Systèmes et Réseaux (S5)
/ L3 Miage
Cours Réseaux / 2020-21

CM 4 :
Couche Transport

D'après le cours de Bruno Martin et les slides du livre "Computer Networking: A Top Down Approach, 6th edition, Jim Kurose, Keith Ross, Addison-Wesley, March 2012"

Ramon APARICIO-PARDO
Ramon.Aparicio-Pardo@unice.fr
(présenté par Michel Syska)

14/10/2020

PLAN CM 4

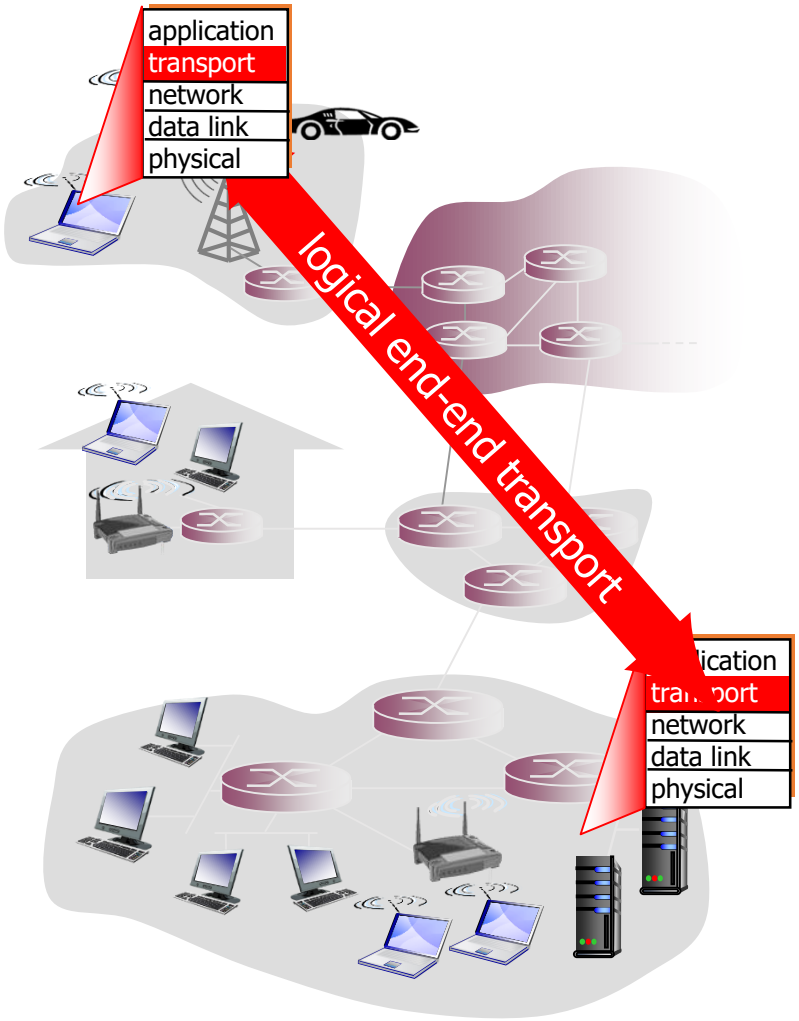
1. Couche Transport

2. TCP : Transmission Control Protocol

3. UDP : User Datagram Protocol

Services et protocoles de transport

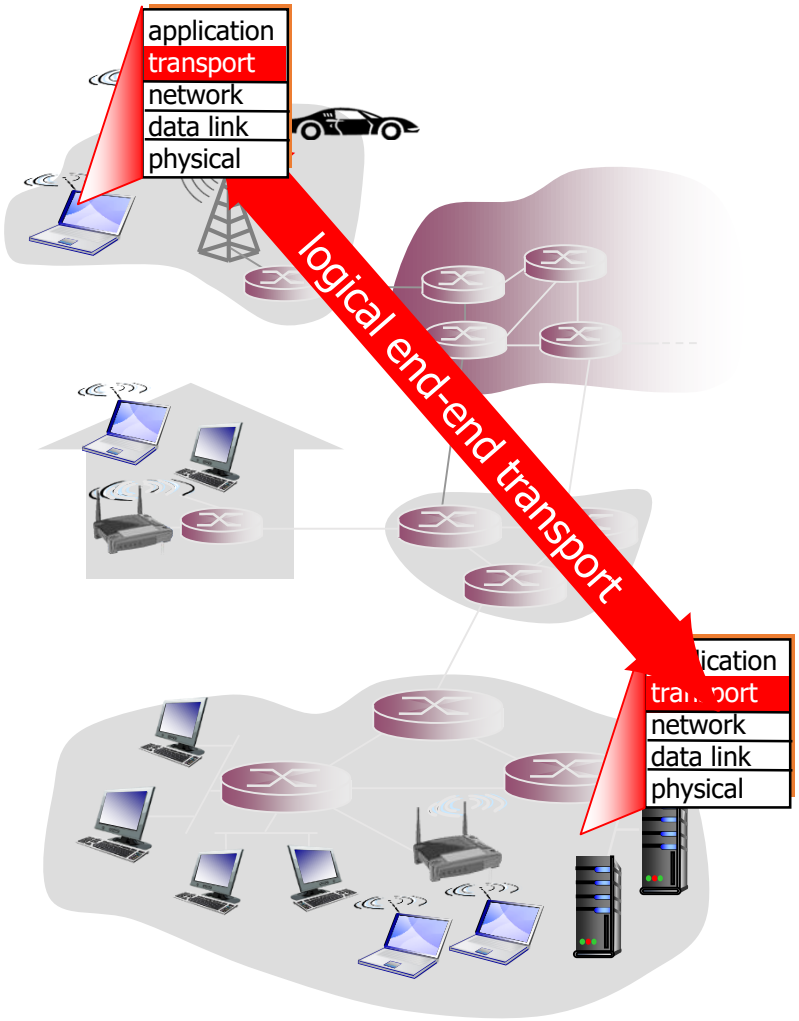
- Couche réseau : communication logique entre les **hôtes**
- Couche **transport** : communication logique entre les **processus** (sur des hôtes différents)
- Fonctionnent dans des systèmes aux extrémités :
  - Émetteur : les messages applicatifs sont fragmentés en segments, et passés ensuite à la couche réseau
  - Récepteur : les segments sont réassemblés en messages, et passés ensuite à la couche application



Services et protocoles de transport

IP : 2 protocoles disponibles

- **TCP** : transfert fiable, segments dans l'ordre
  - contrôle de congestion
  - contrôle de flux
  - établissement d'une connexion
- **UDP** : transfert non fiable, pas d'ordre
- Services non disponibles (ni dans UDP, ni dans TCP) :
  - pas de garantie de délais
  - pas de garantie de bande passante



Vue d'ensemble de TCP : Transmission Control Protocol

- Service fiable de transmission de données en mode connecté entre 2 machines :
  - Utilise IP (non fiable) pour la transmission réseau
  - TCP transporte environ 90% des octets sur Internet
  - RFC : 793, 1122, 1323, 2018, 2581

- Protocole de transport en mode connecté (orienté connexion) :
  - 3 phases de la connexion :
    - établissement de la connexion (triple poignée de main, « *3-way handshaking* »)
    - transfert fiable des données avec contrôle de flux / contrôle de congestion
    - double libération de la connexion
  - Les états d'émetteur et récepteur sont initialisés avant l'échange de données
  - Flux bidirectionnel de données (« full duplex ») dans la même connexion

Vue d'ensemble de TCP : Transmission Control Protocol

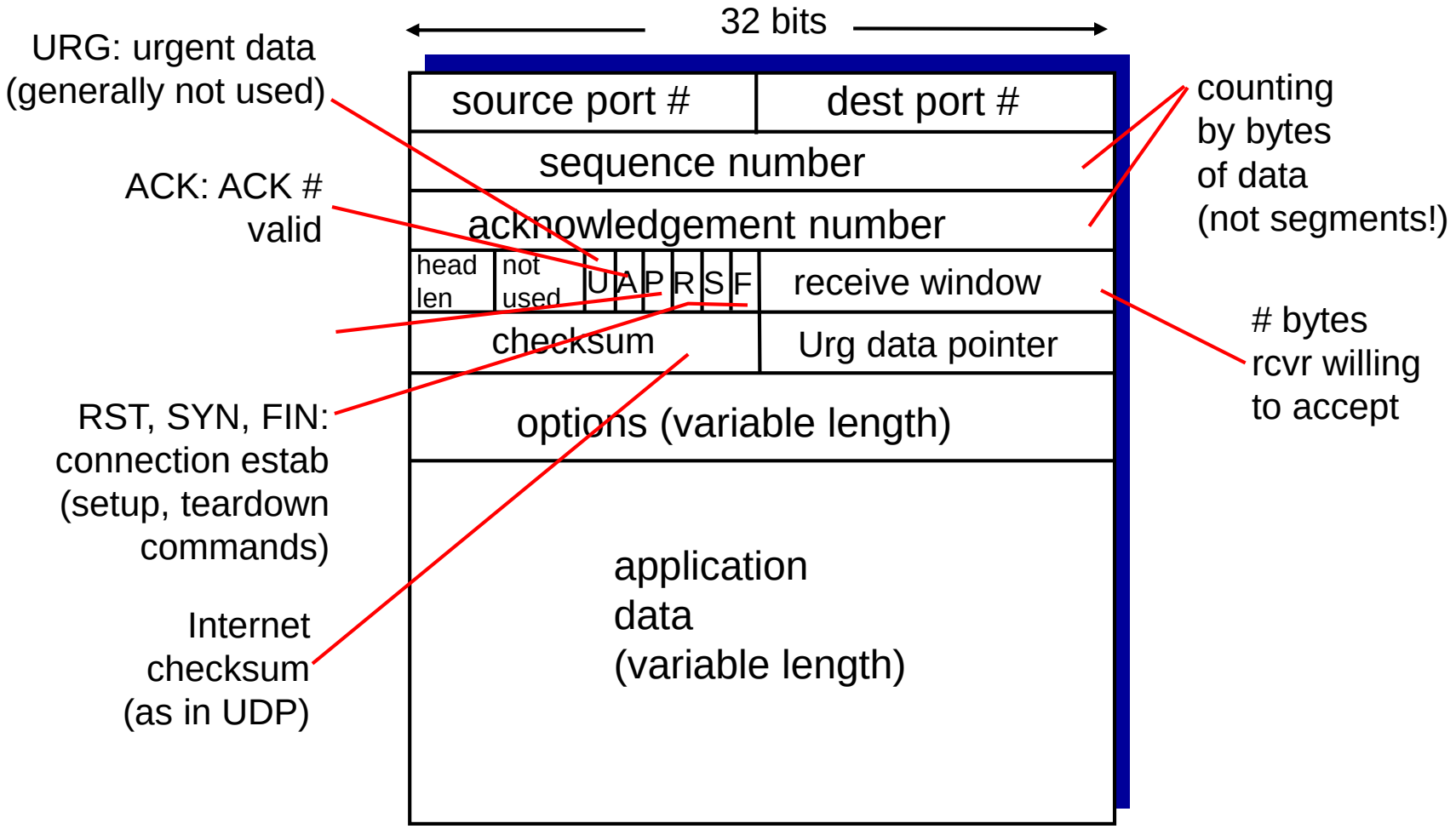
- Démultiplexage orienté connexion
  - Côte récepteur: Uniquement les segments avec les mêmes quatre valeurs (adresse IP source, nº port source) → (adresse IP dest, nº port dest) seront dirigés vers une même socket.

- Transfert d'un flux continu d'octets
  - Les messages applicatifs sont découpés en segments à cause de la maximum transmission unit (MTU) pouvant être transmise en une seule fois sur une interface.
    - p.ex., pour Ethernet (couche 2): 1500 octets
    - p.ex., pour IPv4 (couche 3): min MTU = 576 octets
    - p.ex., pour IPv6 (couche 3): min MTU = 1280 octets
  - Donc, la taille de segment maximale (MSS, Max. Seg. Size) est variable en fonction du MTU du lien à traverser.
  - TCP décide en général de lui-même les endroits où le flux de données doit être coupé
  - Typiquement dans un réseau IP, MSS = 536 octets = 576 – 20 - 20 (entêtes IP et TCP au min 20 octets)
    - Ce MSS réduit les chances de segmentation IP : tout le seg. TCP sera contenu dans un seul paquet IP non fragmenté

Vue d'ensemble de TCP : Transmission Control Protocol

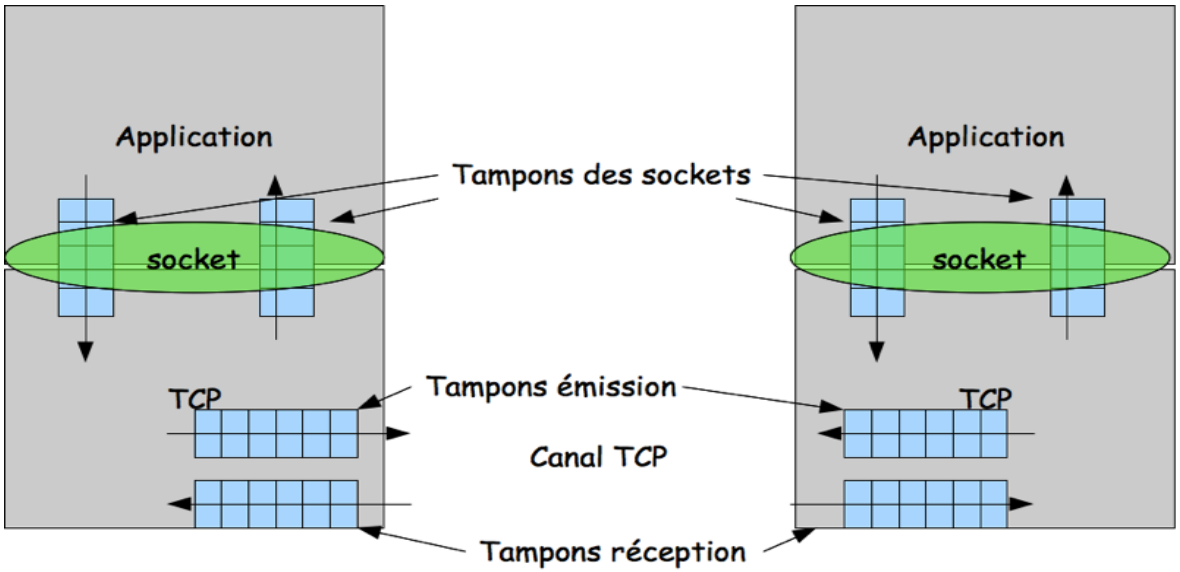
- Transfert fiable : livraison des segments dans l'ordre et gestion des segments perdus
  - utilisation de numéros de séquence:
    - pour réordonner le flux original
    - pour éliminer les segments dupliqués
  - utilisation des acquittements (ACKs) émis par le destinataire
    - retransmission des segments perdus
- Contrôle de flux
  - TCP fournit au destinataire un moyen de contrôler le débit des données envoyées par la source pour que l'émetteur n'émette pas trop vite par rapport au récepteur
- Contrôle de congestion
  - Pertes interprétées comme signal de congestion dans le réseaux.
  - L'émetteur réagit en diminuant le débit de transmission

Format de segment TCP



Les différents buffers dans une connexion TCP

- Une socket contient un buffer pour l'émission Appl→TCP et un autre pour la réception TCP→Appl
- Deux autres buffers entre la couche TCP et la couche réseau:
  - Buffer émission : segments gardés en mémoire tant que l'émetteur n'a pas reçu d'ACK
  - Buffer réception: si récepteur reçoit les segments 1,3 puis 2 : il donne 1 à l'application puis garde 3 en mémoire tant qu'il n'a pas reçu 2 car les données doivent être ordonnées

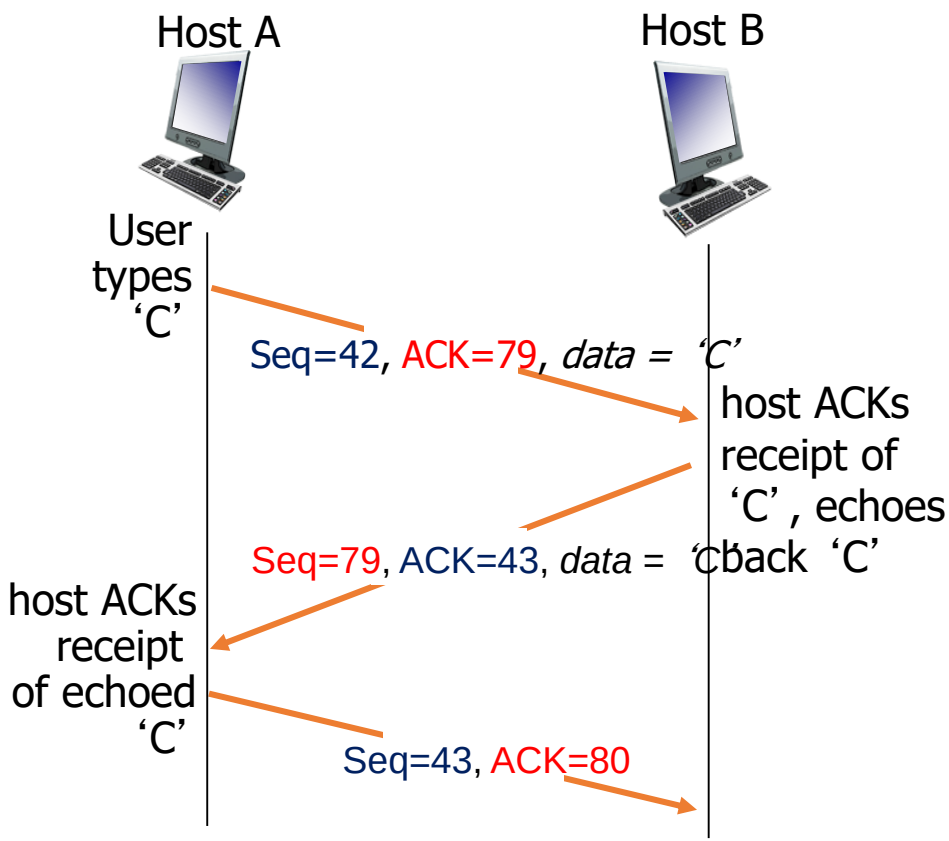


Buffer ou tampon : files d'attente entre les couches ou à l'intérieur des couches

Transfert fiable des données

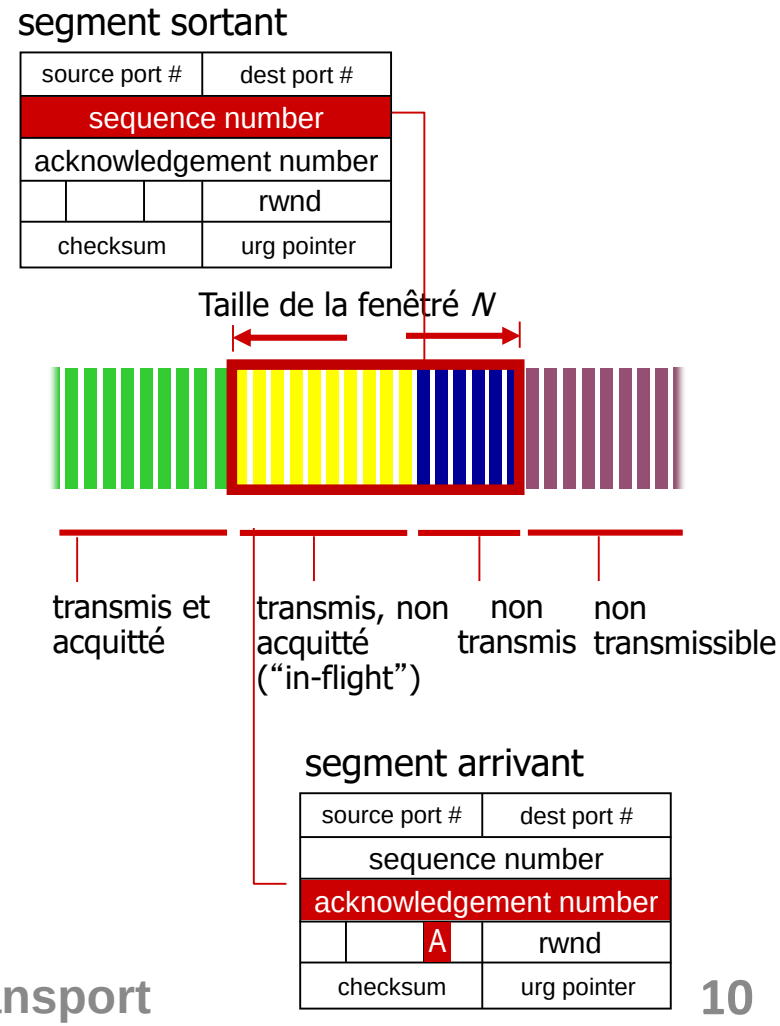
❖ Utilisation de nº de séquence et des ACKs :

- TCP transporte des octets et donc tous les nº de séquence/ACKs sont en octets!
- Pour chaque sens de la connexion :
  - Le numéro de séquence correspond au premier octet porté par le segment
  - Le numéro d'ACK au numéro du prochain octet attendu dans l'autre sens.
- Le numéro d'ACK est cumulatif.



Transfert fiable des données

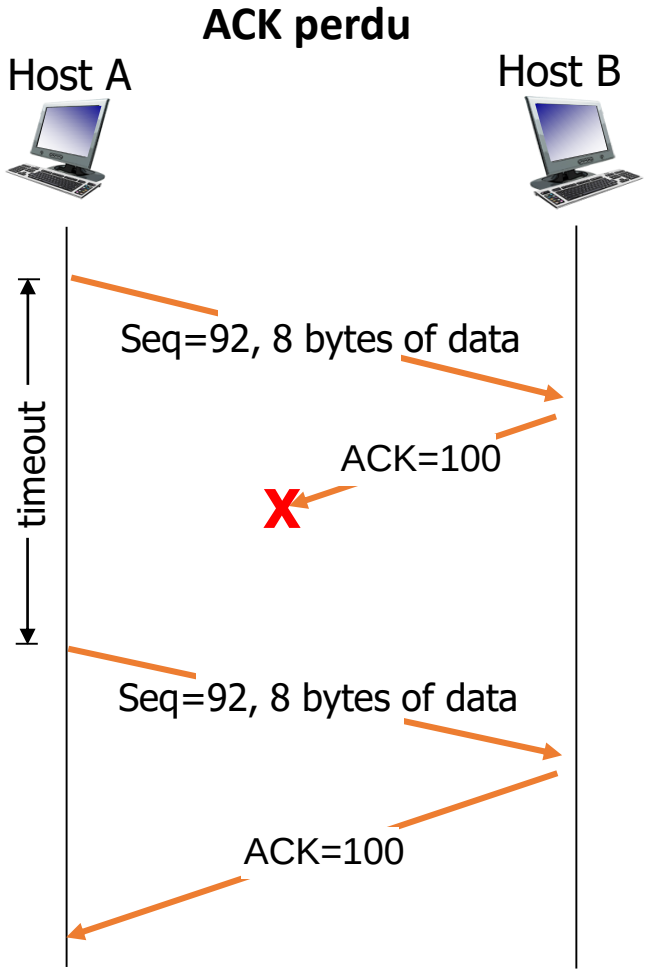
- **Fenêtre coulissante** (*sliding window*) dans le buffer d'émission :
  - On peut transmettre plus d'un segment sans acquittement (ACK)
  - Combien de segments ?
    - Autant que la taille de la fenêtre
  - La fenêtre glisse avec les ACKs



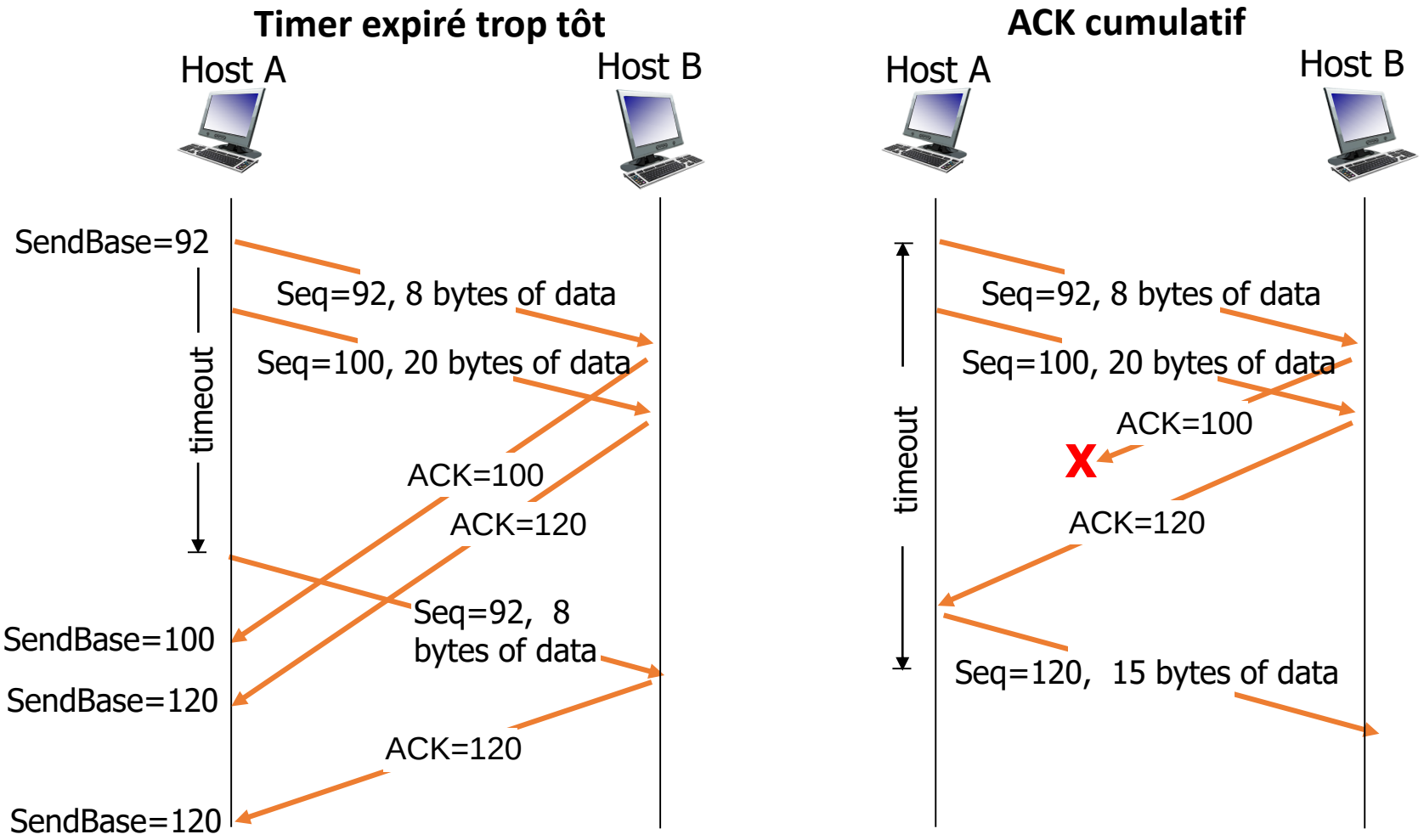
Transfert fiable des données

- Comment l'émetteur sait qu'il y a eu une perte?
  - TCP observe les numéros de séquence
  - Il renvoie, s'il y a des trous :
    - Si le *timer* de retransmission a expiré
    - Ou s'il y a des ACKs dupliqués

- **RTO: Retransmission Timeout :**
  - Le *timer* de retransmission (RTO) démarré à chaque envoi de paquet en attente d'ACK et sa valeur est en fonction de RTT (eg. RTO=2*RTT):
- **RTT: Round Trip Time :**
  - Temps entre envoi paquet et réception ACK



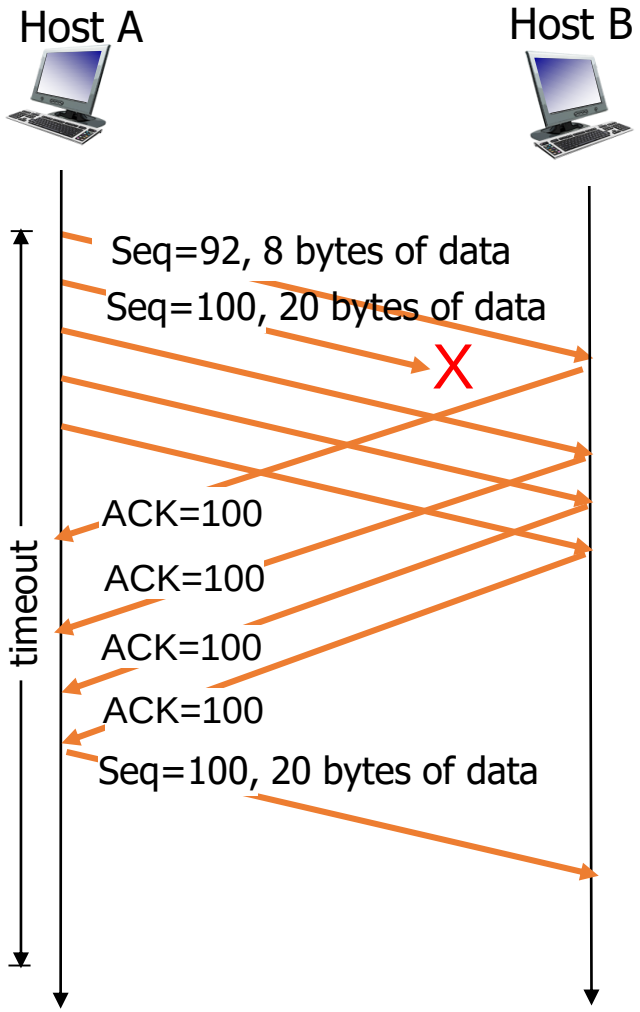
Transfert fiable des données



Transfert fiable des données

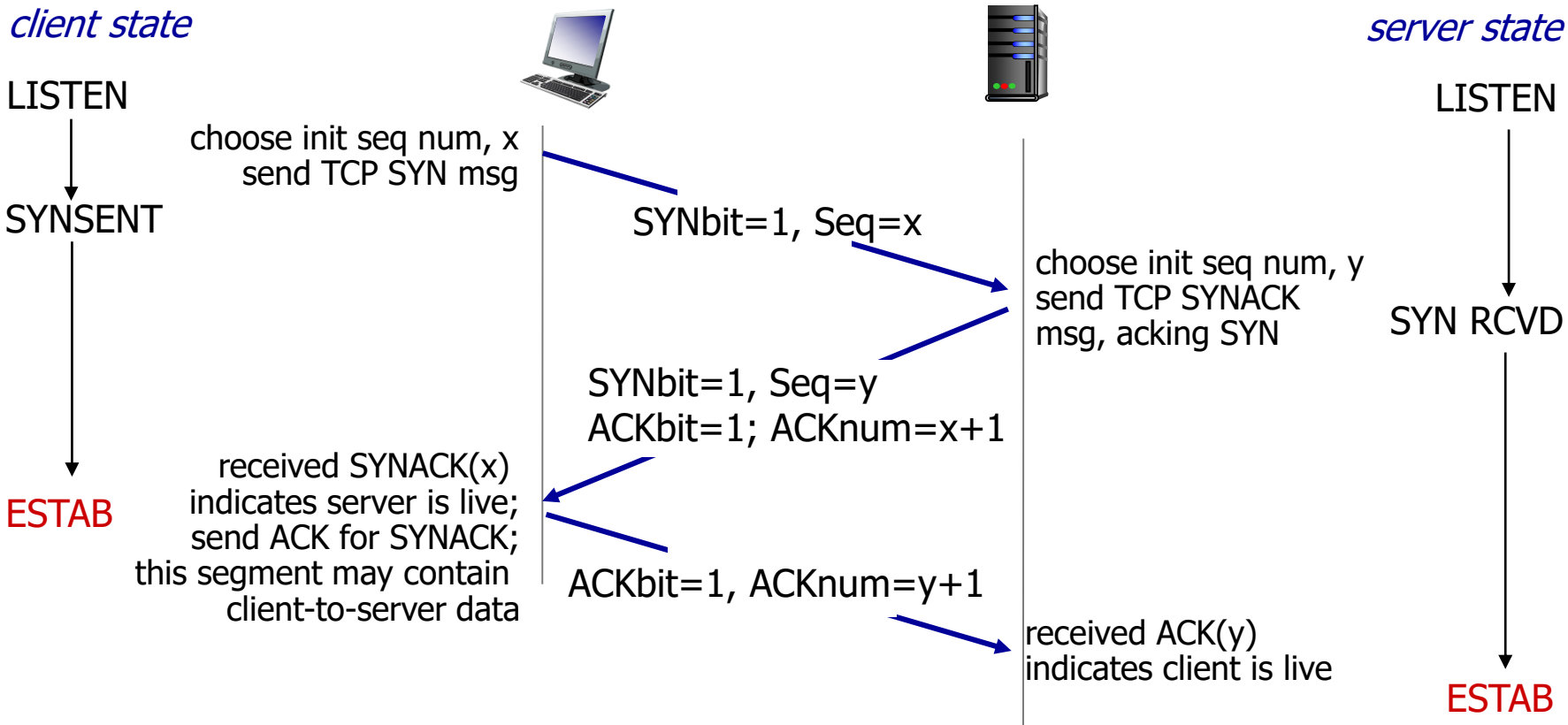
TCP fast retransmit :

- RTO souvent relativement long :
  - Long délai avant de renvoyer seg. perdu
- Détecter des segments perdus via ACK dupliqués
  - Si le segment est perdu, il y aura probablement beaucoup de ACK dupliqués
- Si l'expéditeur reçoit 3 ACKs pour les mêmes données
  - Renvoyer le segment non acquitté avec le plus petit nº de seq
  - Probable que ce segment est perdu, il ne faut pas attendre RTO



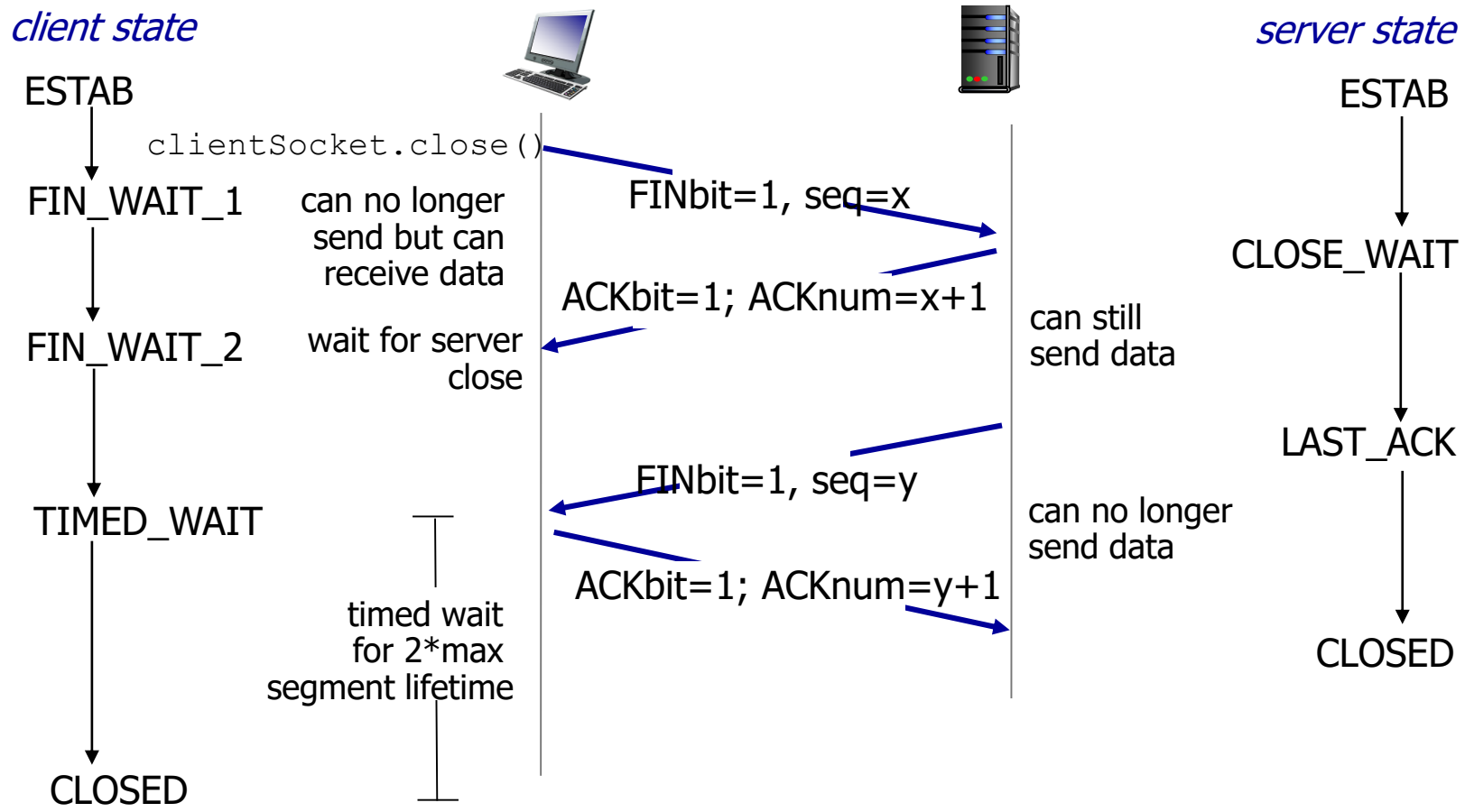
Gestion des connexions

Établissement de la connexion

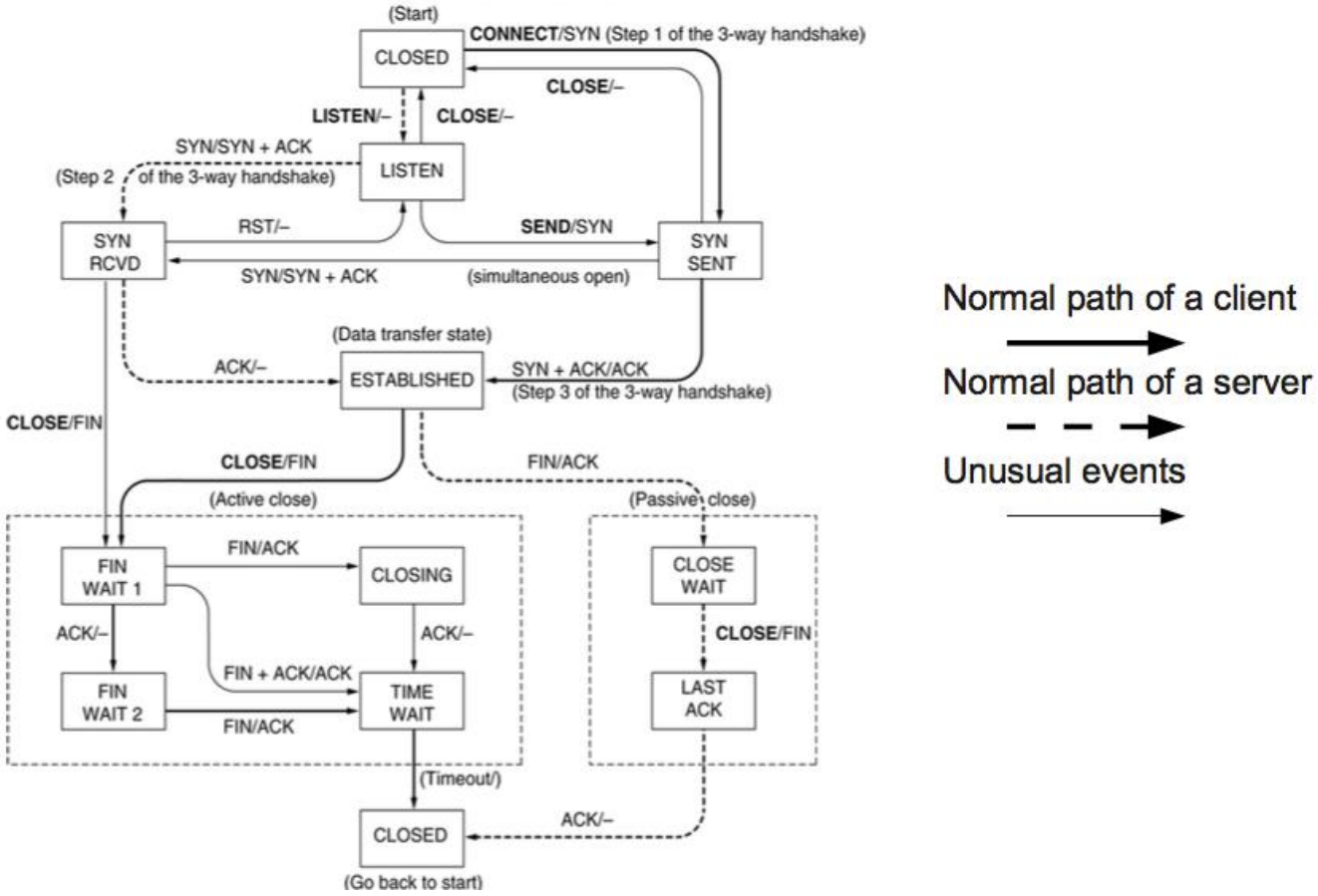


Gestion des connexions

Déconnexion



Machine à états finis des connexions

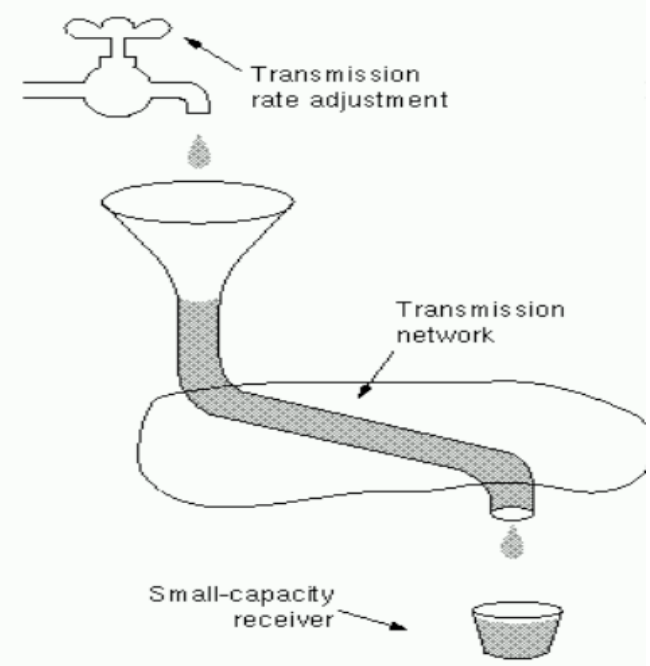


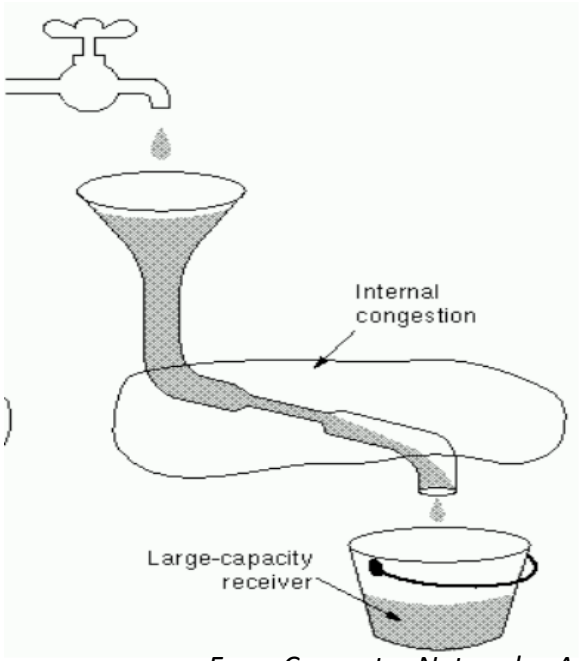
Machine à états finis

- 11 états :

State	Description
CLOSED	No connection is active or pending
LISTEN	The server is waiting for an incoming call
SYN RCVD	A connection request has arrived; wait for ACK
SYN SENT	The application has started to open a connection
ESTABLISHED	The normal data transfer state
FIN WAIT 1	The application has said it is finished
FIN WAIT 2	The other side has agreed to release
TIMED WAIT	Wait for all packets to die off
CLOSING	Both sides have tried to close simultaneously
CLOSE WAIT	The other side has initiated a release
LAST ACK	Wait for all packets to die off

Contrôle de flux vs contrôle de congestion



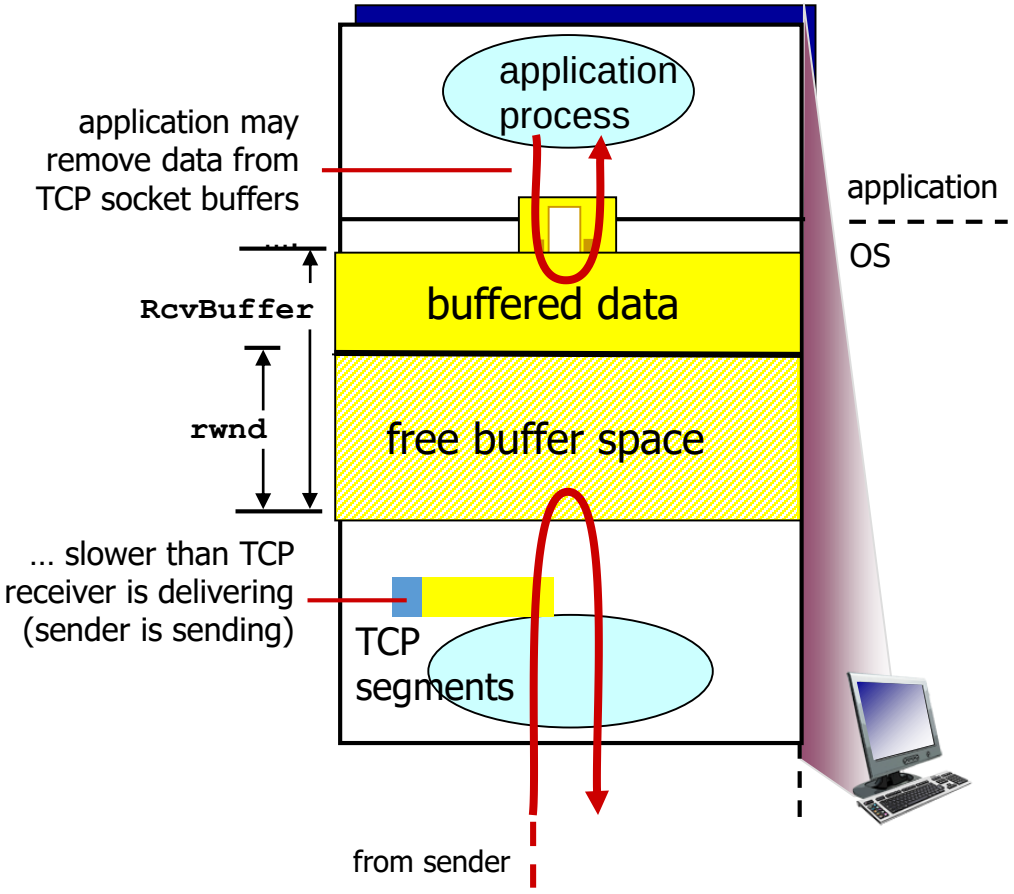


From Computer Networks, A. Tanenbaum

- Un ***réseau rapide*** alimentant un ***récepteur de faible capacité***
  - Problème : pertes du côté récepteur
  - Solution : mécanisme de ***contrôle de flux*** par le biais de `rwnd`
- Un ***réseau lent*** alimentant un ***récepteur de grande capacité***
  - Problème : pertes dans le réseau
  - Solution : mécanisme de ***contrôle de congestion*** par le biais de `cwnd`

Contrôle de flux

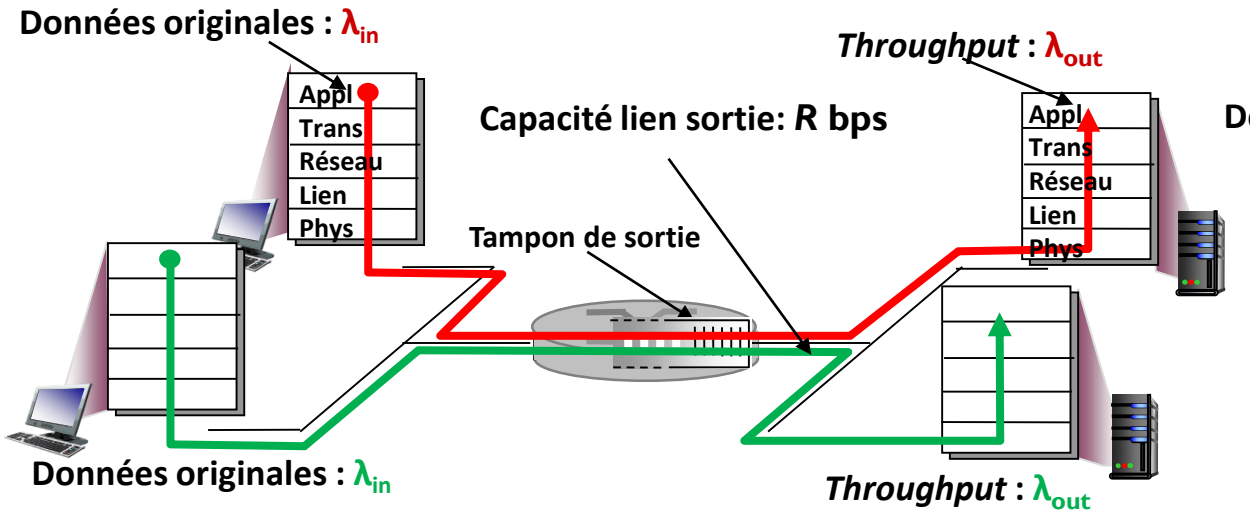
- Pour que l'émetteur n'émette pas trop vite par rapport au récepteur
- Pour éviter ce problème, on surveille le buffer TCP → Application, coté récepteur (au niveau socket)
- TCP envoie le nombre d'octets restants dans le buffer dans un champ fenêtre de réception (`rwnd`) de l'en-tête TCP
- L'émetteur limite la fenêtre d'émission, c.-à-d., la quantité de segments non acquittés (« in flight ») plus les segments transmissibles à la valeur de `rwnd` de récepteur

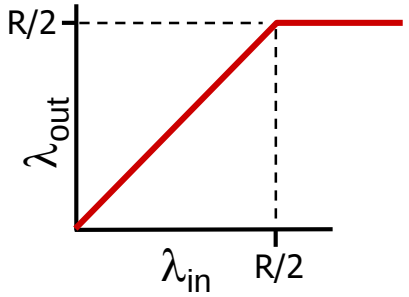


receiver-side buffering

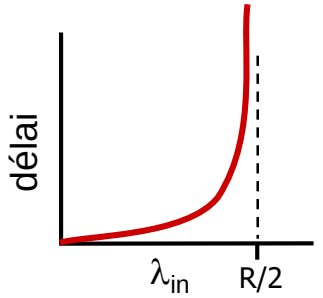
Contrôle de congestion

- Trop de sources qui envoient trop de trafic par rapport à la capacité du réseau
- *C'est diffèrent du contrôle de flux !*
- Manifestations (se rappeler de M/M/1 !!!) :
  - Paquets perdus (buffer *overflow* au niveau des routeurs)
  - Longs retards (files d'attente dans les buffers de routeurs)





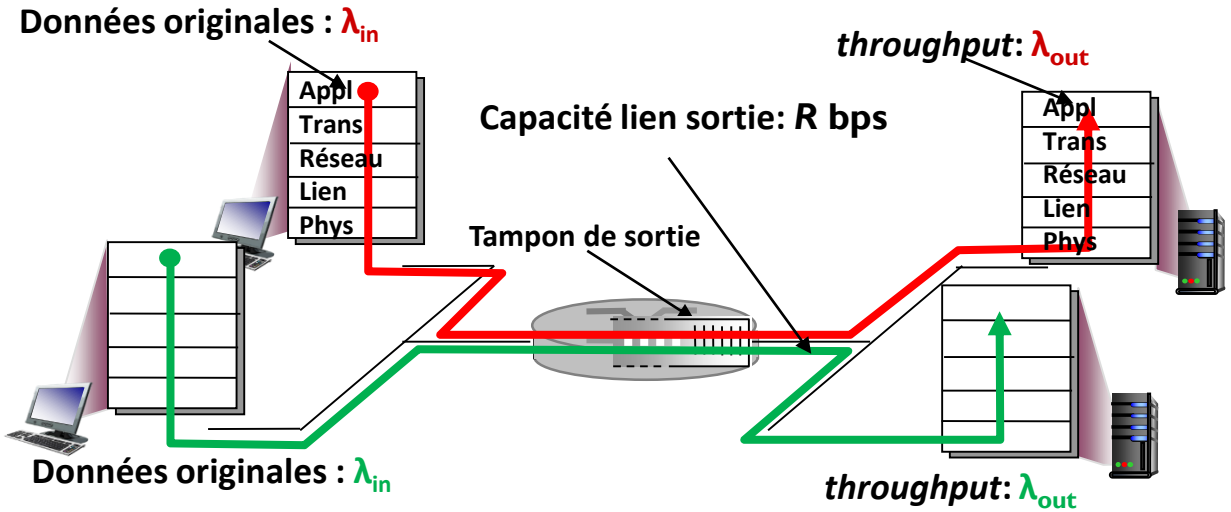
Débit maximal par connexion: R/2

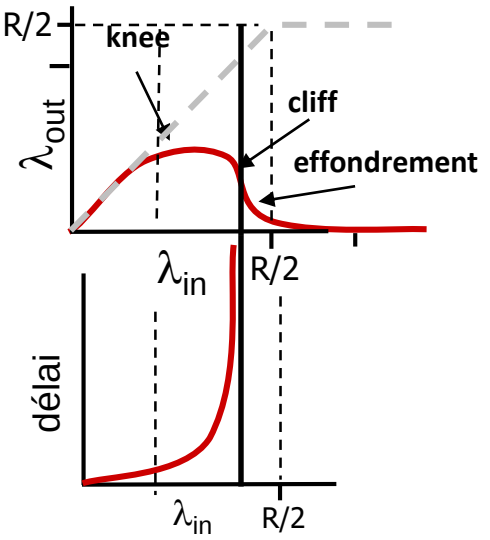


Délais augmentent lorsque l'on s'approche à R/2

Contrôle de congestion

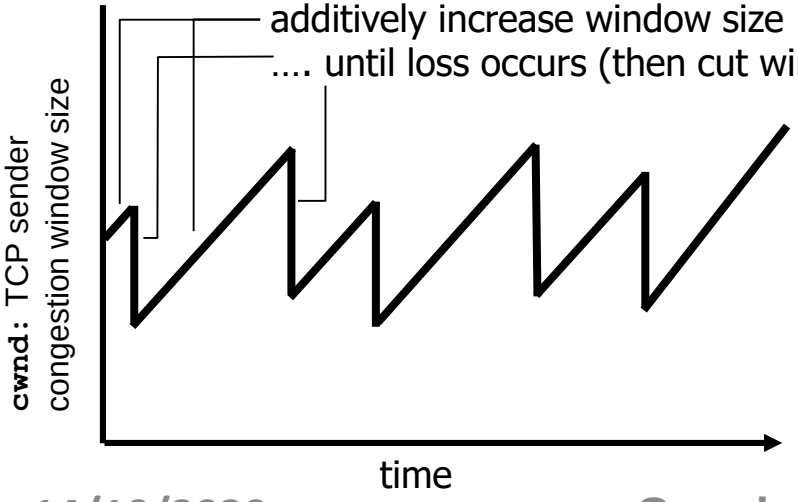
- Conséquences :
  - Plus de pertes -> plus de retransmissions, et plus de trafic -> plus de pertes
  - Retransmissions inutiles et contreproductives :
    1. D'abord, un lien porte plusieurs copies du même paquet → ça ne fait rien gagner !
    2. En plus, on ajoute de la charge au réseau !!
  - Effondrement (*« congestion collapse »*):
    - « *goodput* » λ_{out} (*throughput* effectif *dans la couche applicative*) décroit !!!
    - Il faut éviter de s'approcher du *cliff* !!

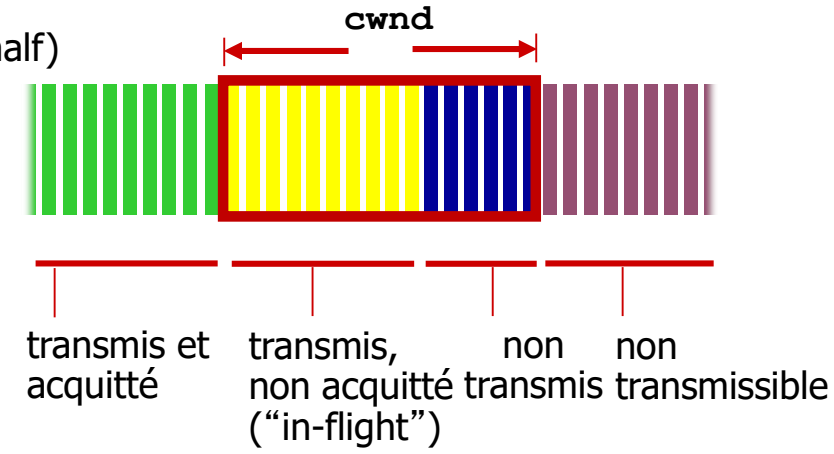




Contrôle de congestion

- Se fait par le contrôle de la fenêtre d'émission (ou de congestion) de taille `cwnd`
- TCP envoie `cwnd` octets et attend RTT pour les ACKS avant de continuer: débit de TCP = `cwnd` / RTT [octets/s]
- `cwnd` ajustée en fonction des pertes:
  - si perte, *diminution multiplicative* : on réduit `cwnd` de moitié après la perte
  - sinon, *augmentation additive* : on augmente `cwnd` par 1 MSS chaque RTT jusqu'à ce que la perte soit détectée



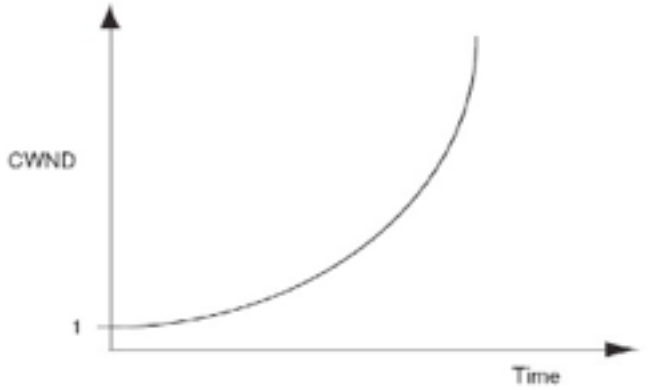


Contrôle de congestion

Slow Start (`cwnd` < `ssthresh`)

- Après une congestion ne pas remonter trop vite :
  - initialiser la fenêtre à 1 segment
  - augmentation de 1 à chaque ACK
  - `cwnd` double à chaque RTT (exponentiel)
- Atteinte le seuil `ssthresh` → *congestion avoidance*
- Perte (timeout) :
  - TAHOE : `ssthresh` = `cwnd`/2; `cwnd` =1 → Slow Start
  - RENO : `ssthresh` = `cwnd`/2; `cwnd` =`cwnd` /2 → Slow Start

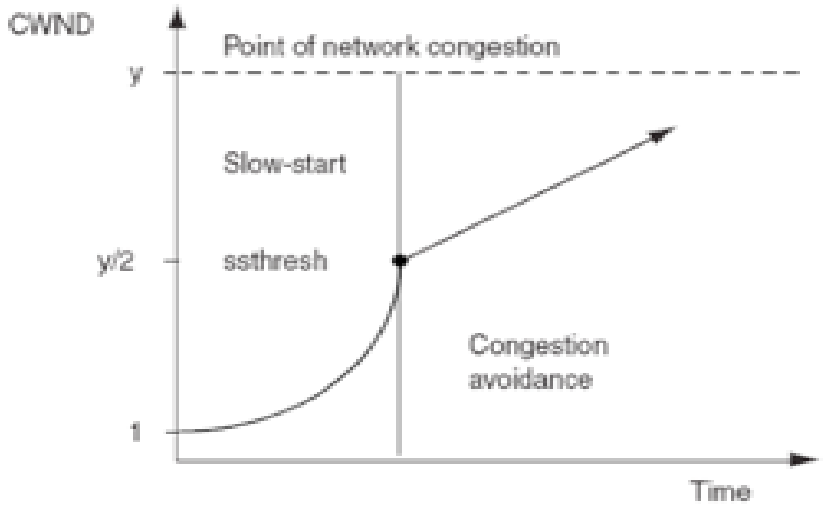
`ssthresh` :
Estimation bande passante disponible



Contrôle de congestion

Congestion Avoidance (`cwnd ≥ ssthresh`)

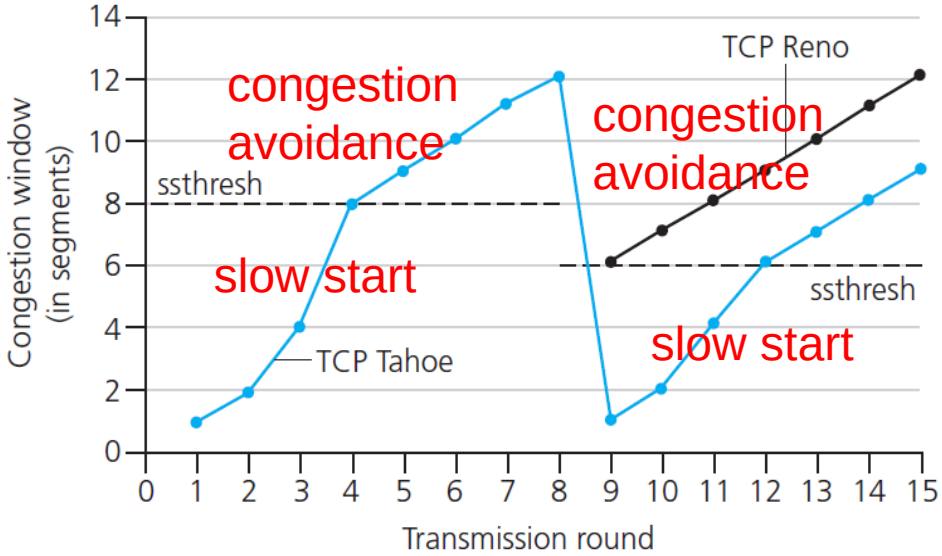
- Pour stopper une augmentation trop rapide
- À partir du seuil `ssthresh` (+1 segment par RTT)
- Perte (timeout) :
  - TAHOE : `ssthresh` = `cwnd`/2; `cwnd` =1 → Slow Start
  - RENO : `ssthresh` = `cwnd`/2; `cwnd` =`cwnd` /2 → Slow Start



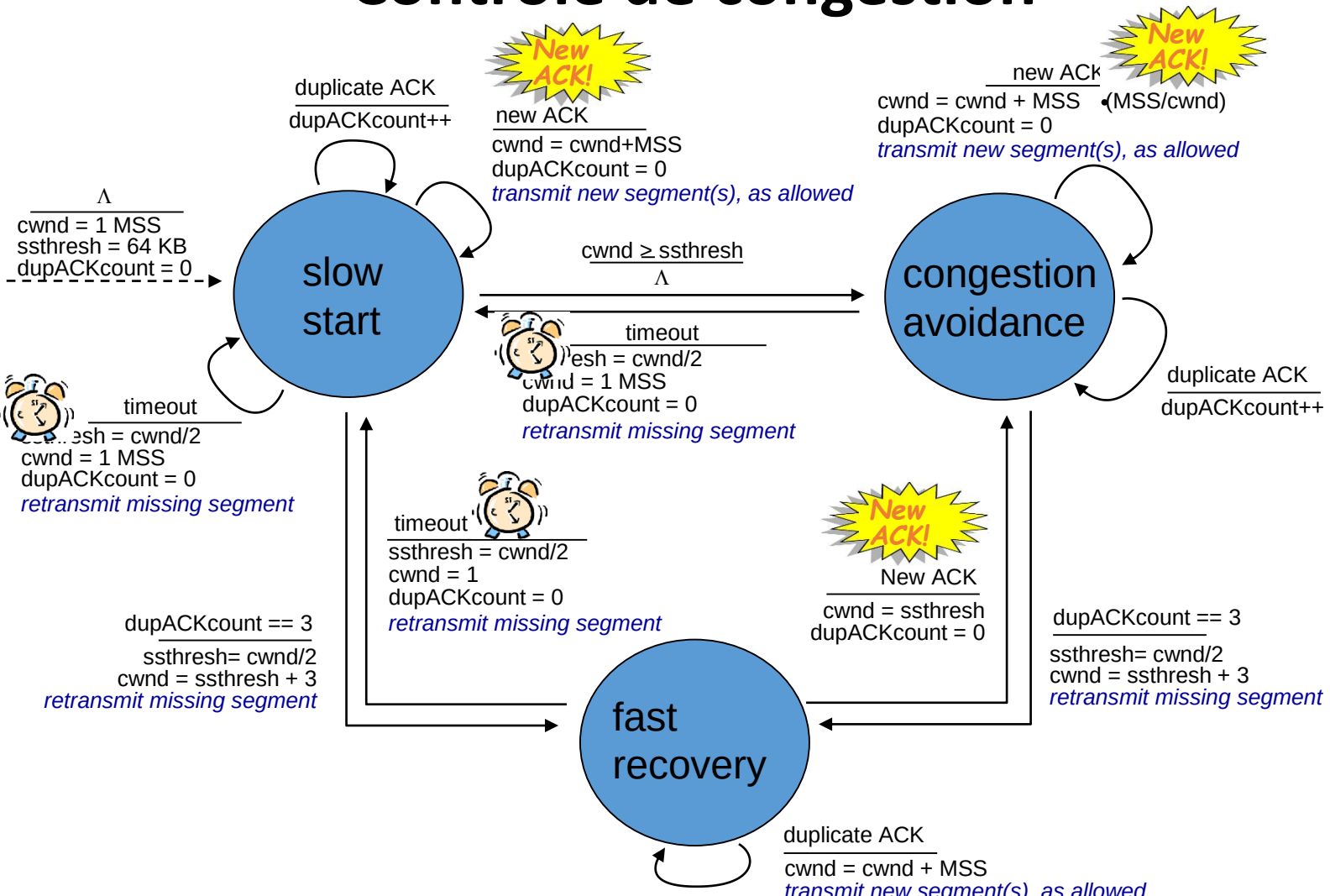
Contrôle de congestion

Fast Recovery

- Si plusieurs (3) `dupACK`, 1 seule perte
- `ssthresh` =`cwnd`/2, `cwnd`=`ssthresh` +3 (car 3 paquets acquittés)
- Pour chaque `dupACK`, `cwnd`++
- Réception non `dupACK` : dégonflement `cwnd`
  - `cwnd` = `ssthresh`
  - → Congestion Avoidance



Contrôle de congestion



Vue d'ensemble d'UDP [RFC 768]

UDP : User Datagram Protocol

- Protocole de transport en mode déconnecté (non orienté connexion) :
  - Pas de « 3-way handshaking » entre l'émetteur et le récepteur
  - Création de segments UDP et transmission immédiate
  - Chaque segment UDP est géré indépendamment des autres
- Démultiplexage non orienté connexion :
  - Côte récepteur : des segments avec le même nº port de destination mais les cordonnées de source (adresses IP, nº port) différentes seront dirigés vers une même socket
- Transfert non-fiable : service «best effort»
  - Les segments UDP peuvent être perdus et/ou livrés dans le désordre à la couche application
  - Si on veut de la fiabilité, c'est à la couche application de l'implémenter : mécanismes spécifiques à chaque protocole application

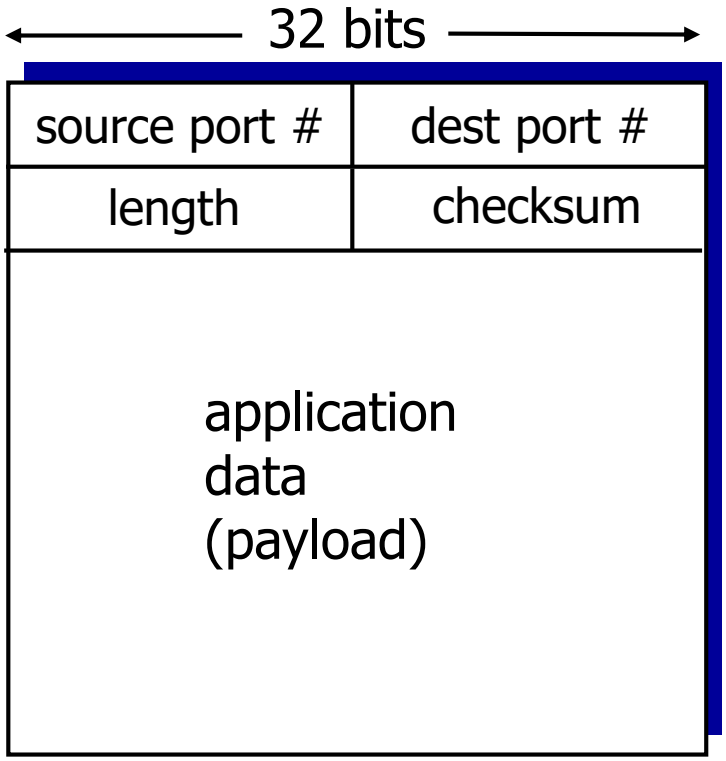
Vue d'ensemble d'UDP [RFC 768]

- Alors, pourquoi UDP existe-t-il ?
  - pas d'établissement de connexion (qui peut ajouter un délai)
  - il est simple : pas d'état de connexion côté émetteur ou récepteur
  - en-tête de petite taille
  - aucun contrôle de congestion : UDP peut transmettre aussi vite que possible

- Utilisation UDP :
  - applications tolérantes à la perte de données mais sensibles aux retards
  - p.ex. : applications de streaming multimédia, DNS

Format de segment UDP

- Segment avec en-tête de 8 octets :
  - Champs dans l'en-tête avec les ports comme TCP
  - Le champ `length` est la longueur en octets du segment UDP, en-tête comprise
  - Champ `checksum` (optionnel) utilisé pour la détection d'erreurs
  - `payload` qui encapsule les données d'application



checksum

- Objectif : détecter les « erreurs » (par exemple des bits renversés) dans le segment transmis
- Côte émetteur
  - Le contenu du segment, y compris l'en-tête, est exprimé comme une séquence de nombres entiers de 16 bits
  - *Checksum* : le complément à 1 de la somme de cette séquence d'entiers de 16 bits
  - L'émetteur met la valeur dans le champ checksum de l'en-tête du segment
- Côte récepteur
  - Il recalcule la *checksum* avec le contenu du segment reçu
  - Il vérifie si la *checksum* calculée est égale à la valeur du champ `checksum` :
    - NO - erreur détectée
    - OUI - pas d'erreur détectée

checksum

- Exemple

3 mots de 16 bits

- 0110011001100000
  0101010101010101
  1000111100001100

un “vrai” + avec retenue

```
 0110011001100000
+0101010101010101
 1011101110110101

 1011101110110101
+1000111100001100
 ----------------
 0100101011000010
+1
 ----------------
 0100101011000011
```

complément à 1=checksum 1011010100111101

destinataire fait la somme des 4 mots=1111111111111111