockopt ");  
    exit(1);  
}
```

# Réception d'un message Multicast (R)

```
int cnt, len_r=sizeof(addr_r);
char buf[100];

/* Réception de paquets */
/* Utilisation de la primitive recvfrom */

cnt = recvfrom (sdr, buf, sizeof(buf), 0,
               (struct sockaddr *)&addr_r, &len_r)
if (cnt < 0) {
    perror("probleme recvfrom");
    exit(1);
}
```

# Quitter un groupe Multicast (R)

```
/* Demande pour abandonner un groupe */  
  
if (setsockopt (sdr, IPPROTO_IP, IP_DROP_MEMBERSHIP,  
    &imr, sizeof(imr)) < 0) {  
    perror("probleme setsockopt ");  
    exit(1);  
}
```

# Initialisation d'un socket Multicast émetteur (E)

```
#define GROUP "239.137.194.111"
#define PORT 55555

int sds;

sds = socket(AF_INET, SOCK_DGRAM, 0);
if (sds < 0) {perror("problème création socket"); exit(1);}

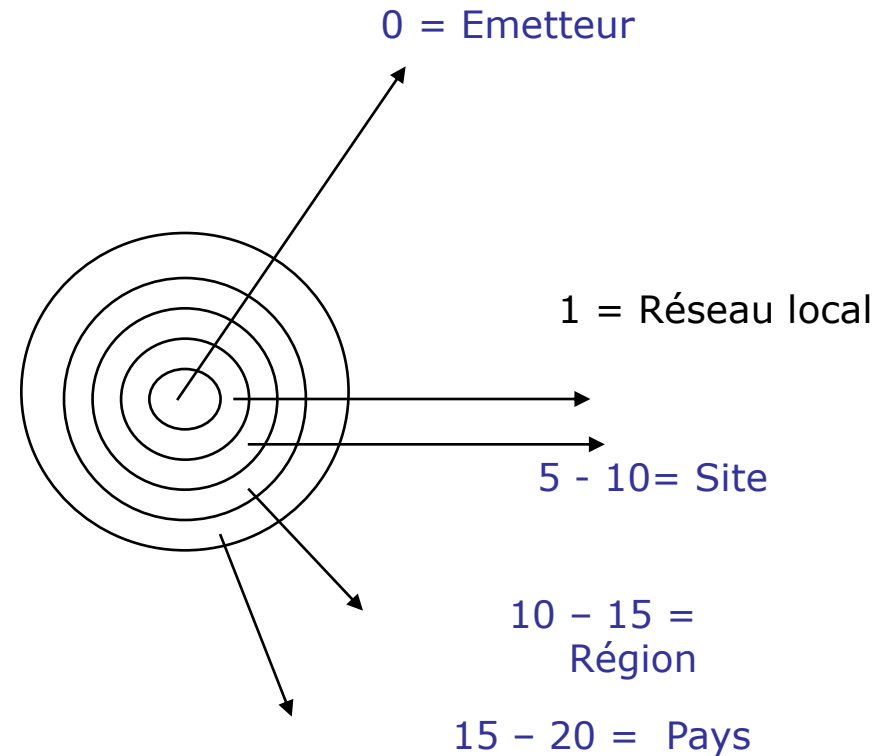
struct sockaddr_in addr_s;
/* Initialisation structure adresse émission */
memset (&addr_s, 0, sizeof(addr_s));
addr_s.sin_family = AF_INET;
addr_s.sin_port = htons(PORT);
addr_s.sin_addr.s_addr = inet_addr(GROUP);
```

# Portée d'un message Multicast (E)

```
unsigned char ttl = 1;

/* Fixer la portée du message */
/* Fixer la valeur du TTL */

if (setsockopt (sds, IPPROTO_IP,
IP_MULTICAST_TTL,
&ttl, sizeof(ttl)) < 0) {
    perror("probleme setsockopt");
    exit(1);
}
```



# Emission d'un message Multicast (E)

```
int cnt, len_s=sizeof(addr_s);
char buf[100];
/* Emission de paquets */
/* Utilisation de la primitive sendto */

cnt = sendto (sds, buf, strlen(buf)+1, 0,
              (struct sockaddr *)&addr_s, len_s)
if (cnt < 0) {
    perror("problème sendto");
    exit(1);
}
```

# Plan

- ① Contrôle de congestion et TCP
- ② Programmation réseaux
  - socket, select(), options de socket, socket raw
- ③ Communication de groupe (multicast)
- ④ Introduction IPv6