3ème section : couche transport

Nos objectifs:

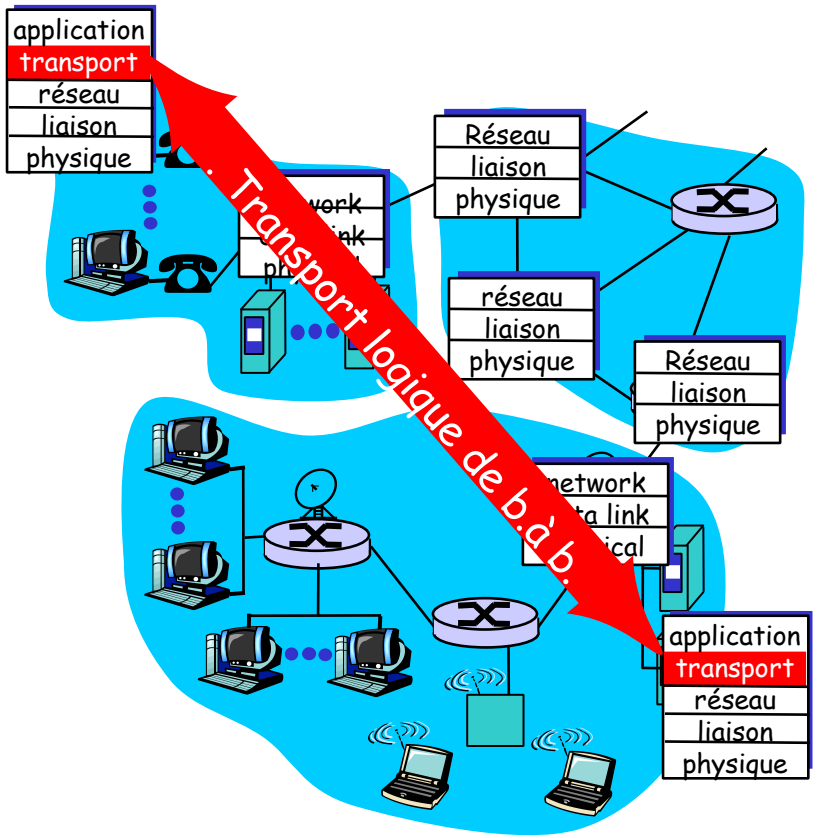
- Comprendre les principes qui sous-tendent les services de la couche transport :
  - Multiplexage/ démultiplexage
  - Transfert de données fiable
  - Contrôle de flot
  - Contrôle de congestion
- Mise en application et implantation dans l'Internet

Survol:

- Services de la couche transport
- multiplexage/démultiplexage
- Transport sans connexion : UDP
- Principes de transfert de données fiable
- Transport orienté connexion : TCP
  - Transfert fiable
  - Contrôle de flot
  - Gestion du contrôle
- Principes du contrôle de congestion
- Contrôle de congestion de TCP

Services et protocoles de transport

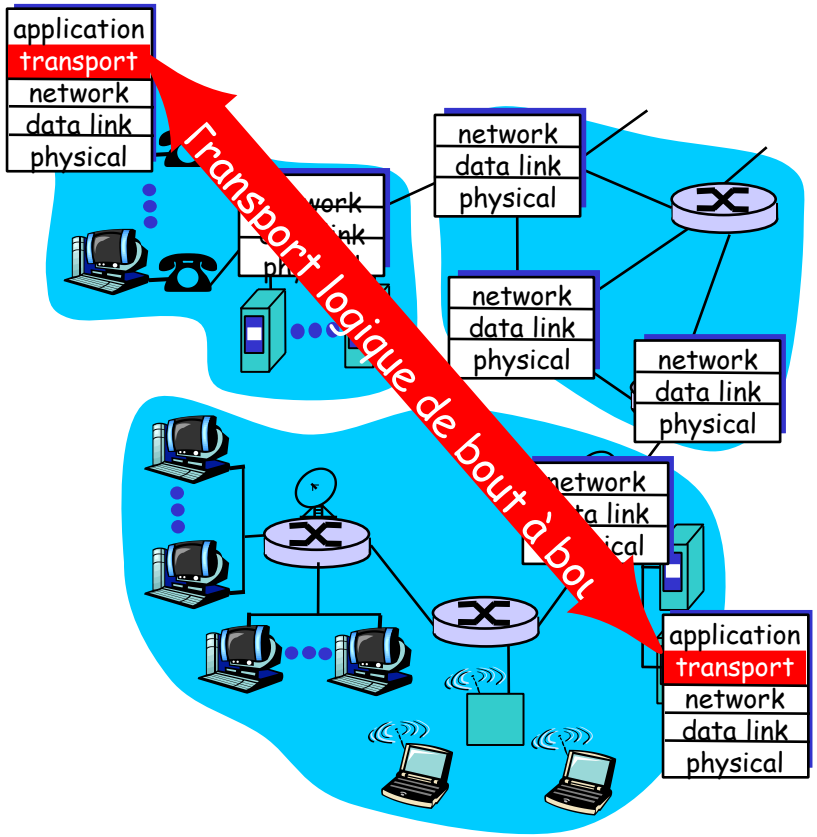
- *Fournissent une communication logique* entre des processus applications exécutés sur des machines différentes.
- Les protocoles de transport sont exécutés dans les noeuds terminaux.
- Services des couches transport ou réseau
- *Couche réseau:* transfert de données entre systèmes terminaux
- *Couche transport:* transfert de données entre processus
  - Repose sur, et améliore, les services de la couche réseau



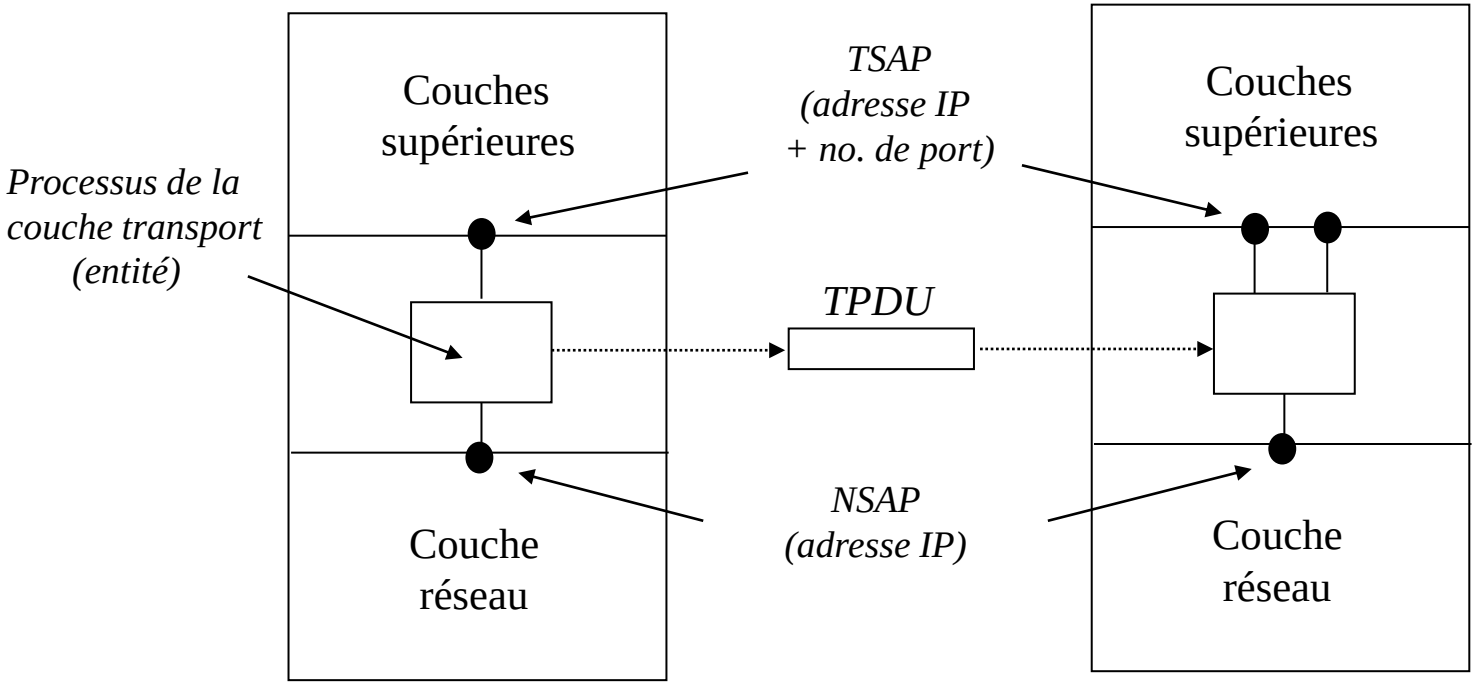
Protocoles de la couche transport

Services de transport de l'Internet :

- Fiable, livraison ordonnée en «unicast» (TCP)
  - Congestion
  - Contrôle de flot
  - Etablissement de connexion
- Non fiable («au mieux»), «unicast» ou «multicast» non ordonné: UDP
- Services non disponibles :
  - Temps-réel
  - Garanties de bande passante
  - Multicast fiable



Communication pair-à-pair

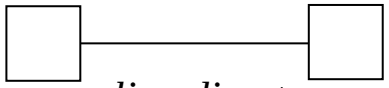


Services de base
exemple plus simple que BSD

Primitive	Message (TPDU) envoyé	Signification
LISTEN	*aucun*	Le processus est bloqué jusqu'à ce qu'un autre processus se connecte
CONNECT	Requête de connexion	Le processus tente d'établir une connexion
SEND	Données	Transmission de données
RECEIVE	*aucun*	Le processus est bloqué jusqu'à ce qu'une TDPU de données arrive.
DISCONNECT	Requête de déconnexion	Le processus veut libérer la connexion

Couche 4 vs couche 2

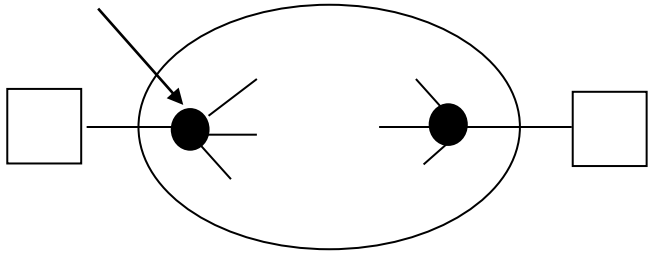
Couche 2
(Liaison de données)



lien direct

Couche 4
(Transport)

Routeur



réseau

Le lien physique n'a pas
de capacité de mémoire

Un seul canal à gérer
→ une seule file d'attente
pour les retransmissions

Le réseau (ensemble des routeurs)
a une grande capacité de mémoire
(et de délais associés)

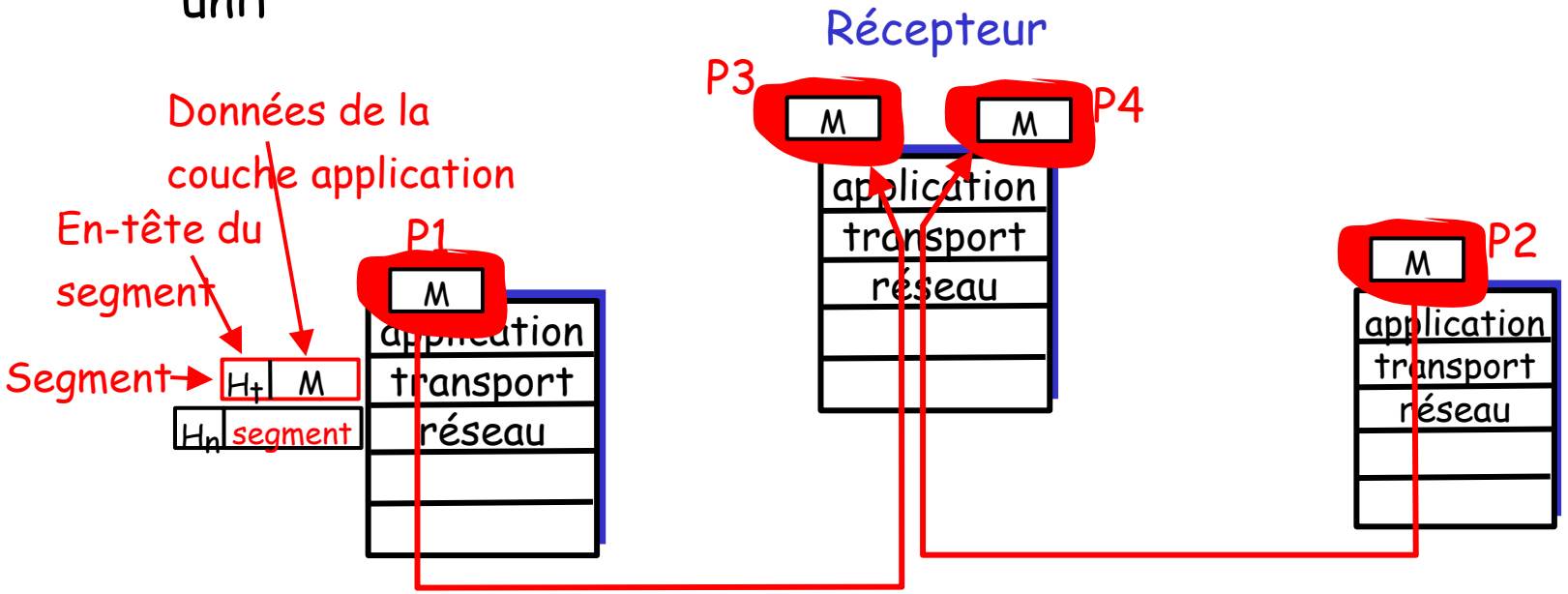
Plusieurs connexions simultanées possibles
→ plusieurs files d'attente
pour les retransmissions

Multiplexage/démultiplexage

Rappelons que: un *segment* est une unité de donnée échangée entre entité de transport

- Appelé également TPDU: transport protocol data unit

Démultiplexage: livrer les segments reçus au bon processus de la couche application

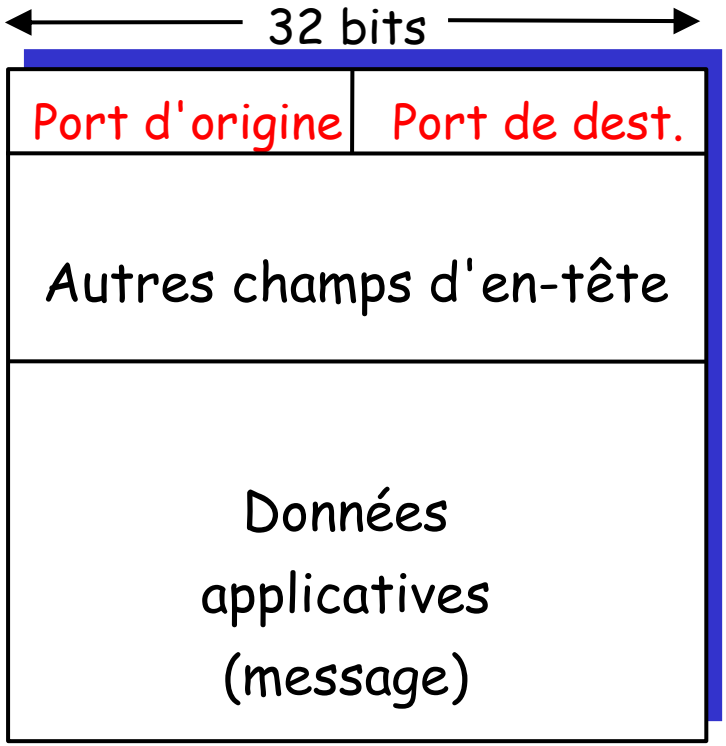


Multiplexage/démultiplexage

Multiplexage:
Rassembler des données de plusieurs applications, et les envelopper avec un en-tête, qui sera utilisé pour le démultiplexage.

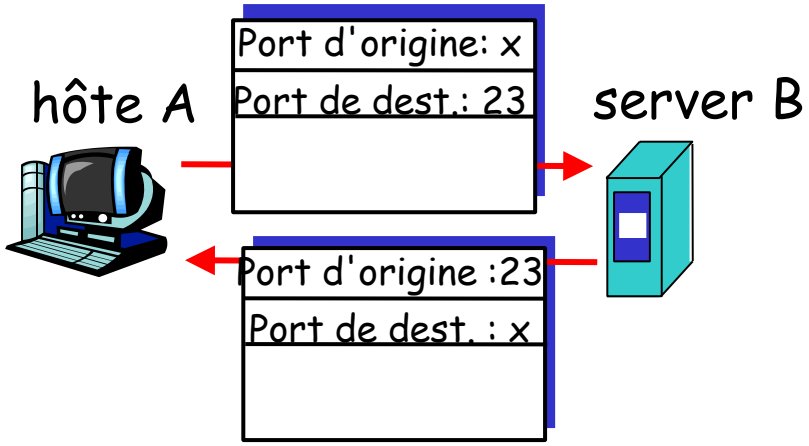
Multiplexage/démultiplexage:

- Basé sur l'expéditeur, le port de réception, l'adresseIP
  - Origine, port de destination dans chaque segment
  - Souvenons-nous: des ports «bien connus» pour les applications



Format des segments TCP/UDP

Multiplexage/démultiplexage : exemples



Utilisation port: simple application telnet

Utilisation du port: serveur Web

UDP: User Datagram Protocol [RFC 768]

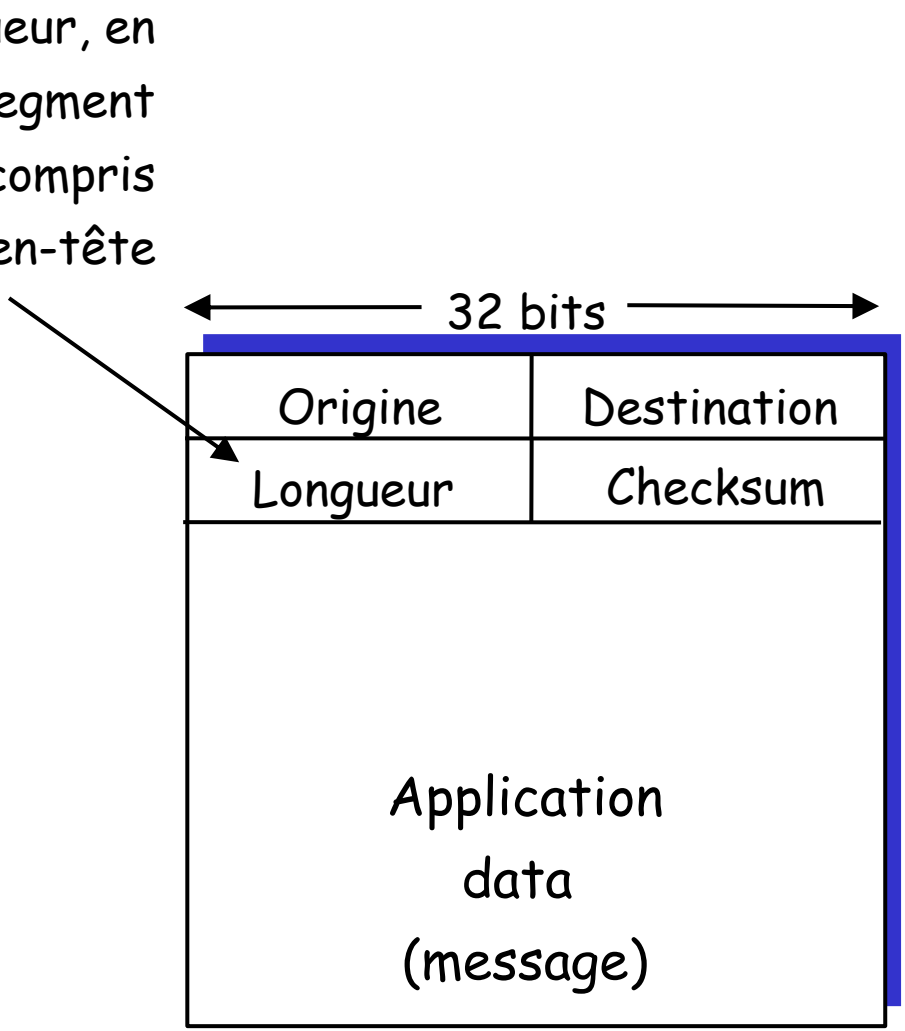
- Protocole de transport Internet minimaliste
- Service «au mieux», les segments UDP peuvent être:
  - Perdus
  - Délivrés dans le désordre
- *Sans connexion:*
  - Pas de poignée de main entre l'expéditeur et le récepteur UDP
  - Chaque segment UDP est traité séparément des autres

Pourquoi UDP existe-t-il?

- Pas d'établissement de connexion (lequel peut ajouter un délai)
- Simplicité : pas d'état de connexion à l'expéditeur ou au récepteur
- Petit en-tête
- Pas de contrôle de congestion : UDP peut transmettre aussi rapidement que nécessaire

UDP: suite

- Souvent utilisé pour les applications multimédia
  - - Résistance aux pertes
  - + Sensitivité au débit
- Autres utilisation d'UDP (mais pourquoi ?):
  - DNS
  - SNMP
- Transfert fiable par dessus UDP: ajouter la fiabilité à la couche application
  - Récupération d'erreur propre à l'application!



Format des segments UDP

«checksum»UDP

But: détecter des erreurs (p.ex. des bits modifiés) dans le segment transmis

Expéditeur:

- Prend le segment comme une séquence de nombres de 16 bits
- checksum: addition (en compléments à 1) du contenu du segment
- L'expéditeur place le résultat dans le champs correspondant

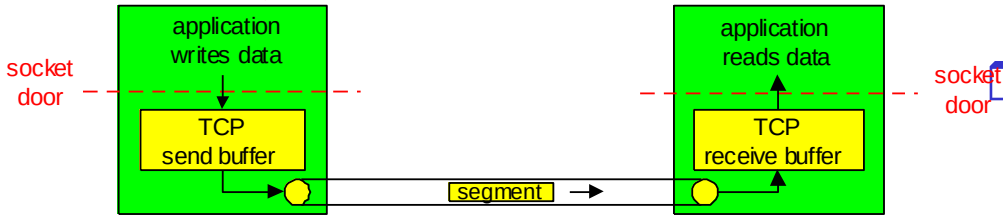
Récepteur:

- Calcule le checksum du segment reçu
- Compare le résultat à la valeur reçue :
  - Différence – erreur détectée
  - Identité – pas d'erreur détectée. Mais n'y en a-t-il vraiment aucune? *A suivre.*

TCP: Survol

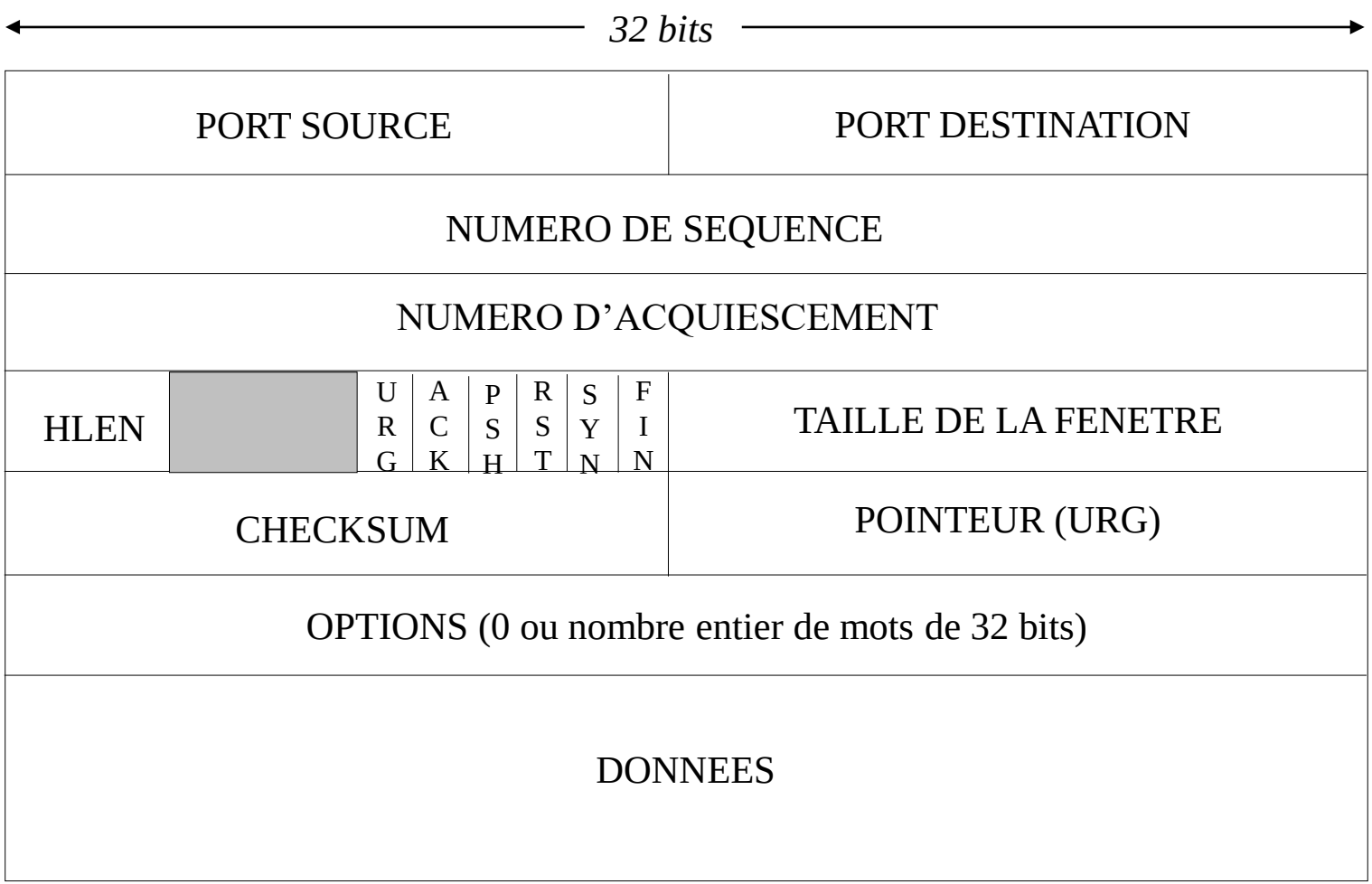
RFCs: 793, 1122, 1323, 2018, 2581

- Point-à-point:
  - Un expéditeur, un récepteur
- Flot d'octets fiable, ordonné:
  - Pas de limites de délimiteurs
- En pipeline:
  - Le contrôle du flot et de la congestion établissent la fenêtre.
- *Tampons d'émission et de réception.*



- Données transmises en full duplex :
  - Flot de données bi-directionnel sur la même connexion
  - MSS: « maximum segment size »
- Orienté connexion:
  - Poignée de main (échange de messages de contrôle) initie l'état de l'émetteur et du récepteur avant échange de données.
- Contrôle de flot :
  - L'expéditeur ne peut pas saturer le récepteur

Le segment TCP



Les champs du segment TCP

- **PORT SOURCE** : (**16 bits**)
  - → porte logique associée à la connexion locale de la source
  - → peut être alloué explicitement (si > 255)
  - → Adresse IP + port = TSAP unique (48 bits au total)

- **PORT DESTINATION** : (**16 bits**)
  - → porte logique associée à la connexion locale de la destination

- **NUMERO DE SEQUENCE** : (**32 bits**)
  - → Indique la position du premier octet du champs de données du segment TCP dans le message (par exemple, un fichier)

- **NUMERO D'ACQUIESCEMENT** : (**32 bits**)
  - → Indique le numéro du prochain octet attendu au récepteur (et non le numéro du dernier octet bien reçu)

Les champs du segment TCP (suite)

- **HLEN** : (**4 bits**)
  - → longueur de l'entête TCP (en mots de 32 bits)
  - → requis puisque le champs OPTIONS est à longueur variable

- **---** : (**6 bits**)
  - → 6 bits non-utilisés (réservés pour de futures modifications qui ne se sont jamais avérées nécessaires)

- **URG** : (**1 bit**)
  - → utilisé conjointement avec le champs POINTEUR (URG)
  - → ce bit est mis à 1 pour signifier que des données doivent être transmises immédiatement
    (rappelons que TCP ne préserve pas nécessairement les divisions entre les messages)

Les champs du segment TCP (suite)

- **ACK** : **(1 bit)**
  → mis à 1 si le champs NUMERO D'ACQUIESCEMENT est valide

- **PSH** : **(1 bit)**
  → indique une information urgente au receveur
  → le receveur doit passer les données reçues immédiatement à la couche application

- **RST** : **(1 bit)**
  → utilisé après un problème majeur (ex.: crash d'une machine)
  → utilisé aussi pour indiquer le refus d'un segment invalide ou d'une tentative de connexion.
  → en général, ce bit indique un problème majeur…

Les champs du segment TCP (suite)

- **SYN** : (**1 bit**)
  - → utilisé lors de la connexion
  - → pour la requête (*connect*), on a SYN=1 et ACK=0
  - → si on accepte (*accept*) on a SYN=1 et ACK=1

- **FIN** : (**1 bit**)
  - → utilisé pour libérer la connexion
  - → signifie que l'émetteur n'a plus d'info à transmettre
  - → puisque l'ordre des paquets n'est pas garantit sur le réseau, le receveur doit examiner les numéros de segments (NUMERO DE SEQUENCE) pour s'assurer d'avoir reçu toute l'information de l'émetteur

Les champs du segment TCP (suite)

- **TAILLE DE LA FENETRE** : (**16 bits**)
  - → indique le nombre d'octets pouvant être transmis à partir du dernier octet acquiescé
  - → contrôlé par le receveur (contrôle de congestion)
  - → similaire à la fenêtre *W* dans les protocoles de liaison de données, sauf qu'ici on compte les octets directement
  - → un valeur de 0 est permise; elle indique que le receveur demande un repos
  - → en transmettant un acquiescement avec le même numéro et une valeur non-nulle de TAILLE DE LA FENETRE, le receveur peut permettre à l'émetteur de transmettre à nouveau

- **CHECKSUM** : (**16 bits**)
  - → bits de parité (sur l'entête et les données)
  - → somme complément-à-1 par mots de 16 bits

de séq. TCP et ACKs

Seq. #'s:

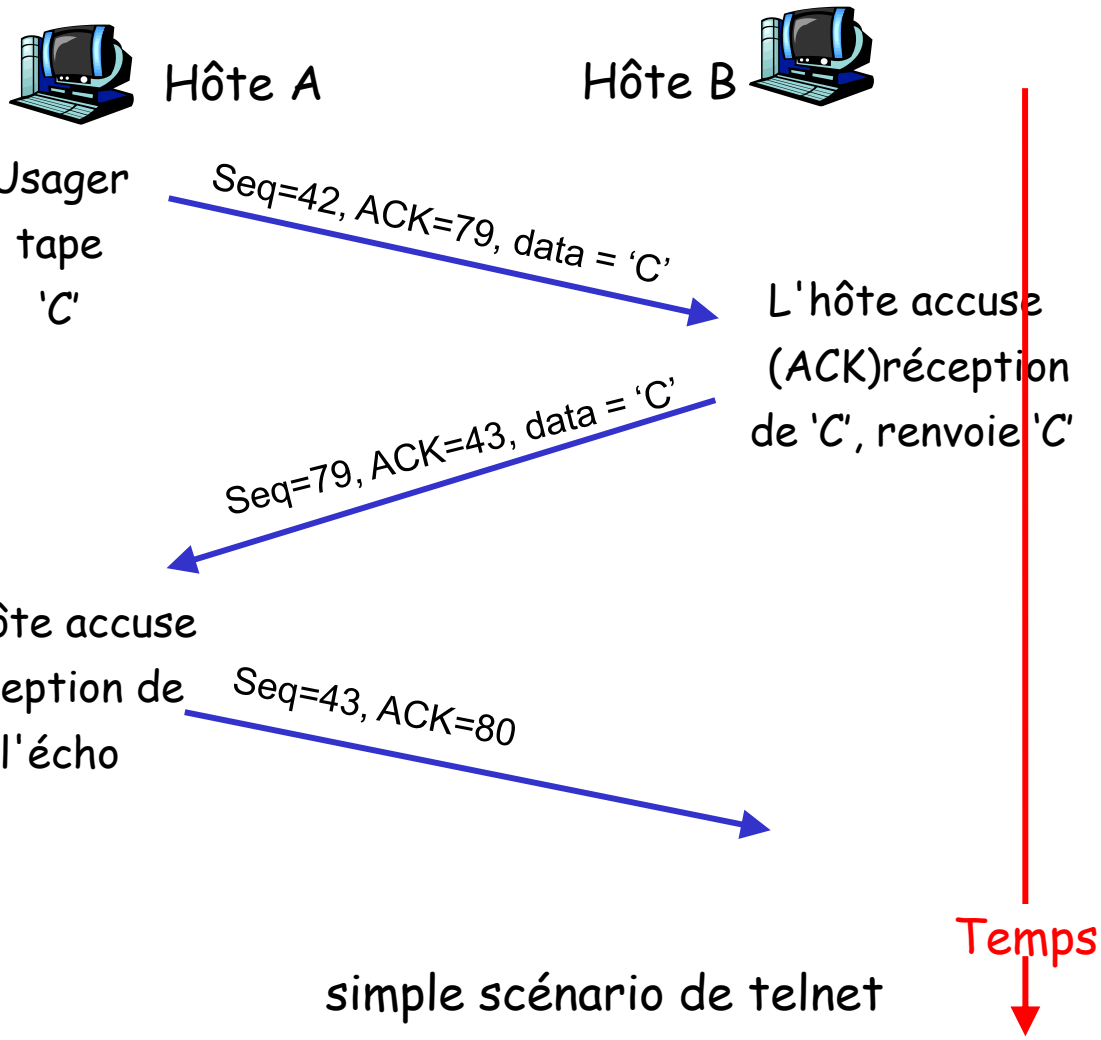
- Position du premier octet du segment dans le flot.

ACKs:

- # de séq de l'octet suivant que l'«autre» coté attend
- ACK cumulatif

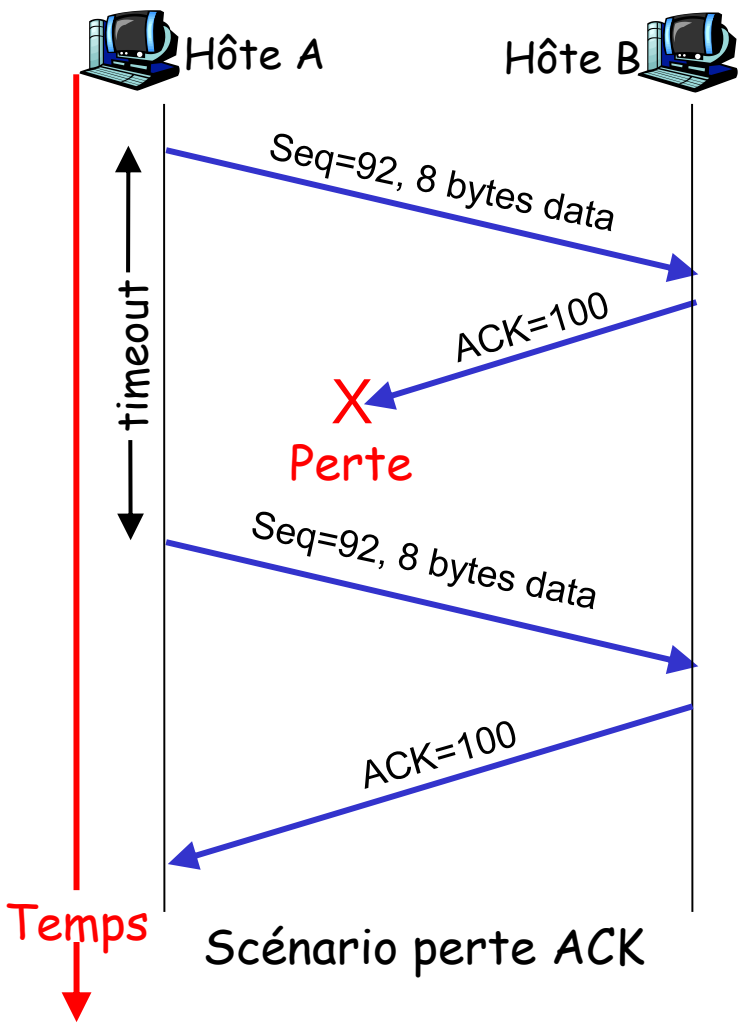
Q: Comment le récepteur traite-t-il les segments reçus en désordre ?

- R: Choix de l'implanteur.

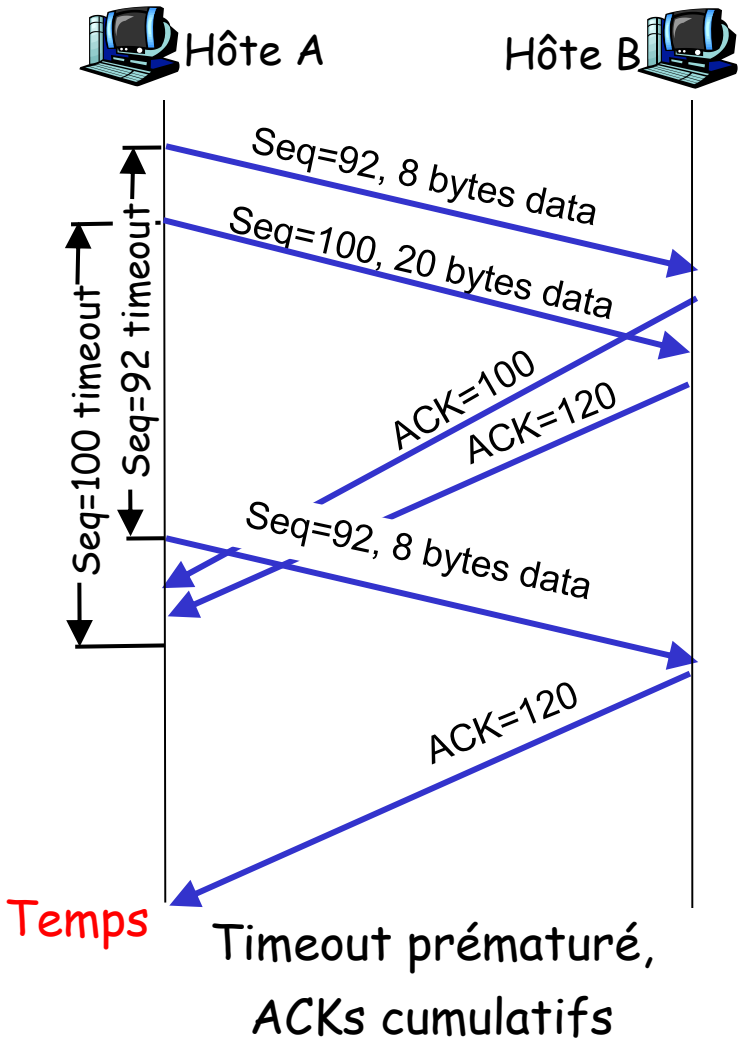


simple scénario de telnet

TCP: scénarios de retransmission



Scénario perte ACK



Timeout prématuré,
ACKs cumulatifs

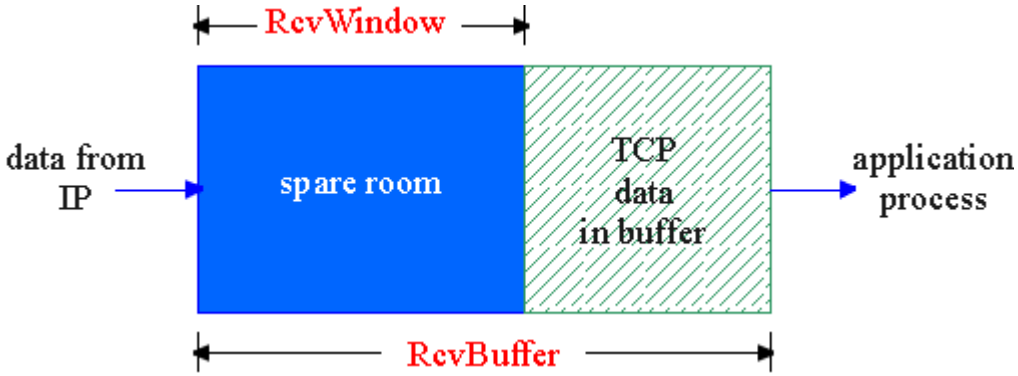
Contrôle de flot de TCP

Contrôle de flot

L'expéditeur ne peut pas déborder le récepteur en envoyant trop d'information trop rapidement

RcvBuffer = taille du tampon de réception

RcvWindow = espace disponible dans le tampon



Tampon au récepteur

récepteur: informe l'expéditeur des changements dans l'espace disponible.

- **RcvWindow**

expéditeur: conserve les données transmises, non acquittées, selon le plus récemment reçu **RcvWindow**

Gestion de la connexion de TCP

Rappelons que: l'expéditeur et le récepteur TCP, établissent une connexion avant d'envoyer des segments.

- Initialisation des variables TCP :
  - # de séq.
  - tampon, information de contrôle de flot (p.ex. **RcvWindow**)
- *client:* initie la connexion

```
Socket clientSocket = new
Socket("hostname","port number");
```

- *serveur:* contacté par le client

```
Socket connectionSocket =
welcomeSocket.accept();
```

Three way handshake:

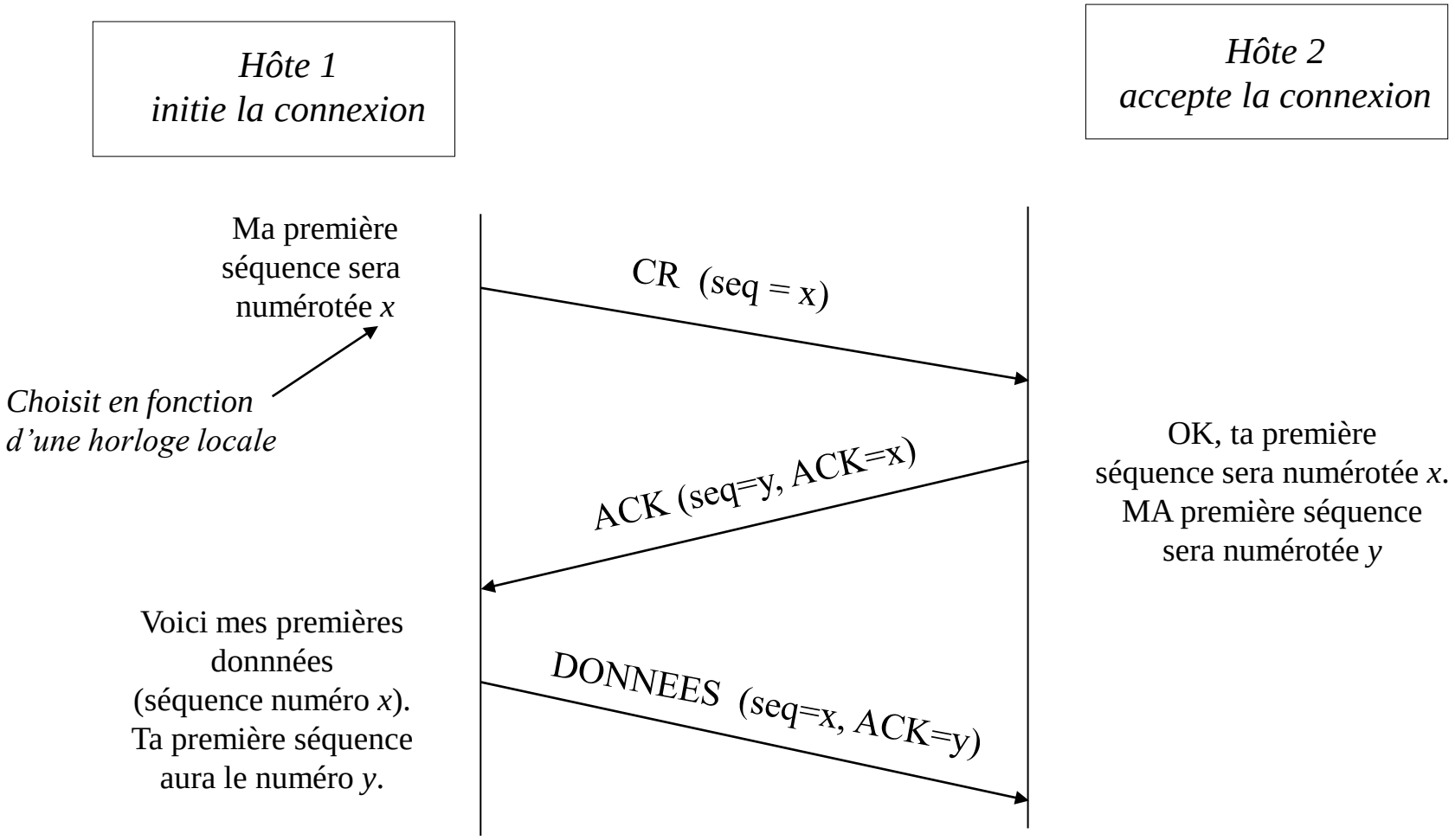
Etape 1: Le client envoie un segment de contrôle TCP SYN au serveur.

- Spécifie le # initial

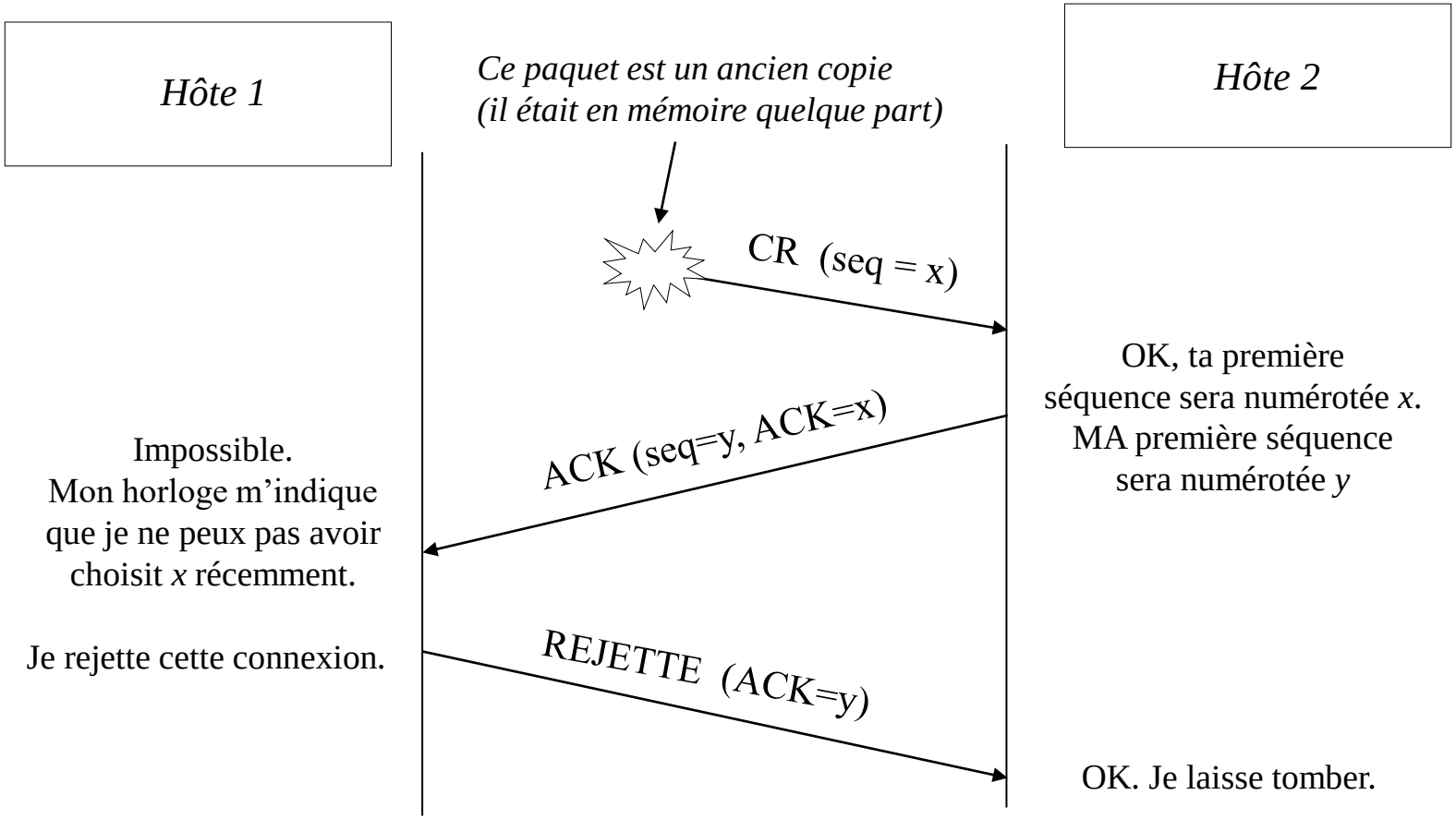
Etape 2: le serveur reçoit le SYN, répond avec un segment de contrôle SYN+ACK

- Acquitte le SYN reçu
- Alloue les tampons
- Spécifie le numéro de séquence initial dans son sens.

Etablissement de la connexion "three-way handshake"



"Three-way handshake" robustesse



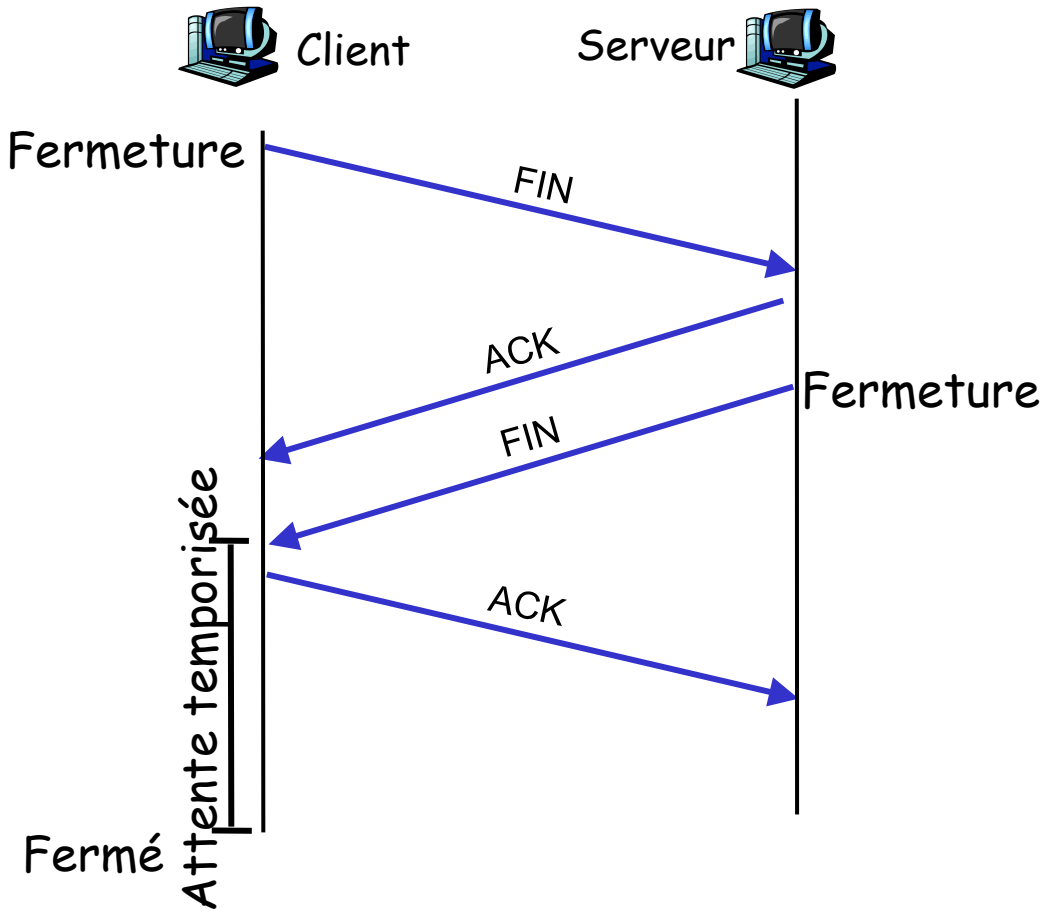
Gestion de la connexion TCP (suite)

Fermeture d'une connexion:

Le client ferme la prise :
clientSocket.close();

Etape1: le client envoye un segment de contrôle TCP FIN au serveur

Step 2: le serveur reçoit FIN, répond avec ACK. Il ferme la connexion et envoye FIN.



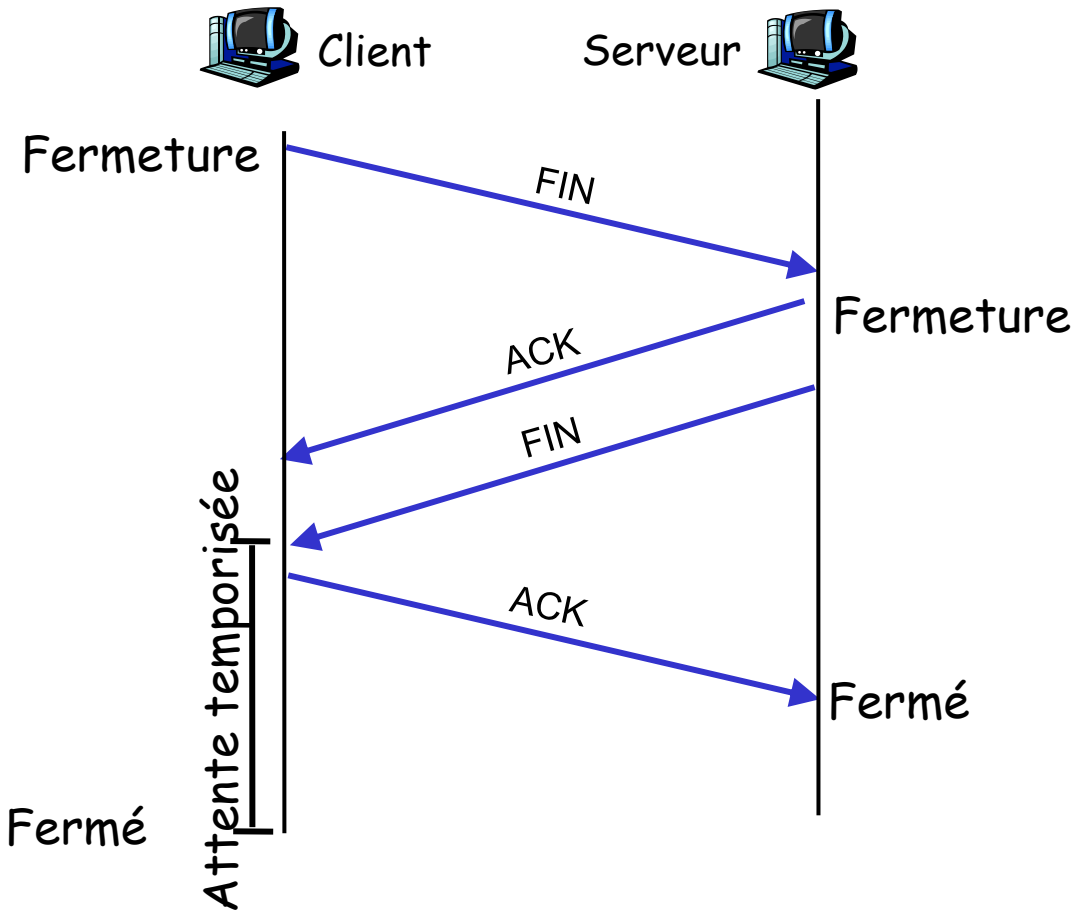
Gestion de la connexion TCP (suite)

Etape 3: le client reçoit FIN, répond avec ACK.

- Entre dans un était «attente temporisée» et répondra par ACK aux FINs reçus

Etape 4: le serveur, reçoit le ACK. Connexion fermée.

Note: avec de légères modifications, il est possible de traiter des FINs multiples.



Principes du contrôle de congestion

Congestion:

- Informellement: «trop de sources transmettent trop de données simultanément pour que le réseau puisse les absorber ».
- Donc différent du contrôle de flot!
- Manifestations:
  - Paquets perdus (débordement des tampons aux routeurs)
  - Long délais (mise en file dans les tampons des routeurs)
- Un problème du top-10!

Approches pour le contrôle de la congestion

Deux approches pour le contrôle de congestion :

Contrôle de congestion de bout à bout :

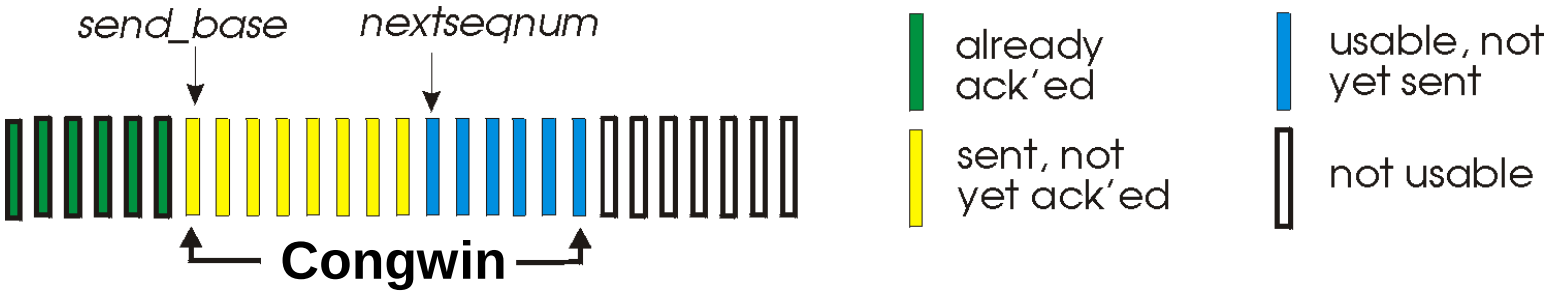
- Pas de feedback explicite du réseau
- Congestion déduite du délai et des pertes
- Approche retenue par TCP

Contrôle de congestion assisté par le réseau :

- Les routeurs donnent un feedback aux systèmes terminaux
  - Un simple bit indique la congestion (SNA, DECbit, TCP/IP ECN, ATM)
  - Taux d'émission pour l'expéditeur

Contrôle de congestion de TCP

- Contrôle de bout à bout, sans assistance réseau
- Le taux de transmission est limité par la taille de la fenêtre de congestion, **Congwin**, sur les segments:



- w segments, chacun avec MSS (Maximum Segment Size)

  octets envoyés dans chaque RTT (Round-Trip delay Time):

$$\text{Throughput} = \frac{w * MSS}{RTT} \text{ Octets/sec}$$

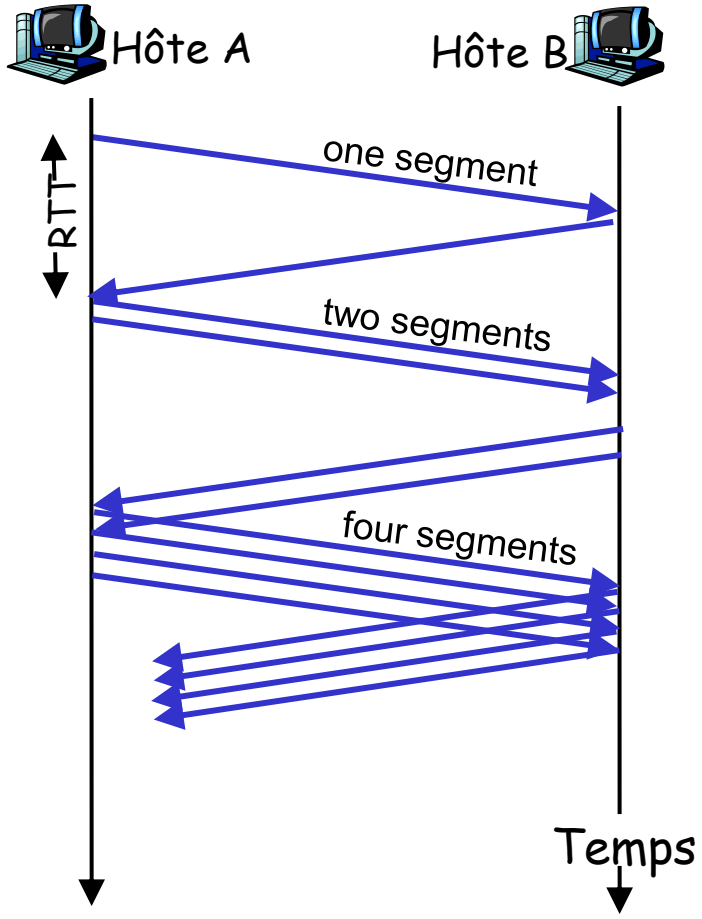
Contrôle de congestion de TCP :

- «sonde» pour une bande passante stable:
  - Dans l'idéal: transmettre aussi vite que possible (**CongWin** aussi large que possible) sans perte
  - *augmenter* **CongWin** jusqu'à perte (congestion)
  - perte: *décroître* **CongWin**, et puis tester (augmenter) à nouveau
- Deux «phases»
  - «slow start» (départ lent)
  - «congestion avoidance» (évitement de congestion)
- Variables importantes:
  - **CongWin**
  - **threshold** : définit la transition entre la phase de démarrage, et celle de congestion.

TCP «Slowstart»

Algorithme du Slowstart

initialize: CongWin = 1
for (each segment ACKed)
    CongWin++
until (loss event OR
    CongWin > threshold)

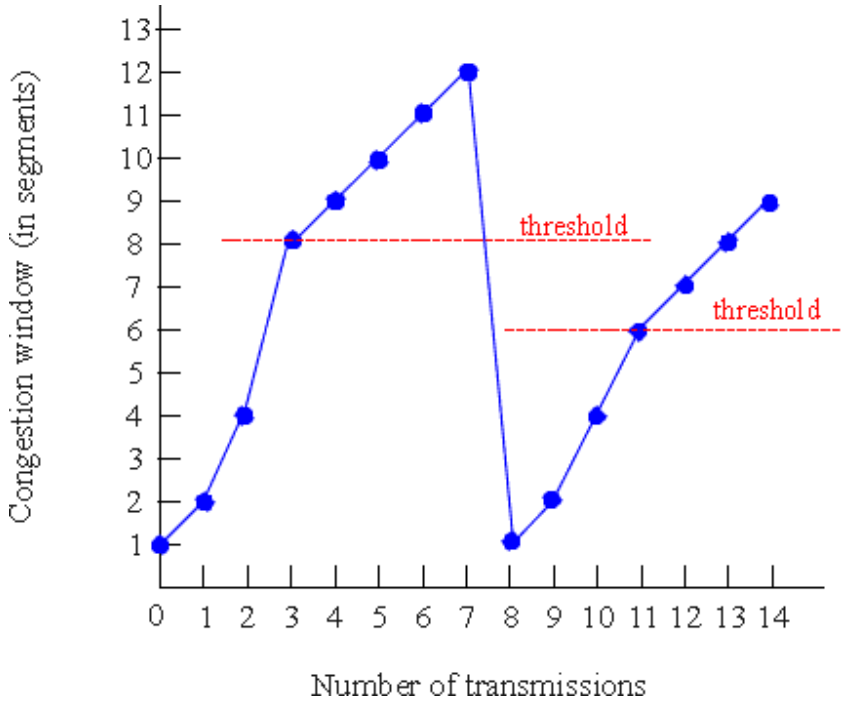


- Augmentation exponentielle (par RTT) en taille de fenêtre (pas si lent!)
- perte: timeout (TCP Tahoe) et/ou trois ACKs dupliqués (TCP Reno)

TCP Tahoe: « Congestion Avoidance »

TCP Tahoe Congestion avoidance

```
/* slowstart is over        */
/* CongWin > threshold */
Until (loss event) {
   every w segments ACKed:
        CongWin++
   }
threshold = CongWin/2
CongWin = 1
perform slowstart
```



TCP Congestion Avoidance: Reno

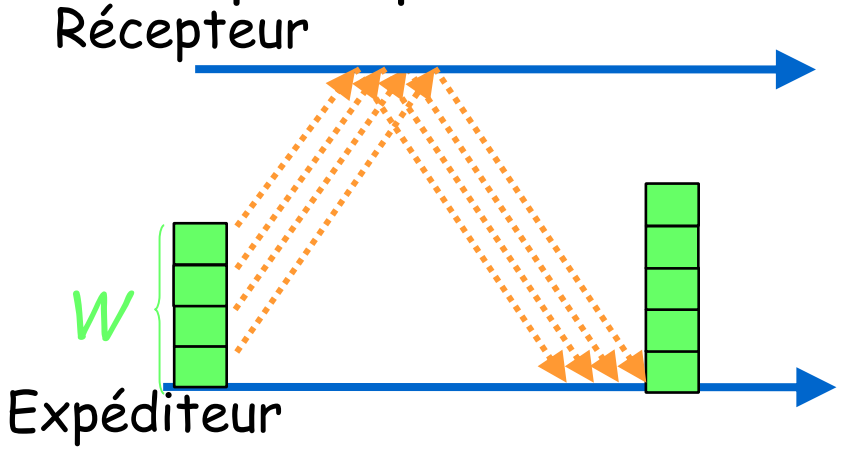
- Trois ACKs dupliqués (Reno TCP):
- Certains segments passent correctement !
- Ne pas réagir trop vite en amenant la fenêtre à 1 comme pour Tahoe
  - Décroitre la taille de la fenêtre de moitié

TCP Reno Congestion avoidance

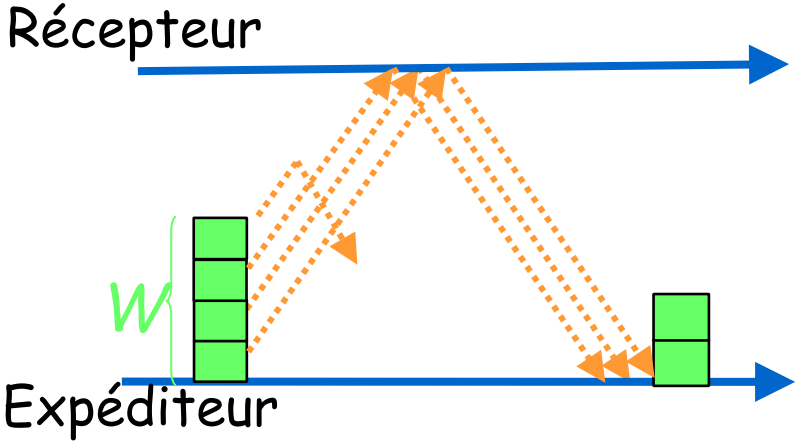
```
/* slowstart is over        */
/* CongWin > threshold */
Until (loss event) {
  every w segments ACKed:
      CongWin++
  }
threshold = CongWin/2
If (loss detected by timeout) {
    CongWin = 1
    perform slowstart }
If (loss detected by triple
duplicate ACK)
    CongWin = CongWin/2
```

Congestion Avoidance: Reno

- Augmenter la taille de la fenêtre par 1 par RTT sans perte: CongWin++



- Décroître la fenêtre de moitié si détection de perte par triple ACK dupliqués: CongWin = CongWin/2 *W* <- *W*/2



TCP Reno VS. TCP Tahoe:

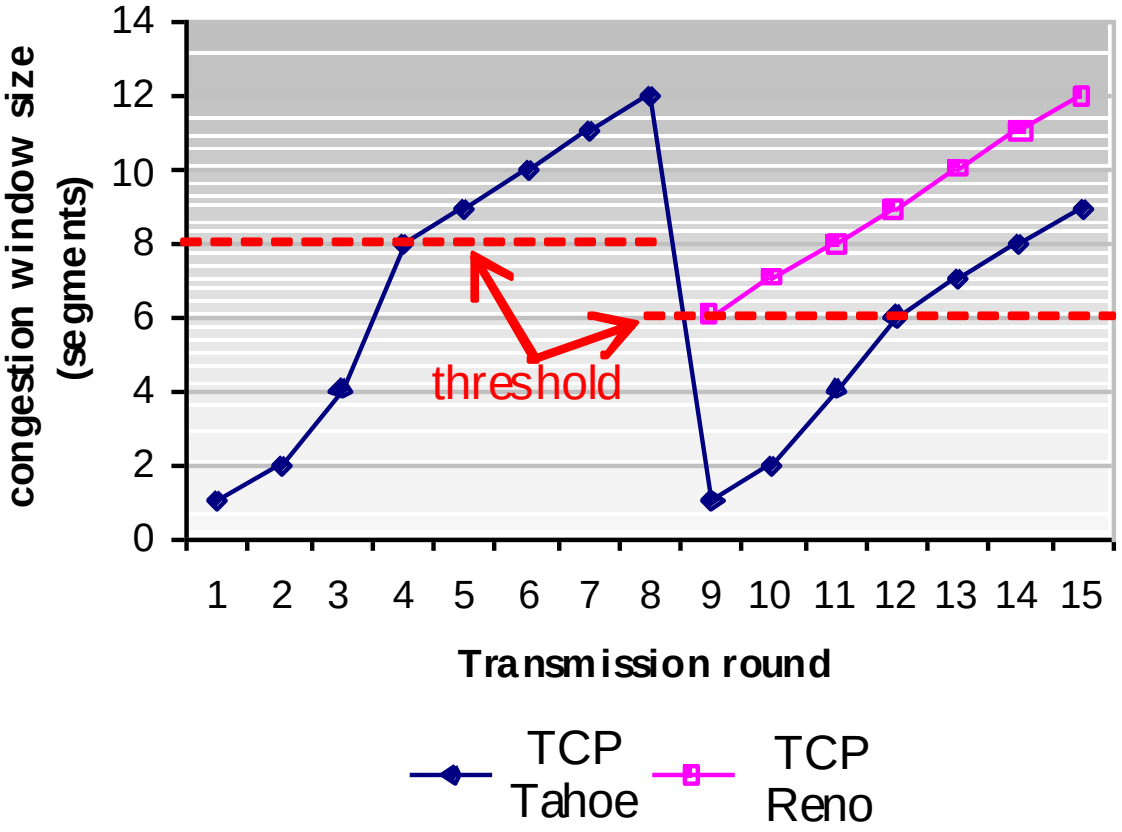


Figure 3.49 (revised): Evolution of TCP's Congestion window (Tahoe and Reno)

Chapitre 2: Résumé

- Principes sous les services de la couche de transport:
  - Multiplexage/démultiplexage
  - Transfert de données fiable
  - Contrôle de flot
  - Contrôle de congestion
- Instanciation and implémentation dans l'Internet
  - UDP
  - TCP