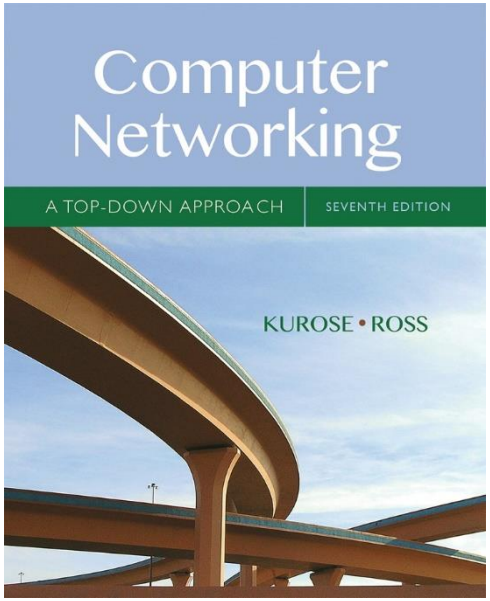
Chapitre III
La couche transport (suite)



Computer Networking: A Top Down Approach
7ème édition
Jim Kurose, Keith Ross
Addison-Wesley
2017

A note on the use of these Powerpoint slides:

We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you see the animations; and can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:

- If you use these slides (e.g., in a class) that you mention their source (after all, we'd like people to use our book!)
- If you post any slides on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

Thanks and enjoy! JFK/KWR

Chapitre 3: plan (suite)

3.1 services de la couche transport

3.2 multiplexage et démultiplexage

3.3 transport sans connexion: UDP

3.4 principes du transfert fiable des données

3.5 transport orienté connexion: TCP

- structure d'un segment
- transfert fiable
- contrôle de flux
- gestion de connexion

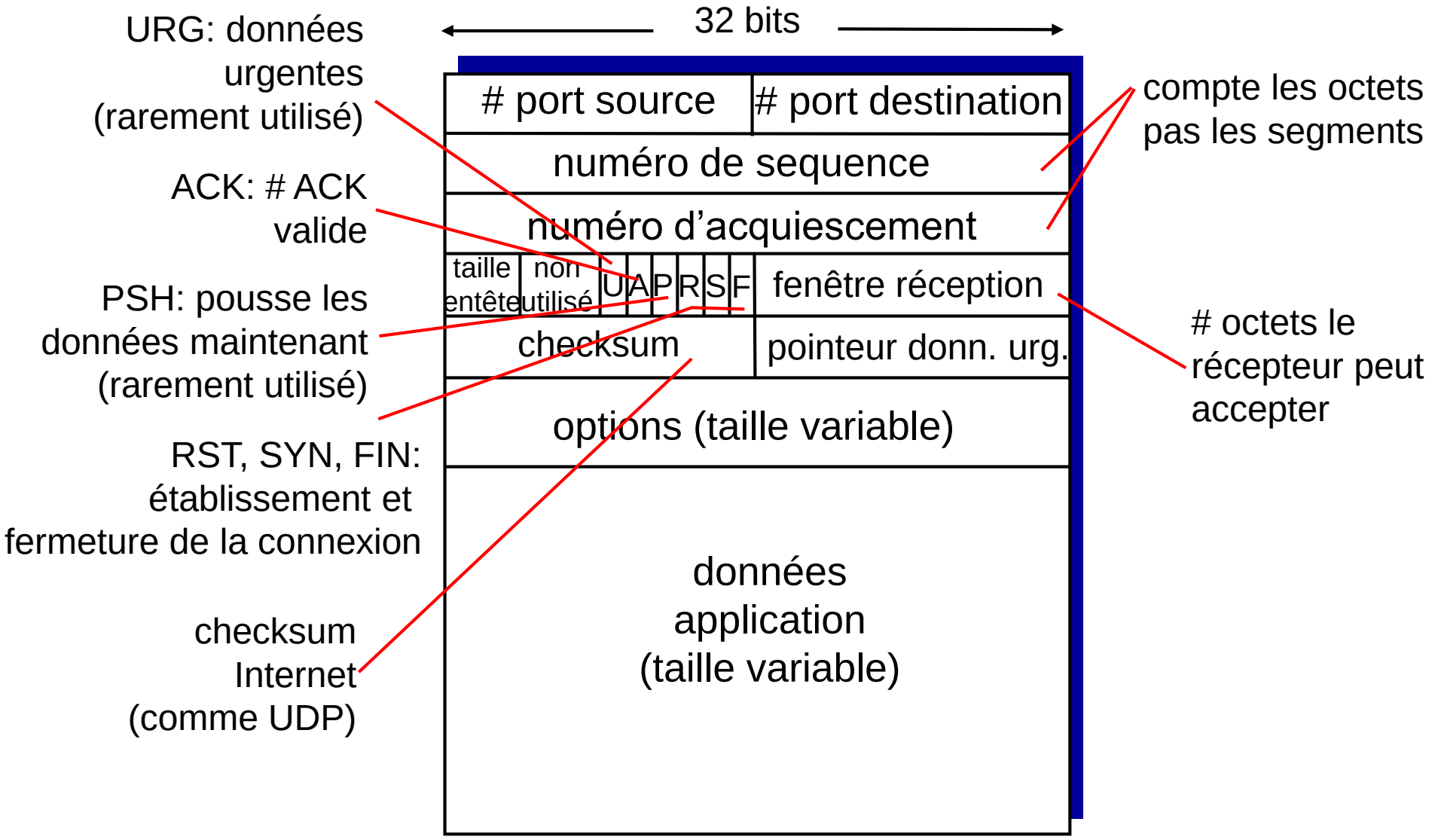
3.6 principes du contrôle de congestion

3.7 contrôle de congestion dans TCP

TCP: RFCs: 793,1122,1323, 2018, 2581

- point-à-point:
  - 1 émetteur, 1 récepteur
- fiable, flux ordonné d'octets:
  - pas de limites sur les messages
- "pipelining":
  - taille de la fenêtre dépend du contrôle de flux et de congestion
- données en duplex:
  - même connexion pour un flux bidirectionnel
  - MSS: "maximum segment size"
- orienté connexion:
  - handshaking (échange de msgs de contrôle) initialisations des variables d'état
- flux contrôlé:
  - l'émetteur ne submerge pas le récepteur

Structure du segment TCP



TCP numéros de séq., ACKs

numéros de séquence:

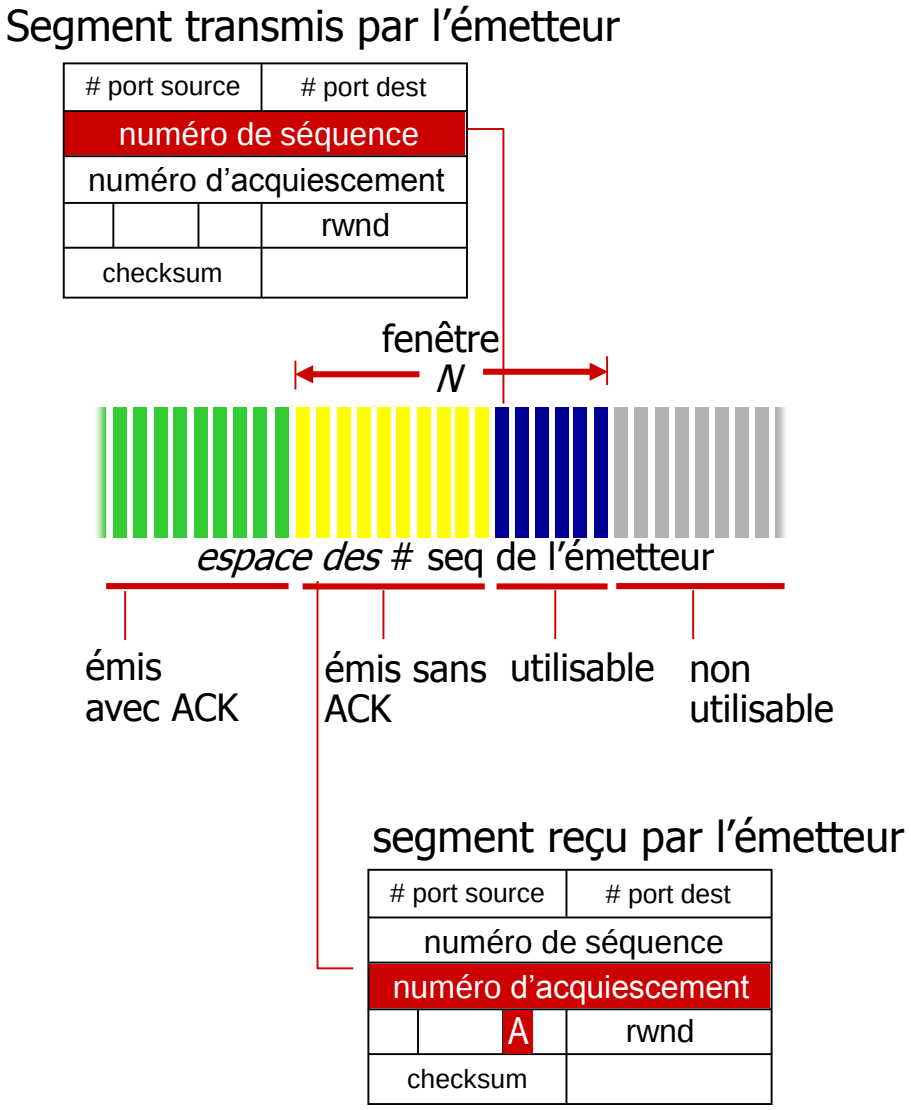
- position du premier octet dans le segment par rapport au flux

accusés de réception:

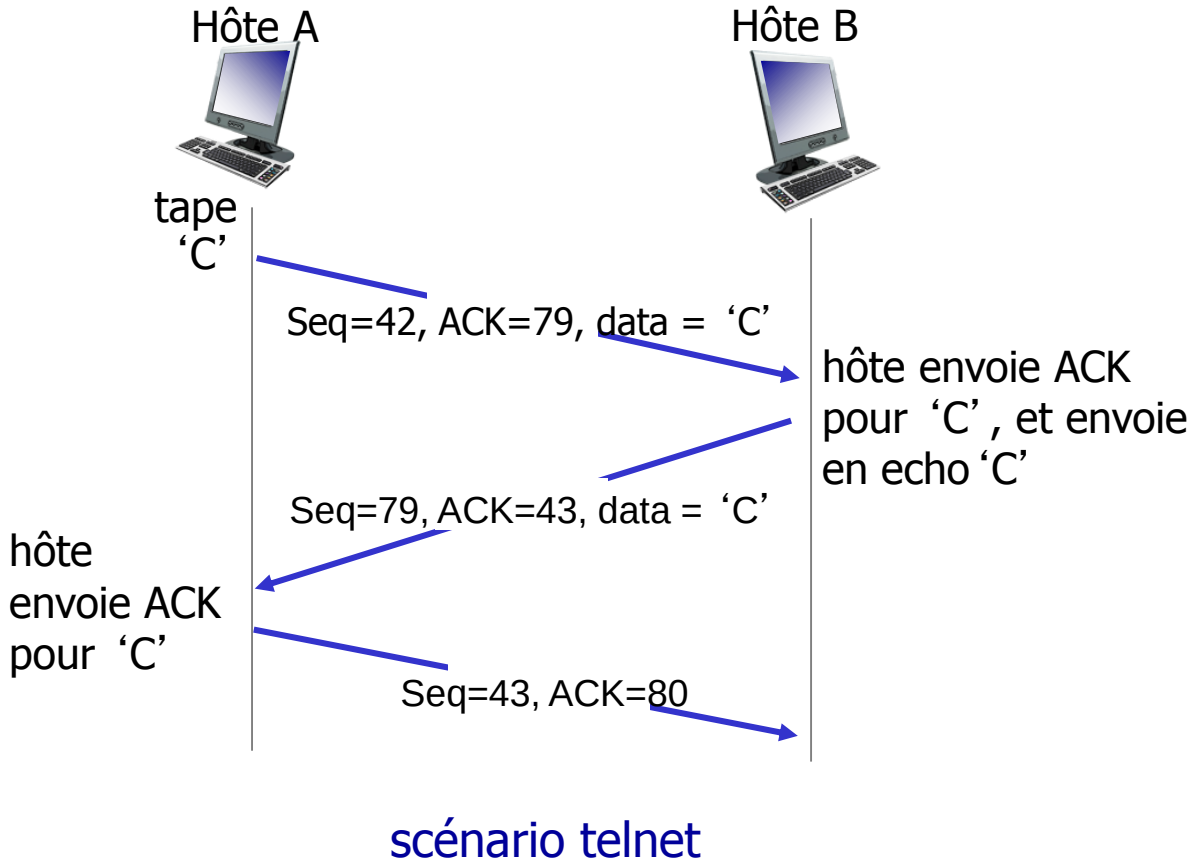
- # séq du prochain octet attendu
- ACK cumulatif

Q: comment le récepteur traite les segments non ordonnés?

- R: dépend de l'implémentation



TCP numéros de séq., ACKs



scénario telnet

TCP transfert fiable

- TCP crée un service fiable au dessus d'un service IP non fiable
  - segments en pipeline
  - acks cumulatifs
  - un seul timer pour la retransmission
- retransmissions déclenchées par:
  - timeout
  - acks dupliqués

Commençant par un émetteur simplifié:

- ignorer les acks dupliqués
- ignorer les contrôles de flux et de congestion

TCP évènements chez l'émetteur:

données reçues de l'app:

- créer un segment avec # seq
- démarrer le timer s'il ne l'est pas déjà
  - timer associé au plus ancien segment non acquiescé

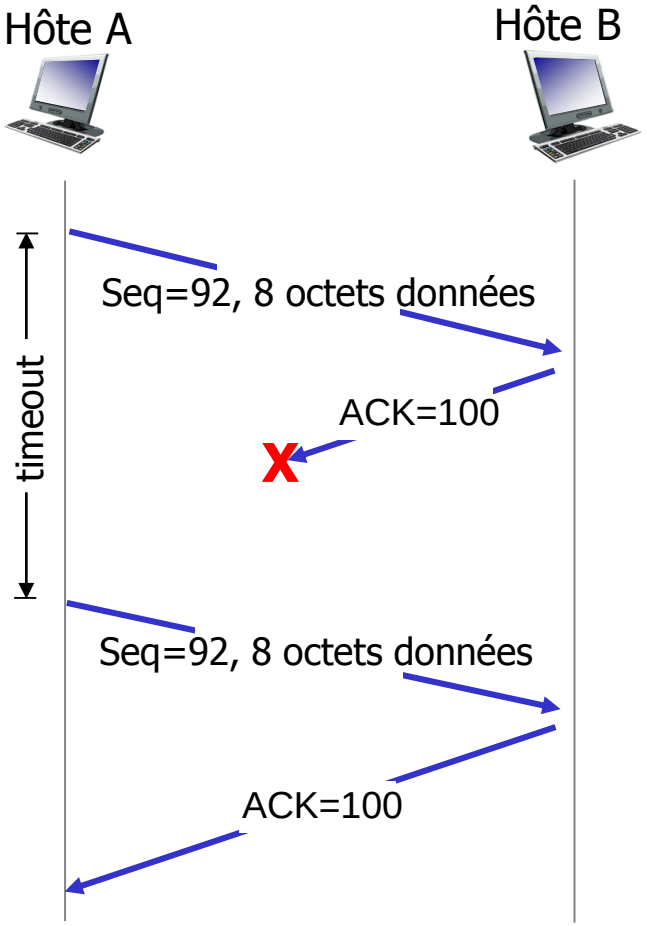
timeout:

- retransmettre le segment responsable
- redémarrer le timer

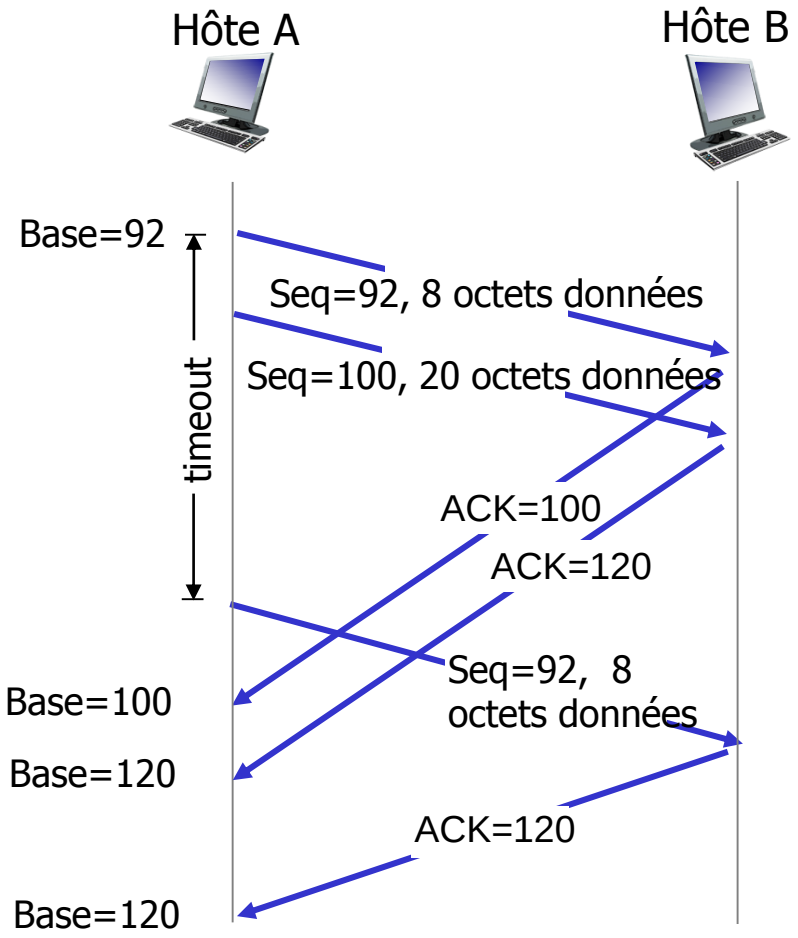
ack reçu:

- si ack accuse la réception d'un paquet non acquiescé
  - mettre à jour les segments acquiescés
  - démarrer le timer s'il y a des segment non acquiescés

TCP: scénarios de retransmission

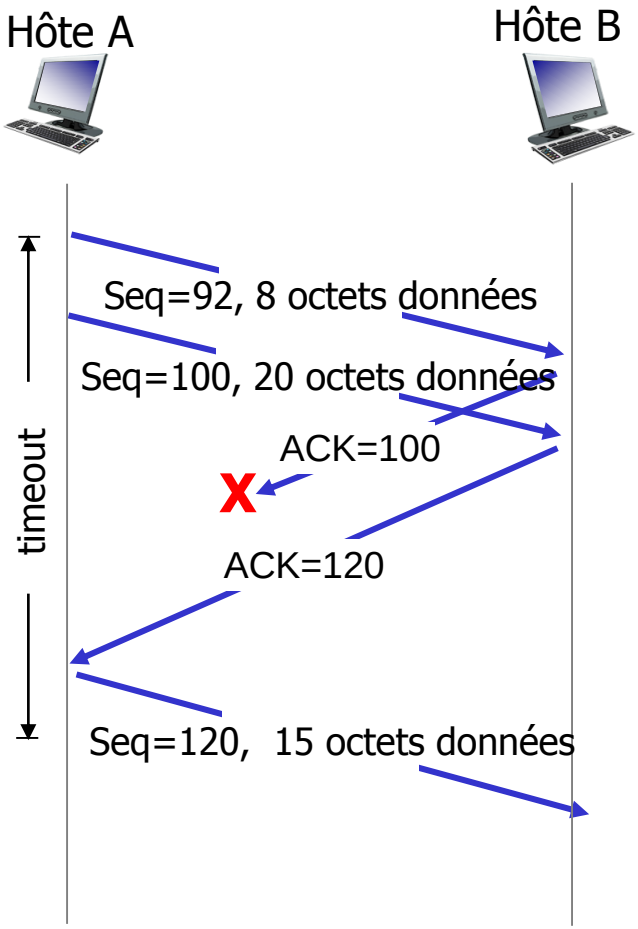


Ack perdu



timeout prématuré

TCP: scénarios de retransmission



ACK cumulatif

Note: Doubler la valeur du timeout

génération des ACK TCP [RFC 1122, RFC 2581]

événement au récepteur	*Action du récepteur*
segment en ordre avec # séq attendu. données avant acquiescées	Retarder ACK. Attendre 500ms pour le prochain segment. Si rien, envoyer un ACK
segment en ordre avec # séq attendu Un autre segment attend son ACK	Immédiatement envoyer un ACK cumulatif,
segment hors ordre avec # séq supérieur à celui attendu ➔ Gap	Immédiatement envoyer un *ACK dupliqué*
segment qui remplit partiellement ou complètement un gap	Immédiatement envoyer un ACK,

TCP retransmission rapide

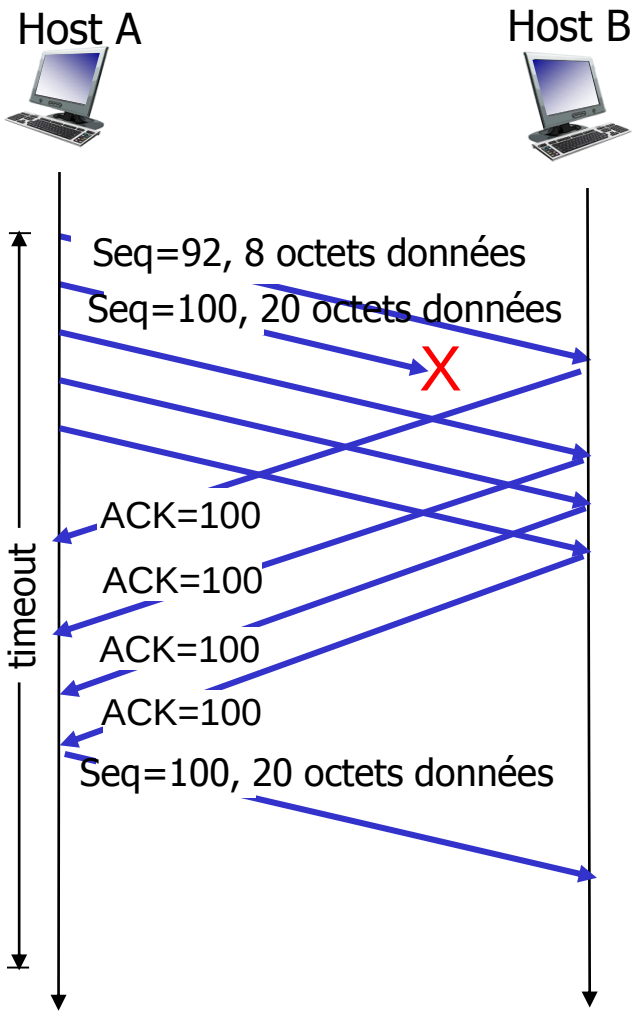
- time-out est souvent long
- détecter segments perdus via ACKs dupliqués
  - émetteur envoie successivement plusieurs segments
  - si segment est perdu, il y aura plusieurs ACKs dupliqués.

TCP fast retransmit

si émetteur reçoit 1+3 ACKs pour la même données, renvoyer le segment non acquiescé ayant le plus petit seq

- ne pas attendre le timeout (la probabilité de la perte est très grande)

TCP retransmission rapide



retransmission rapide après la réception de trois ACK dupliqués

RTT TCP et timeout

Q: comment fixer la valeur de TCP timeout?

- plus longue que le RTT
  - mais RTT varie
- S'il est trop court: (timeout prématuré)
  - retransmissions indésirables
- S'il est trop long: réaction lente à la perte du segment

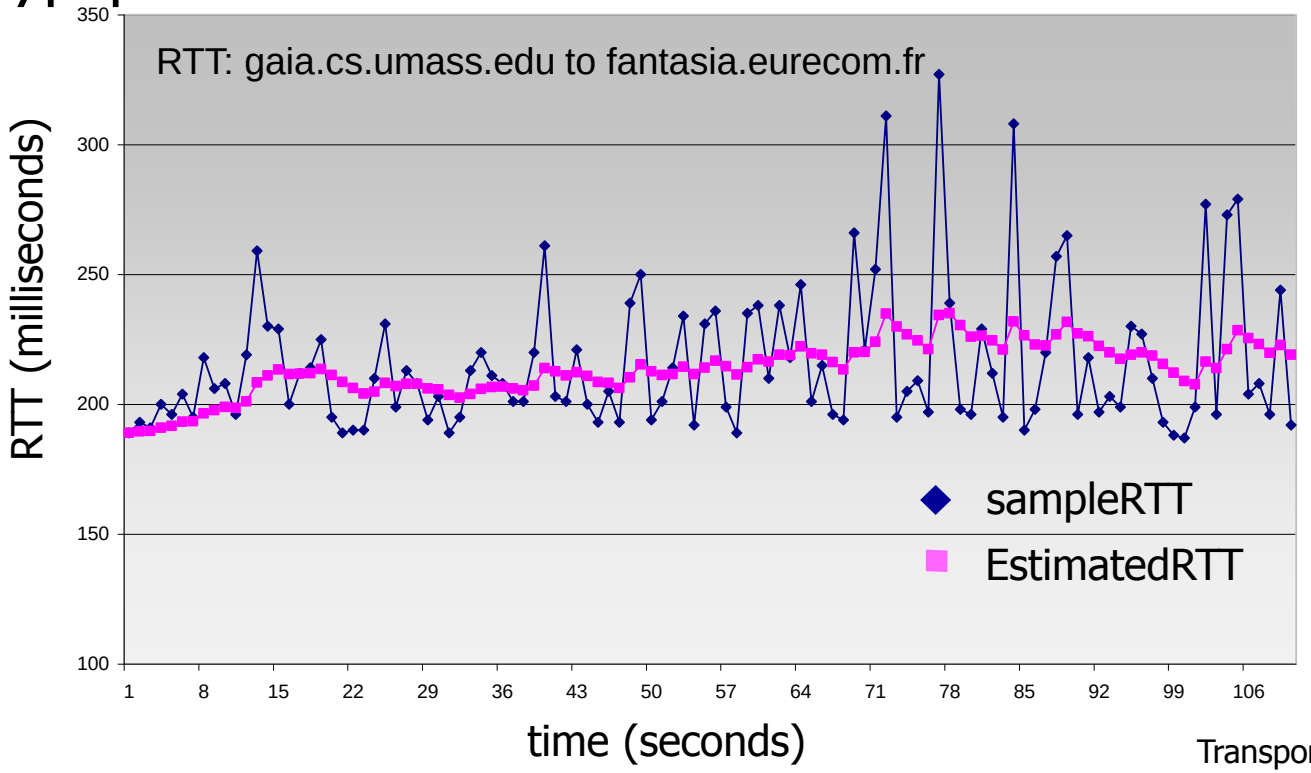
Q: comment estimer le RTT?

- **`SampleRTT`**: mesuré entre la transmission d'un segment et la réception du ACK
  - ignorer les retransmissions
- **`SampleRTT`** va varier, il faut améliorer l'estimation du RTT
  - utiliser une moyenne à la place d'une valeur instantannée

RTT TCP et timeout

EstimatedRTT = (1- α)*EstimatedRTT + α*SampleRTT

- moyenne pondérée exponentielle glissante
- l'influence des anciennes mesures diminue d'une manière exponentielle
- typiquement α = 0.125



RTT TCP et timeout

- timeout: **`EstimatedRTT`** plus "marge de sécurité"
  - grande variation dans **`EstimatedRTT`** -> grande marge
- estimer de combien SampleRTT s'éloigne de EstimatedRTT:

```
DevRTT = (1-β)*DevRTT +
                 β*|SampleRTT-EstimatedRTT|

                 (typically, β = 0.25)
```

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

RTT estimé

"marge de sécurité"

Chapitre 3: plan

3.1 services de la couche transport

3.2 multiplexage et démultiplexage

3.3 transport sans connexion: UDP

3.4 principes du transfert fiable des données

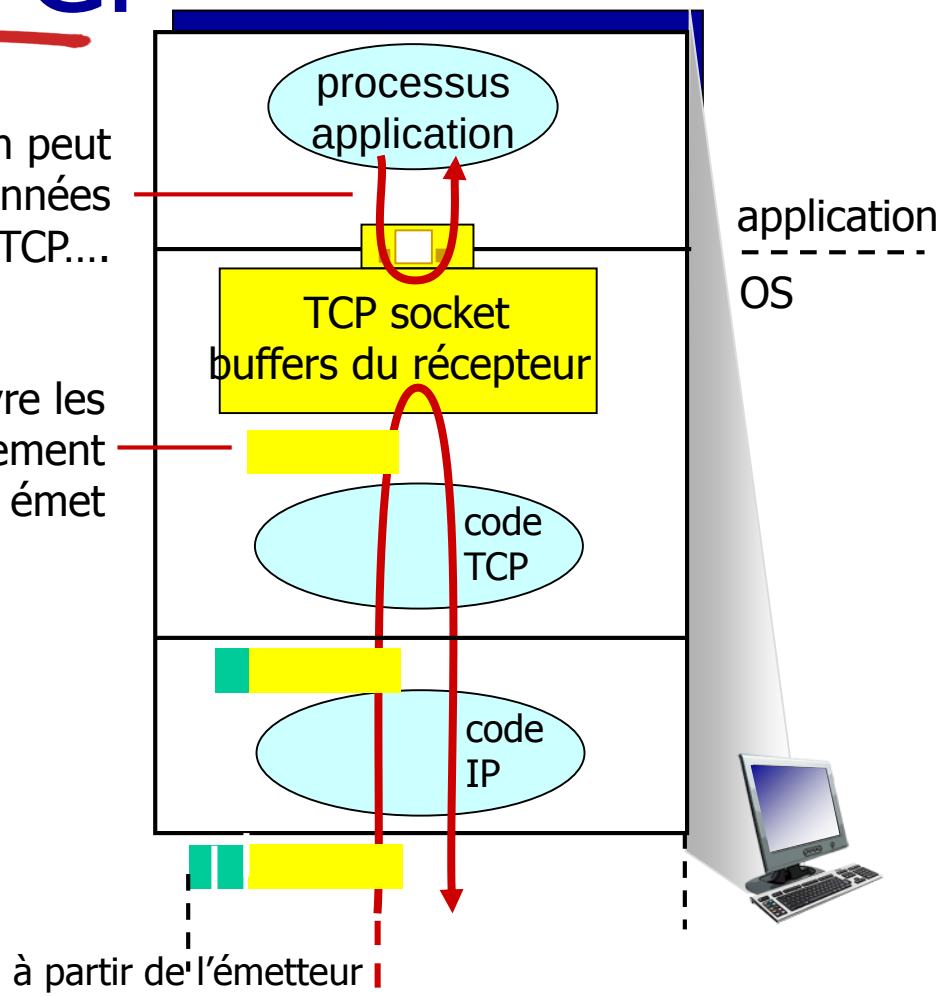
3.5 transport orienté connexion: TCP

- structure d'un segment
- transfert fiable
- contrôle de flux
- gestion de connexion

3.6 principes du contrôle de congestion

3.7 contrôle de congestion dans TCP

Contrôle de flux TCP



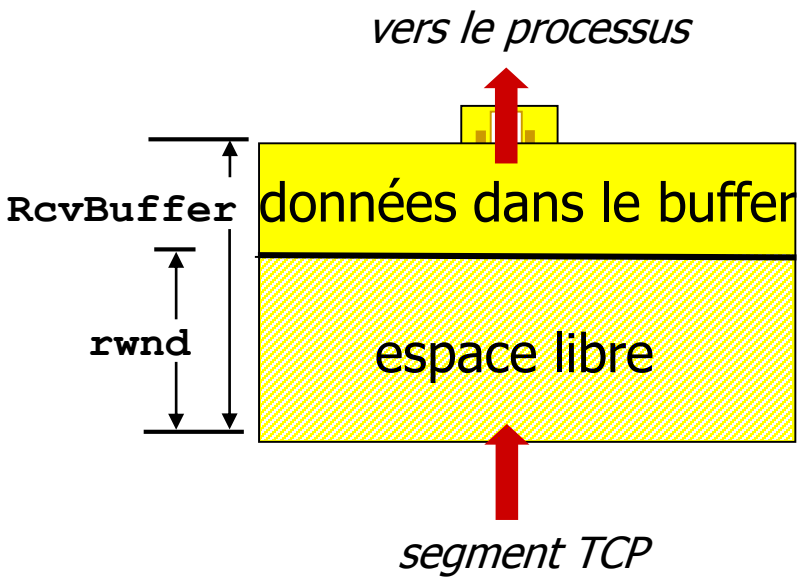
pile de protocole du récepteur

contrôle de flux

le récepteur contrôle l'émetteur, l'émetteur ne submergera pas le buffer du récepteur

Contrôle de flux TCP

- le récepteur "annonce" l'espace libre dans son buffer en incluant la valeur **`rwnd`** dans l'en-tête TCP
- l'émetteur limite la quantité de données sans ACKs au **`rwnd`** du récepteur
- garantie que le buffer du récepteur ne déborde pas



mise en mémoire tampon chez le récepteur

Chapitre 3: plan

3.1 services de la couche transport

3.2 multiplexage et démultiplexage

3.3 transport sans connexion: UDP

3.4 principes du transfert fiable des données

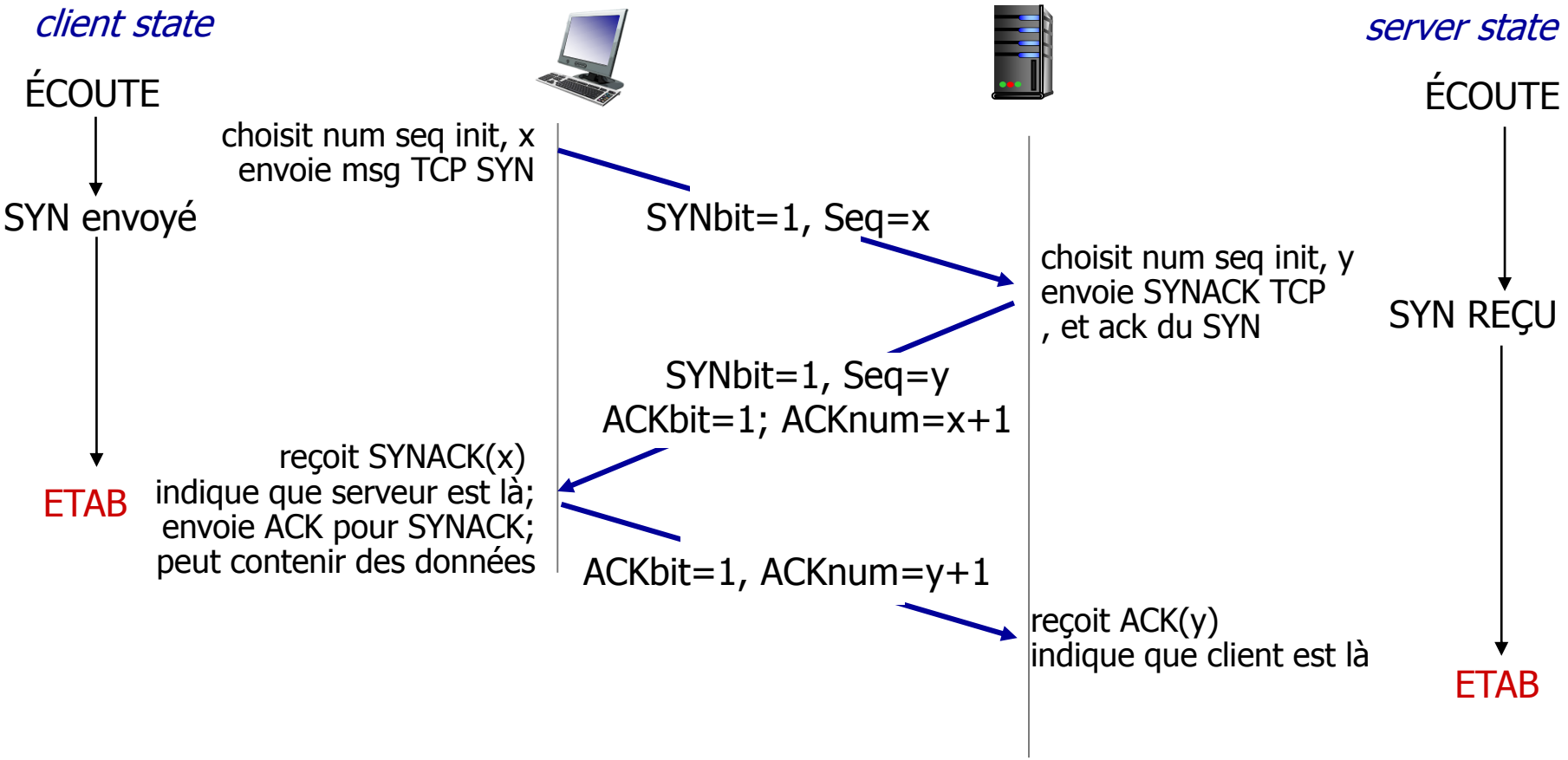
3.5 transport orienté connexion: TCP

- structure d'un segment
- transfert fiable
- contrôle de flux
- gestion de connexion

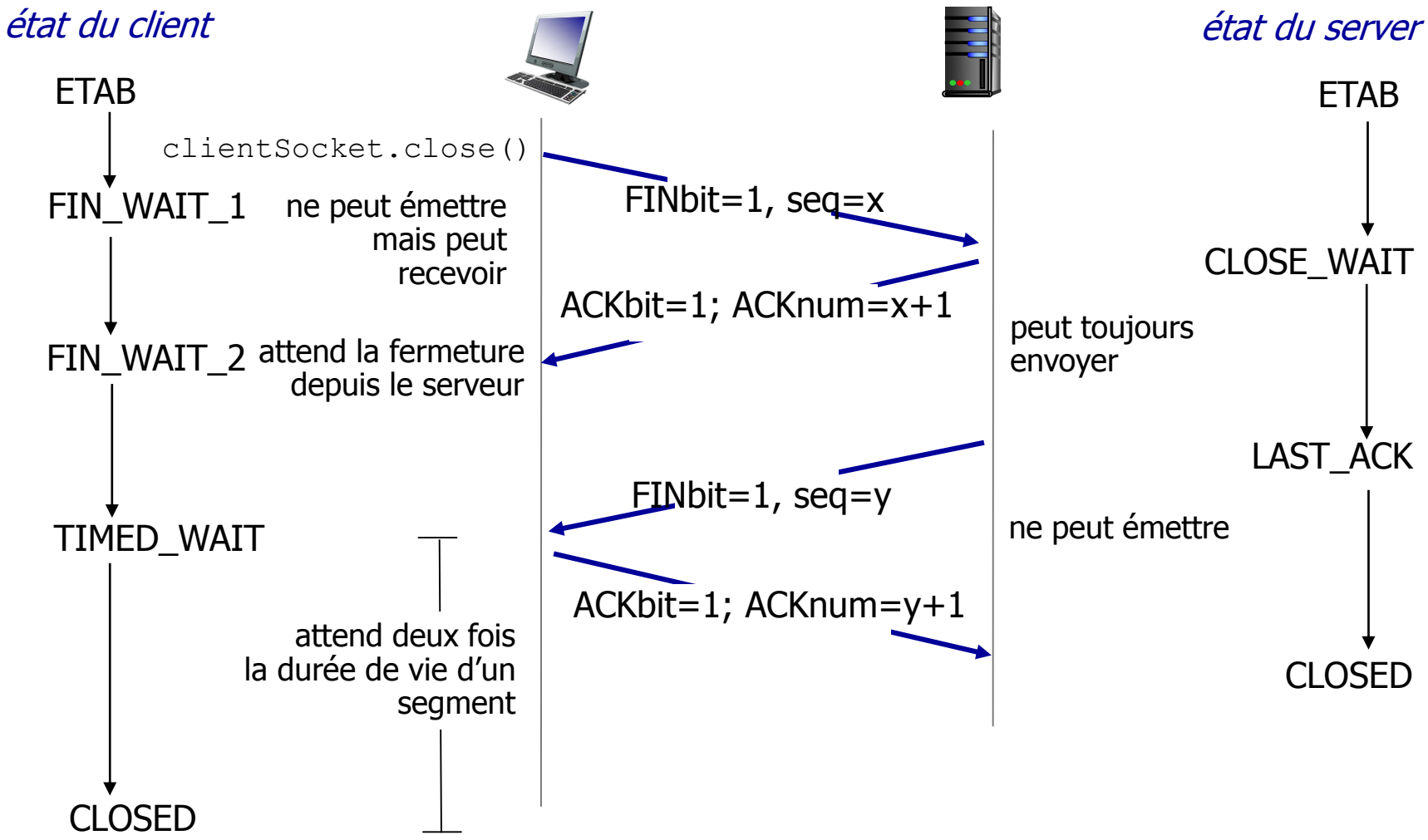
3.6 principes du contrôle de congestion

3.7 contrôle de congestion dans TCP

TCP 3-way handshake



TCP: Fermer une connexion



Chapitre 3: plan

3.1 services de la couche transport

3.2 multiplexage et démultiplexage

3.3 transport sans connexion: UDP

3.4 principes du transfert fiable des données

3.5 transport orienté connexion: TCP

- structure d'un segment
- transfert fiable
- contrôle de flux
- gestion de connexion

3.6 principes du contrôle de congestion

3.7 contrôle de congestion dans TCP

Principes du contrôle de congestion

congestion:

- informelle: "un grand nombre de sources envoient beaucoup de données à une vitesse excessive que le réseau ne peut supporter"
- différent du contrôle de flux!
- conséquences:
  - perte de paquets (buffer overflow aux routeurs)
  - délais longs (dans les files d'attente des routeurs)

Deux approches

Il existe deux approches pour réaliser un contrôle de congestion:

bout-à-bout:

- la congestion est détectée par les terminaux: perte et délai
- approche de TCP

assisté par le réseau:

- routeurs fournissent un feedback aux terminaux
  - un bit indiquant la congestion
  - indiquer le taux auquel l'émetteur doit se conformer

Chapitre 3: plan

3.1 services de la couche transport

3.2 multiplexage et démultiplexage

3.3 transport sans connexion: UDP

3.4 principes du transfert fiable des données

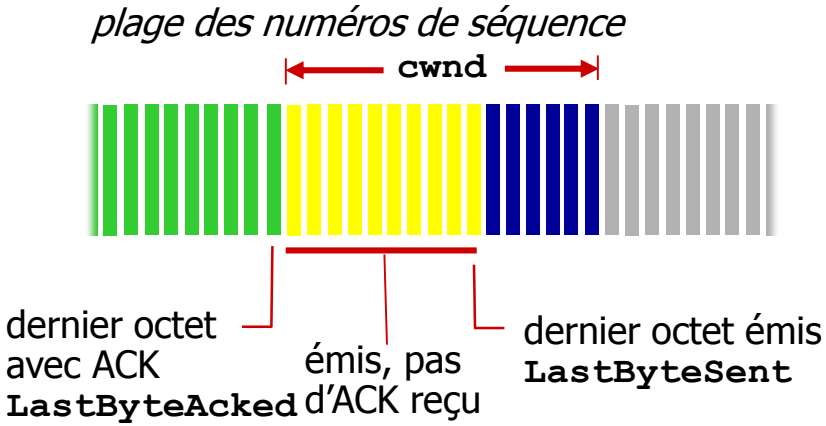
3.5 transport orienté connexion: TCP

- structure d'un segment
- transfert fiable
- contrôle de flux
- gestion de connexion

3.6 principes du contrôle de congestion

3.7 contrôle de congestion dans TCP

TCP contrôle de congestion



- limitation de l'émission:

$$\texttt{LastByteSent} - \texttt{LastByteAcked} \leq \texttt{cwnd}$$

- **cwnd** est dynamique, en fonction de la congestion perçue

taux d'émission de TCP :

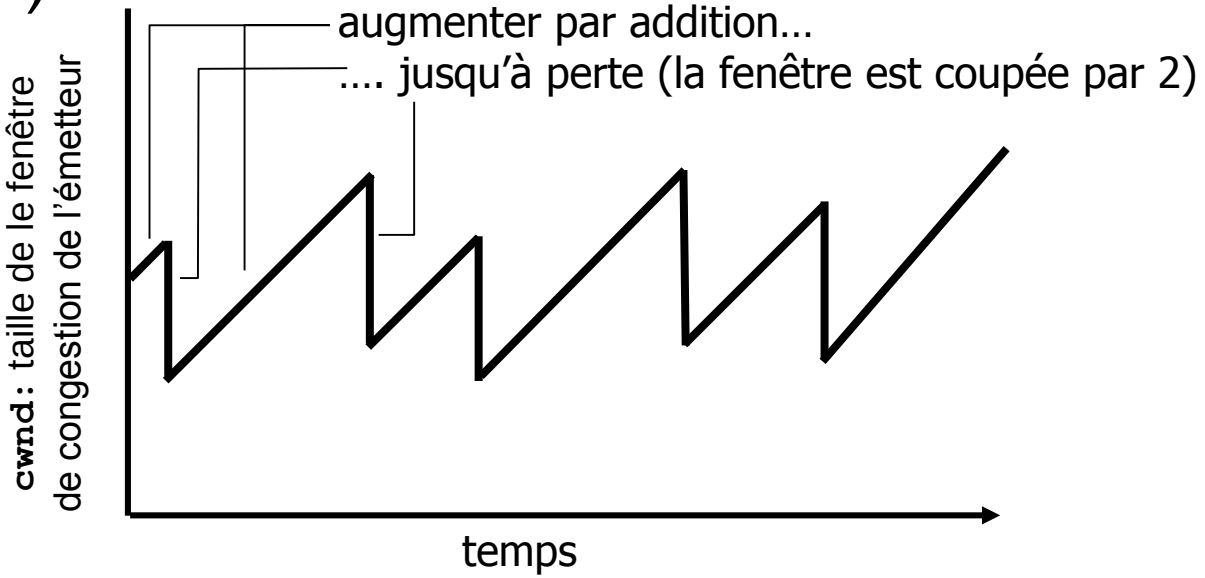
- envoyer cwnd octets, attendre RTT pour ACKS, puis envoyer plus d'octets

$$\text{rate} \approx \frac{\text{cwnd}}{\text{RTT}} \text{ bytes/sec}$$

TCP contrôle de congestion : additive increase multiplicative decrease

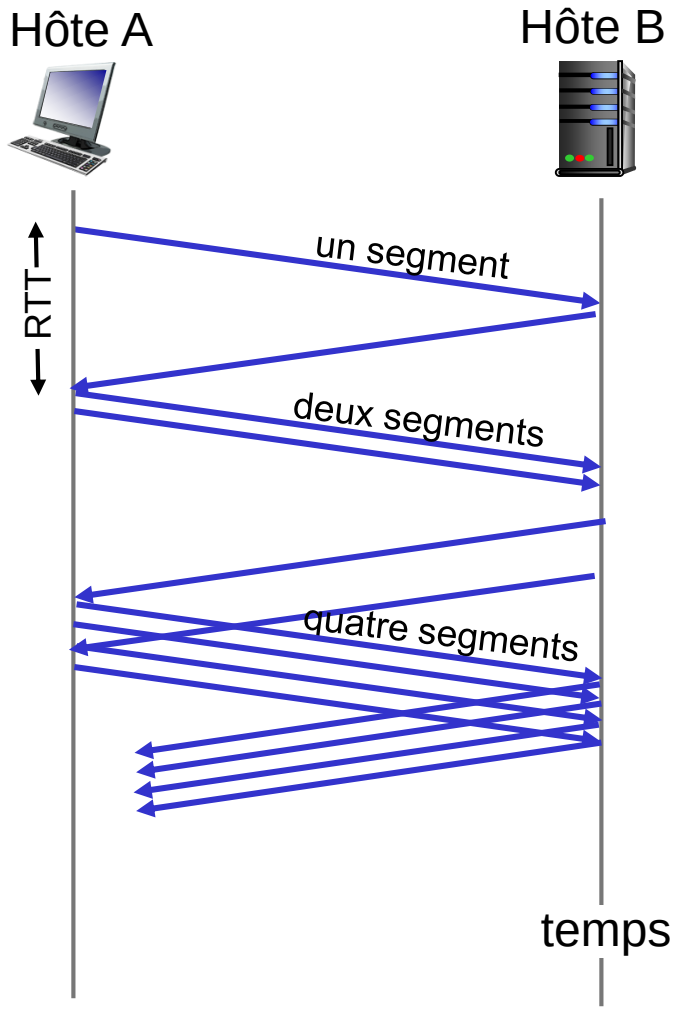
- *approche:* l'émetteur augmente son taux de transmission (taille de la fenêtre), jusqu'à la perte
  - *additive increase:* augmenter **`cwnd`** par 1 MSS à chaque RTT jusqu'à détection d'une perte
  - *multiplicative decrease:* couper **`cwnd`** à la moitié (après perte)

comportement AIMD en dents de scie: sonder la bande passante disponible



TCP Slow Start

- début de la connexion: augmenter le débit exponentiellement jusqu'à la première perte:
  - **`cwnd`** = 1 MSS
  - doubler **`cwnd`** chaque RTT (incrémenter le **`cwnd`** pour chaque ACK reçu)
- *résumé:* le débit initial est lent mais croît rapidement (exponentiellement)



TCP: détecter et réagir aux pertes

- perte indiquée par un timeout :
  - **`cwnd`** est réinitialisée à 1 MSS;
  - après la fenêtre croît exponentiellement (slow start) jusqu'à la limite, après elle croît linéairement
- perte indiquée par 3 acks dupliqués : TCP RENO
  - **`cwnd`** devient la moitié de la fenêtre puis croît linéairement
- TCP Tahoe réinitialise toujours **`cwnd`** à 1 (timeout ou 3 acks dupliqués)

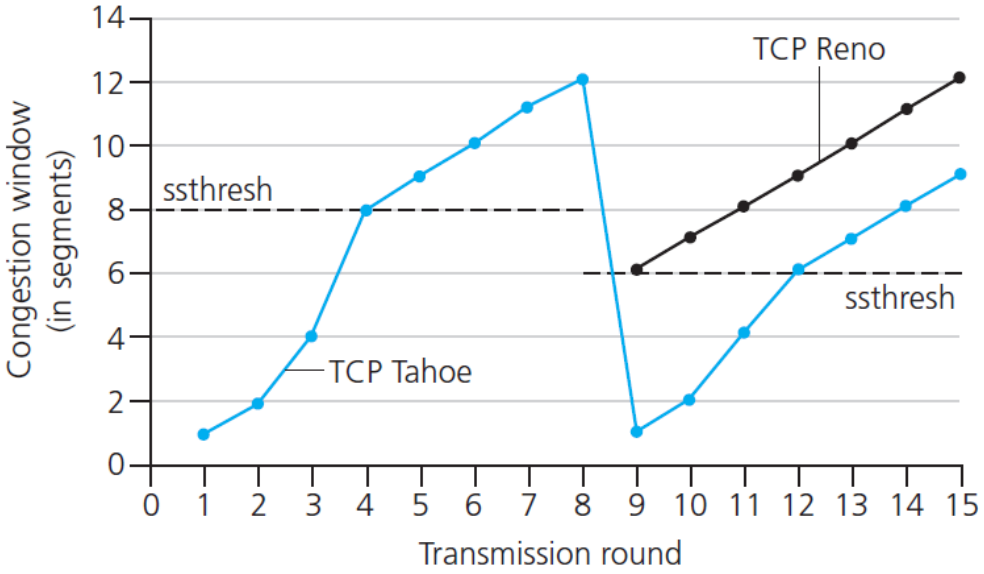
TCP: passer de slow start à CA

Q: Quand passer d'une croissance exponentielle à linéaire?

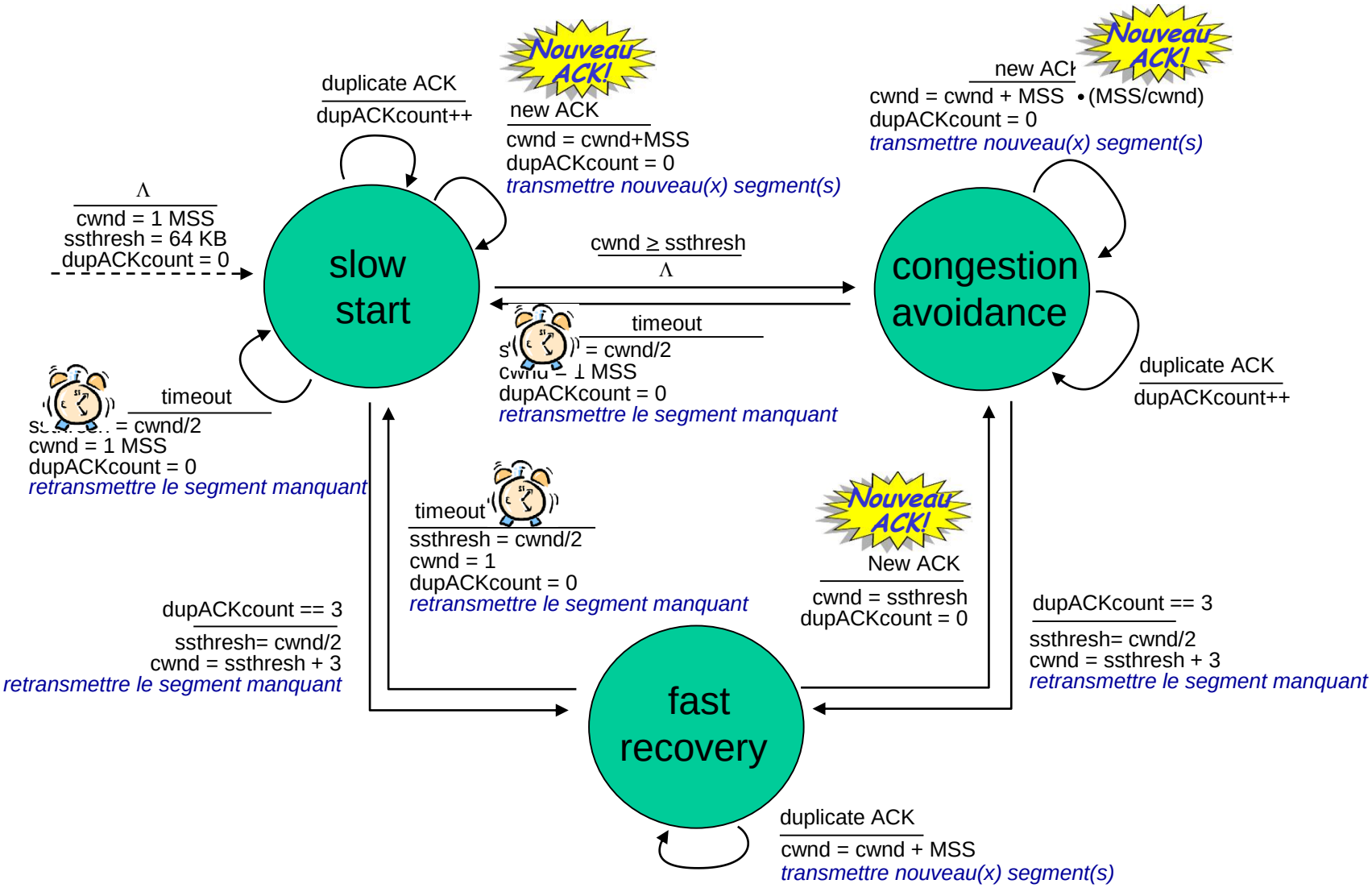
A: quand **`cwnd`** arrive à 1/2 de sa valeur avant timeout.

Implémentation:

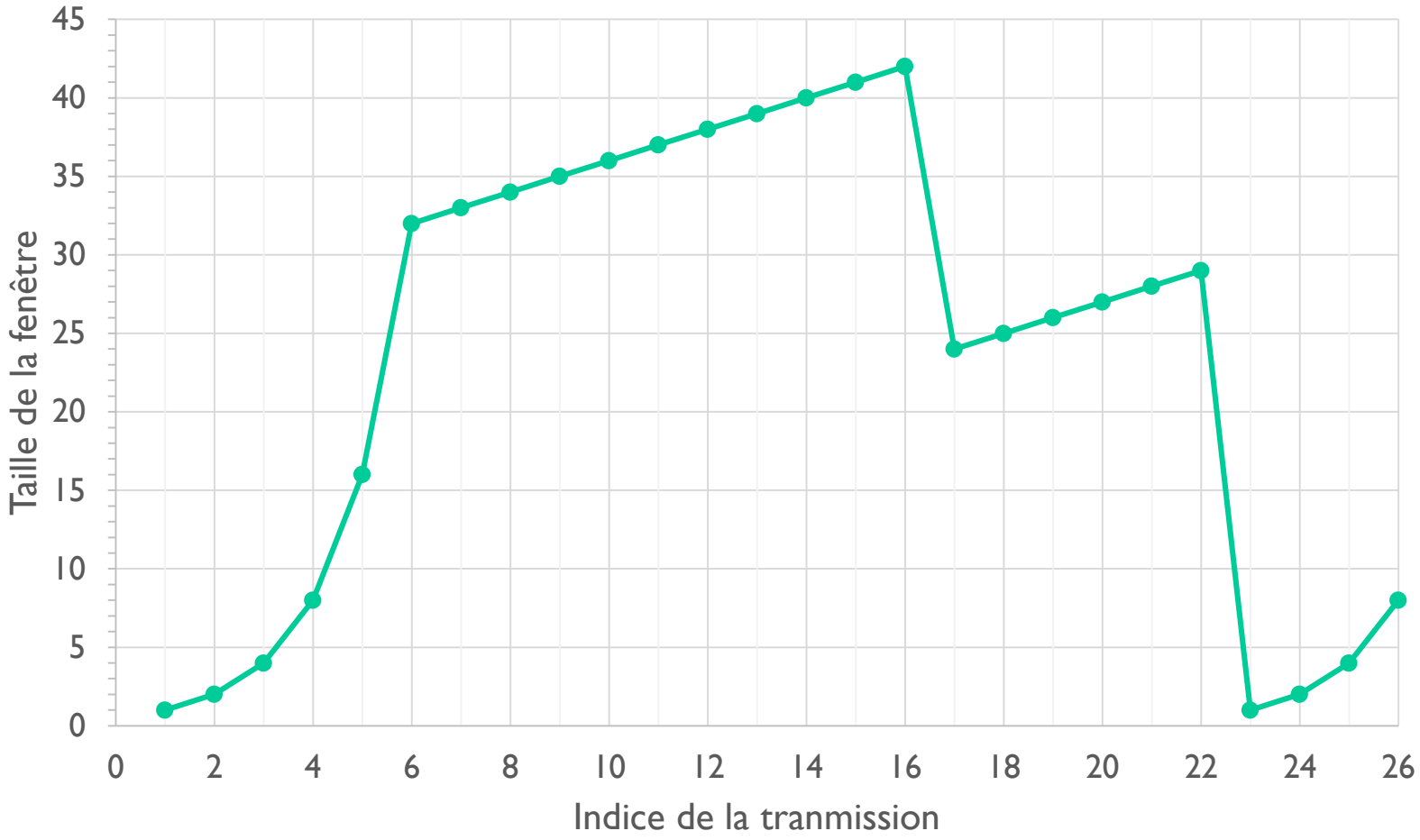
- variable **`ssthresh`** (limite)
- si perte, **`ssthresh`** devient 1/2 de **`cwnd`** (taille enregistrée juste avant la perte)



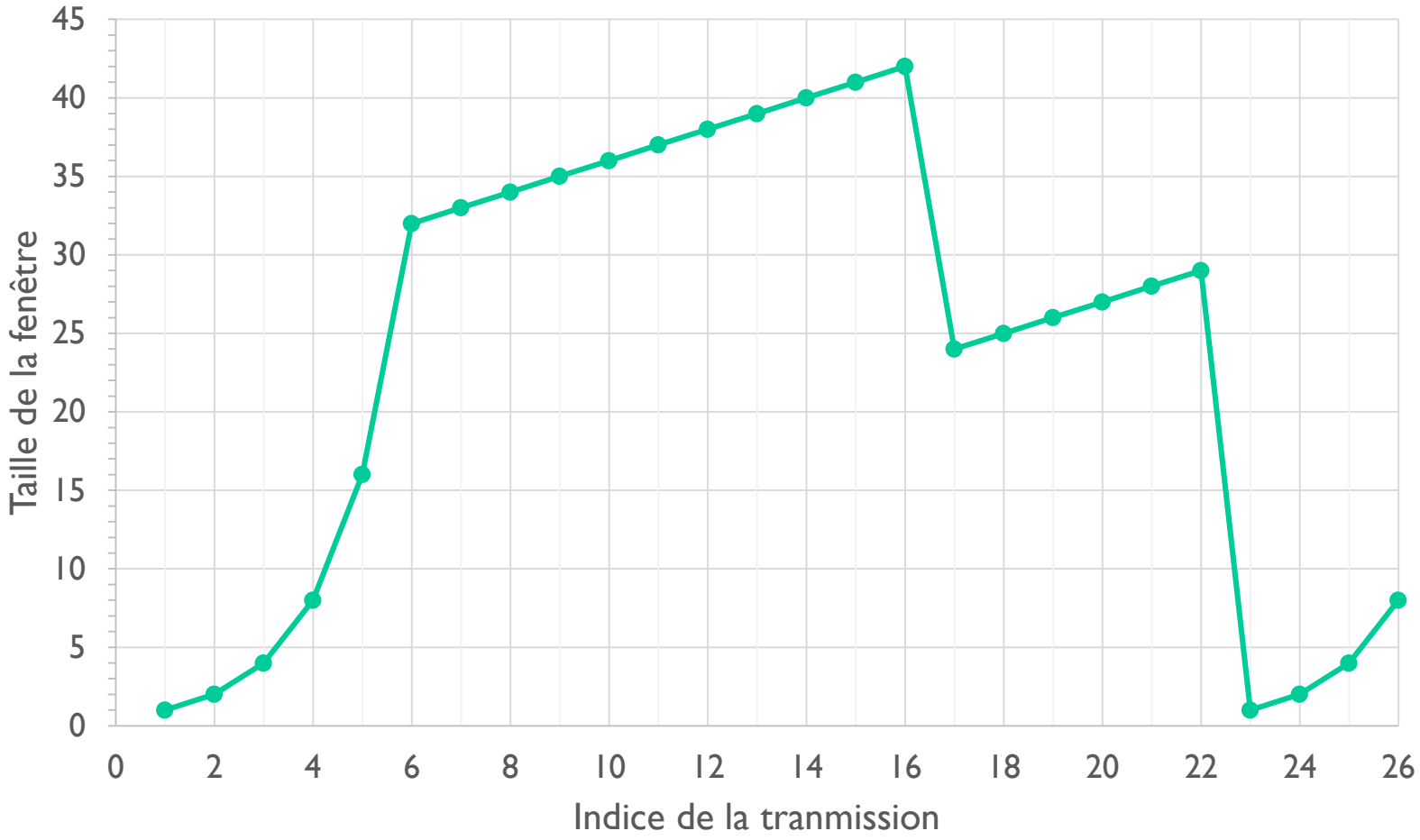
Résumé du contrôle de congestion TCP



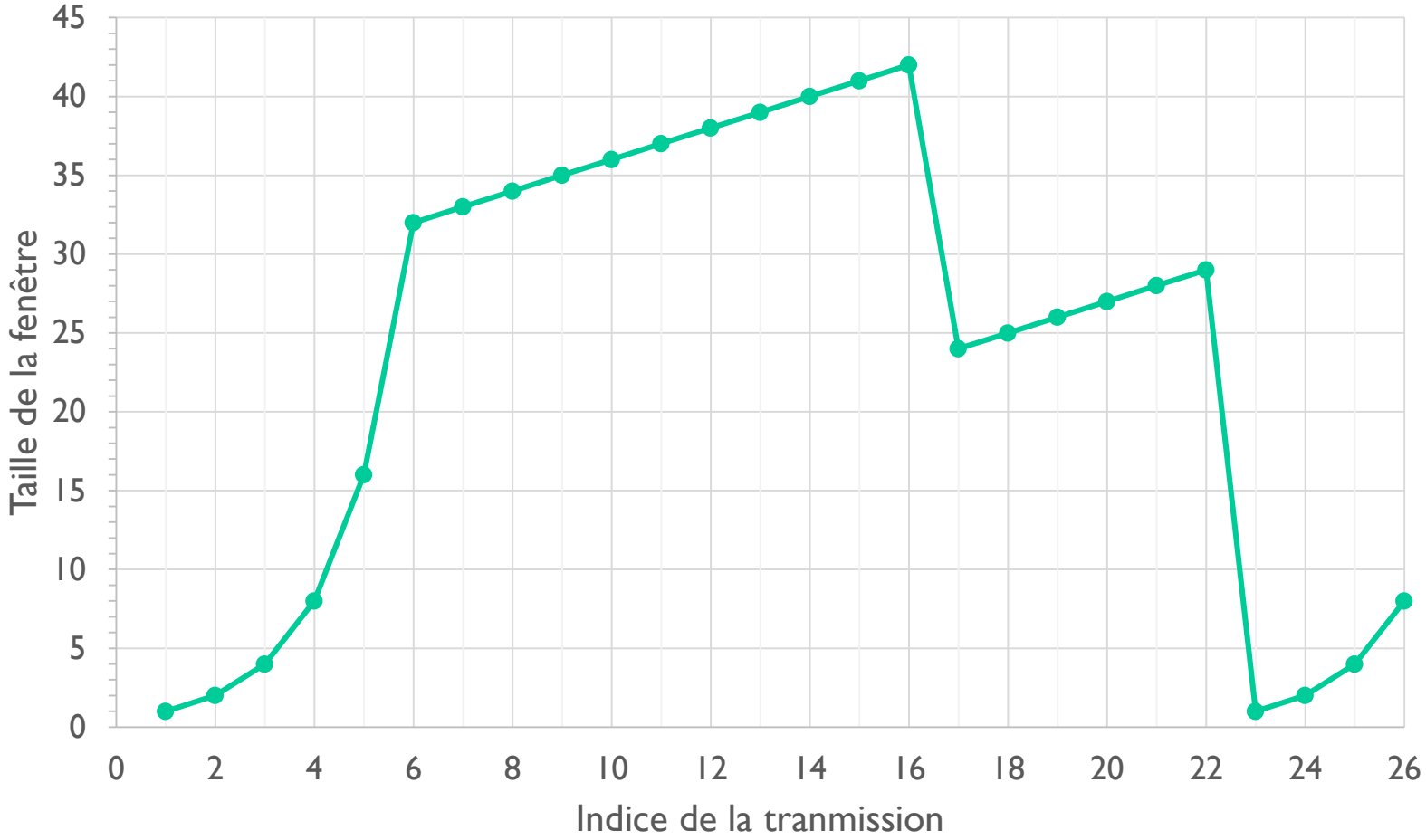
Exercices



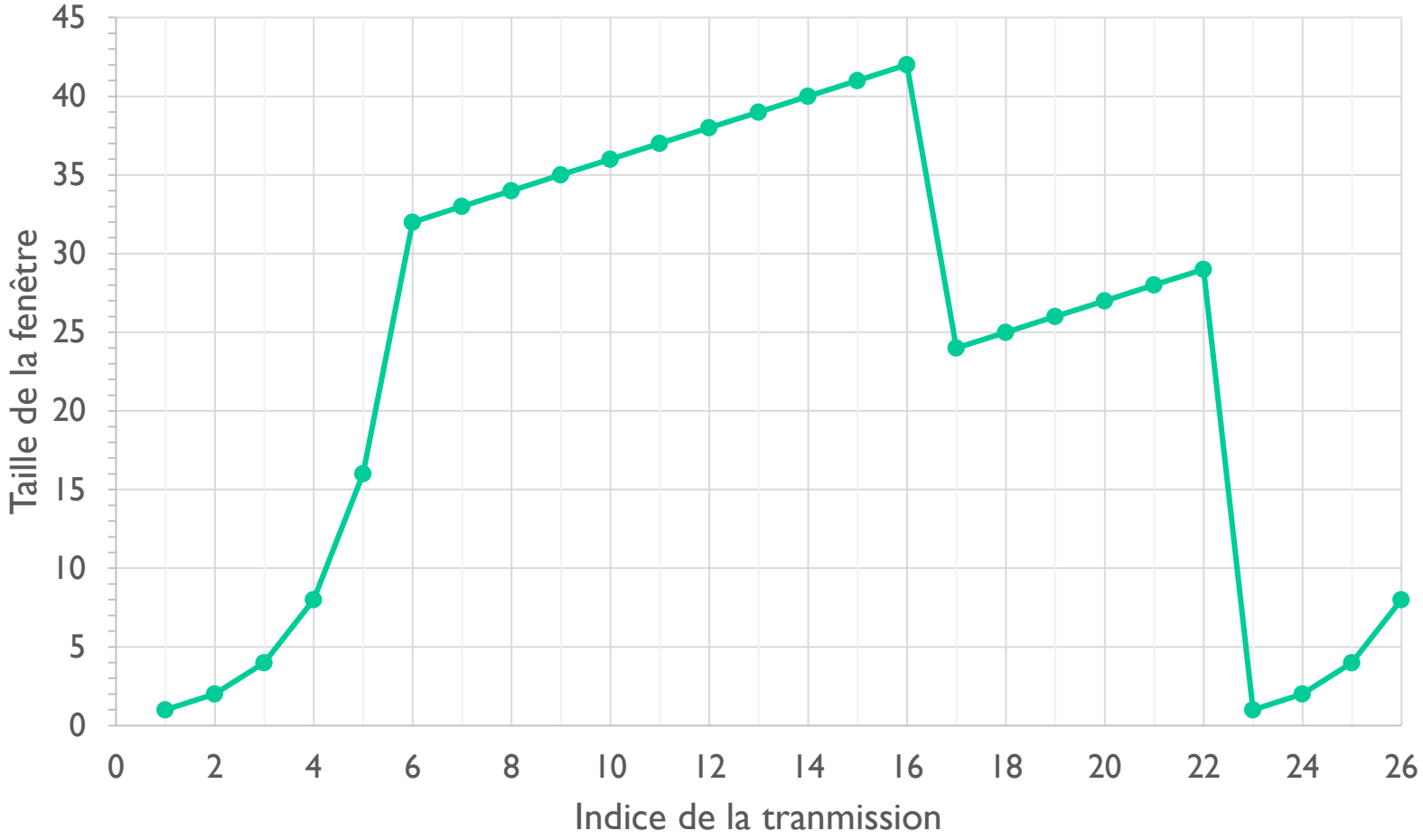
- Identifier les intervalles slow start
  - [1,6], [23,26]
- Identifier les intervalles congestion avoidance
  - [6,16], [17,22]



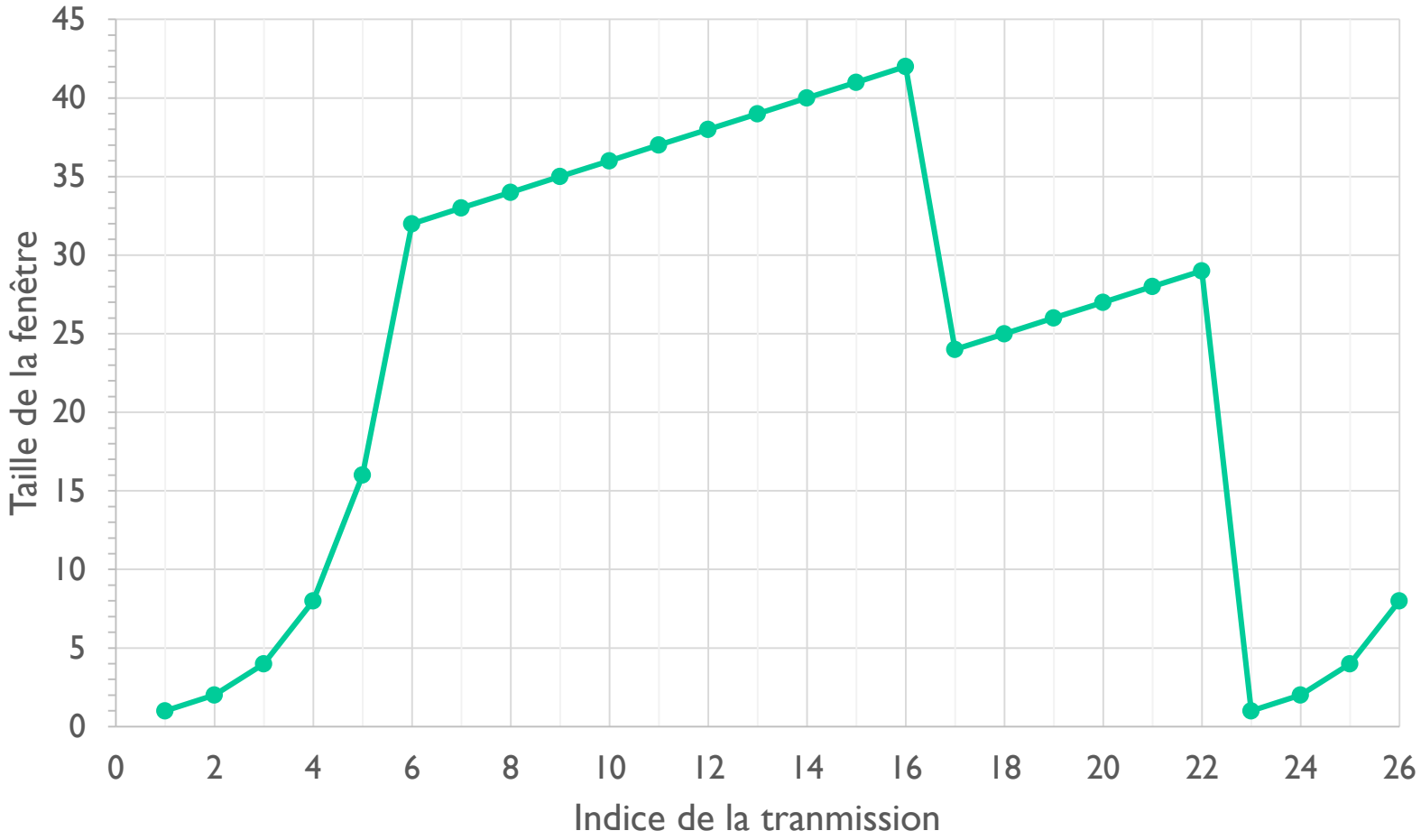
- Comment la perte est détectée après 16 et après 22?
  - Après 16, par acks dupliqués
  - Après 22, par timeout



- ssthresh initial? à 18? et à 24?
  - intial : 1
  - à 18 : 21
  - à 24 : 14



- À quelle transmission, le segment 70 est envoyé?
- Supposons 3Acks dupliqués à la transmission 26, donner cwnd et ssthresh



- Supposons Tahoe
  - cwnd et ssthresh à 19?
  - timeout à 22, combien de segments envoyés entre 17 et 22?

Exercice 2

- Un paquet de taille L
- part de A, passe par deux routeurs puis arrive à B
  - d_i = longueur du lien i
  - s_i = vitesse de propagation sur le lien i
  - R_i = débit sur le lien i
  - d_{proc} = délai de traitement dans chaque routeur
- Les routeurs utilisent « store and forward »

Q1: *Formule du délai total de bout en bout?*

Exercice 2

- L = 1500 octets
- vitesse de propagation = 2,5 10^8 m/s
- $R_1 = R_2 = R_3$ = 2Mbps
- d_{proc} = 3 msec
- d_1 = 5000 km, d_2 = 4000 km, d_2 = 1000 km

Q2: *valeur du délai total de bout en bout?*

Exercice 2

- $R_1 = R_2 = R_3 = R$
- d_{proc} = 0 msec
- sans « store and forward »

Q3: *formule du délai total de bout en bout?*

Exercice 2

- L = 1500 octets
- vitesse de propagation = 2,5 10^8 m/s
- $R_1 = R_2 = R_3$ = 2Mbps
- d_{proc} = 0 msec
- d_1 = 5000 km, d_2 = 4000 km, d_2 = 1000 km

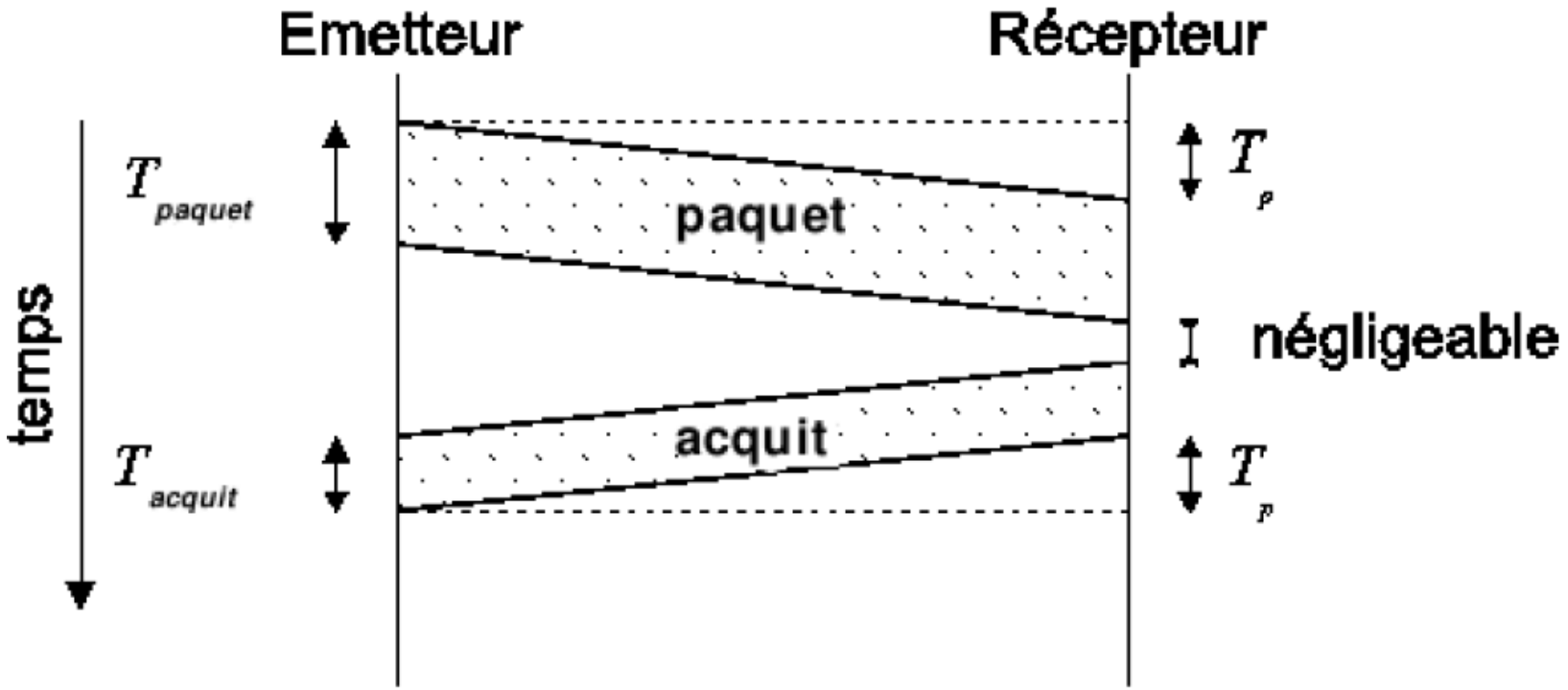
Q4: *valeur du délai total de bout en bout?*

Exercice 3

- Un canal entre émetteur et récepteur de 4Kbps et d_{prop}= 20msec
- Chaque paquet a un en-tête de 6 octets
- Chaque accusé de réception fait 8 octets
- On utilise « stop and wait »

Q: *quelle est la taille de paquet N qui permet une utilisation (fraction du temps durant laquelle la source est occupé) de 50%?*

Exercice 3


Emetteur
Récepteur
temps
T_{paquet}
T_{acquit}
paquet
acquit
T_p
T_p
négligeable

Exercice 3

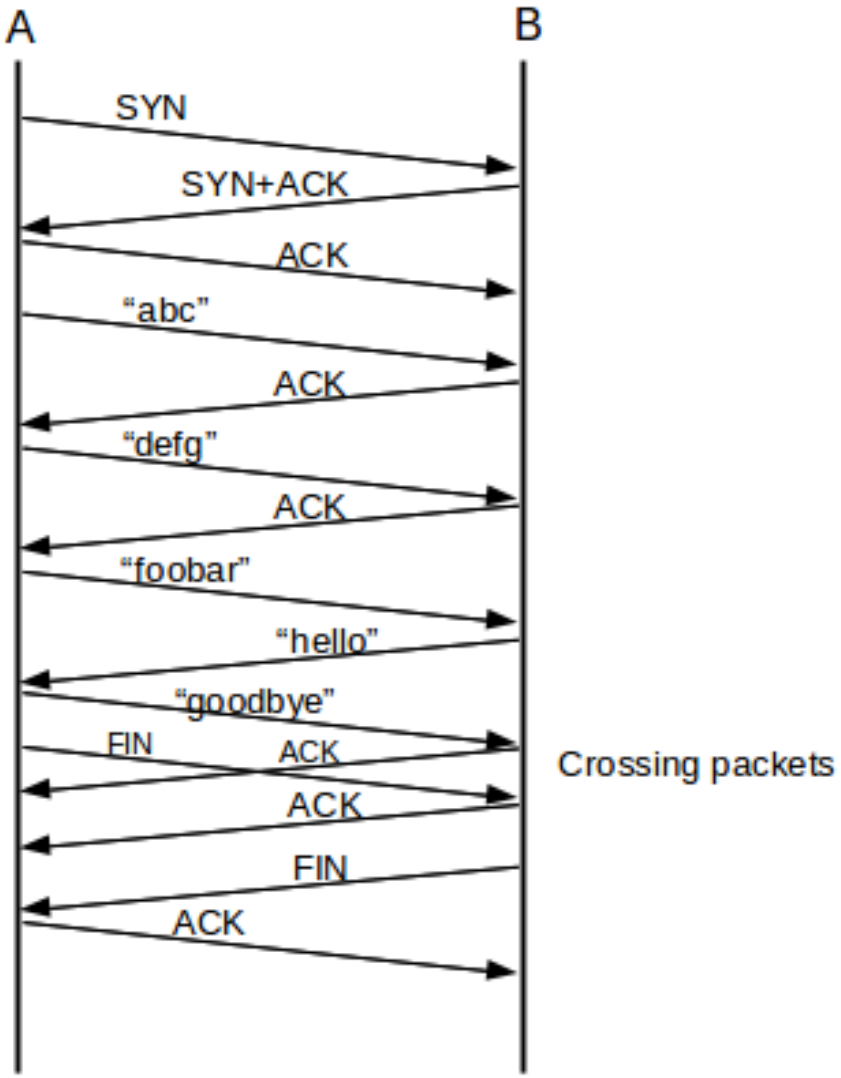
- $U = \dfrac{T_p}{T_p + 2\,T_{prop} + T_{ack}}$
- $U = \dfrac{\frac{N+48}{4000}}{\frac{N+48}{4000} + 2 \cdot 0{,}04 + \frac{64}{4000}} = \dfrac{1}{2}$
- $\dfrac{N+48}{N+272} = \dfrac{1}{2}$

- $N = 176\ bits = 22\ octets$

Exercice 4

	A	B
1	SYN, **seq=0**	
2		SYN+ACK, **seq=0, ack=__**
3	ACK, **seq= __, ack=__**	
4	"abc", **seq= __, ack=__**	
5		ACK, **seq= __, ack=__**
6	"defg", **seq= __, ack=__**	
7		**seq= __, ack=__**
8	"foobar", **seq= __, ack=__**	
9		**seq= __, ack=__**,"hello"
10	**seq= __, ack=__**,"goodbye"	
11,12	**seq= __, ack=__**, FIN	**seq= __, ack=__** ;;ACK de "goodbye"
13		**seq= __, ack=__** ;; ACK de FIN
14		**seq= __, ack=__**, FIN
15	**seq= __, ack=__** ;; ACK de FIN	

Exercice 4


A
B
SYN
SYN+ACK
ACK
“abc”
ACK
“defg”
ACK
“foobar”
“hello”
“goodbye”
FIN
ACK
ACK
FIN
ACK
Crossing packets

Exercice 4 (solution)

	A	B
1	SYN, **seq=0**	
2		SYN+ACK, **seq=0, ack=1**
3	ACK, **seq= 1, ack=1**	
4	"abc", **seq= 1, ack=1**	
5		ACK, **seq= 1, ack=4**
6	"defg", **seq= 4, ack=1**	
7		**seq= 1, ack=8**
8	"foobar", **seq= 8, ack= 1**	
9		**seq= 1, ack=14**, "hello"
10	**seq= 14, ack= 6**, "goodbye"	
11,12	**seq= 21, ack= 6**, FIN	**seq= 6, ack=21** ;;ACK de "goodbye"
13		**seq= 6, ack=22** ;; ACK de FIN
14		**seq= 6, ack=22**, FIN
15	**seq= 22, ack=7** ;; ACK de FIN	