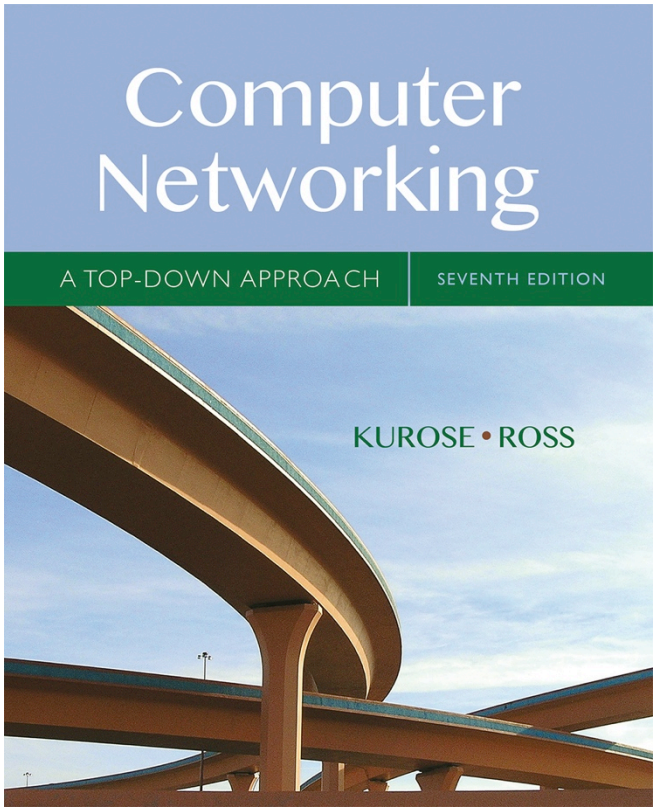
Cours 3 Couche Transport : TCP



Planches adaptées de « Computer Networking : A Top Down Approach »

Computer Networking: A Top Down Approach

7th edition
Jim Kurose, Keith Ross
Pearson/Addison Wesley
April 2016

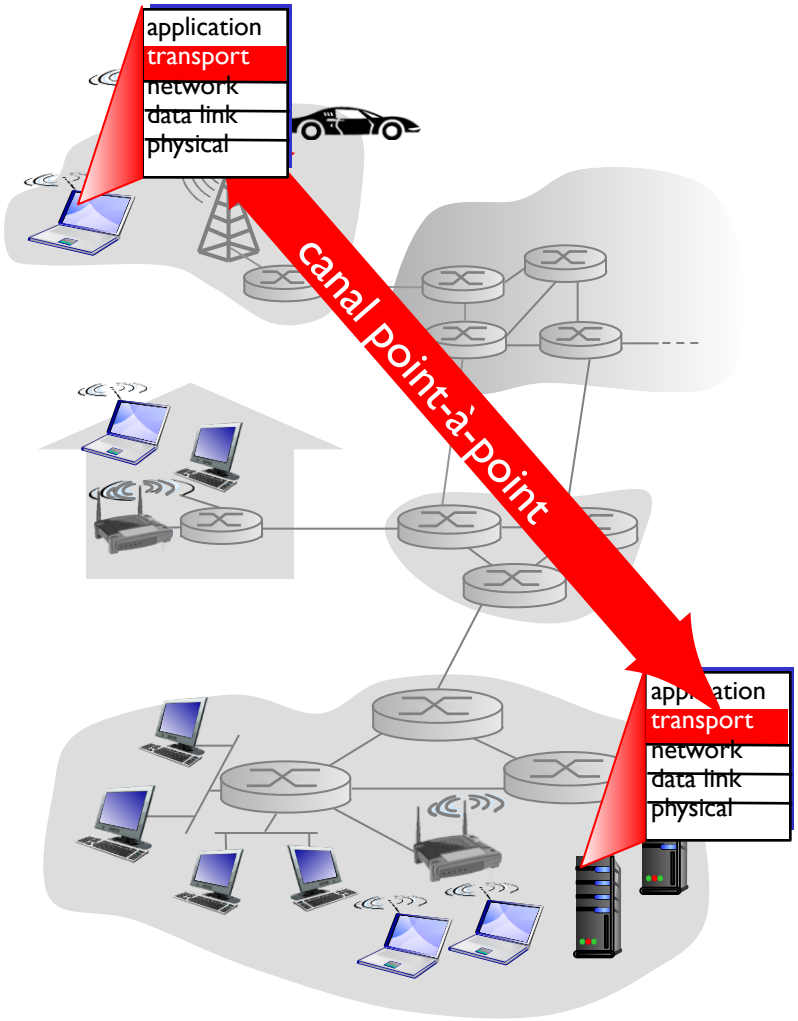
Couche transport : TCP

objectifs :

- Comprendre les services rendus par la couche transport
  - multiplexage, démultiplexage
  - transfert de données fiable
  - contrôle de flux
  - contrôle de congestion
- Appréhender leur implémenation par le protocole TCP

Rappel : services de la couche transport

- canaux de *communication logiques* entre des processus s'exécutant sur plusieurs machines
- protocoles de la couche transport :
  - coté émetteur : découpe les message de l'appli en *segments*, et les passe à la couche réseau
  - coté récepteur: assemble les segments en messages, livre à la couche appli.
- Plusieurs protocoles existent
  - Internet: TCP et UDP

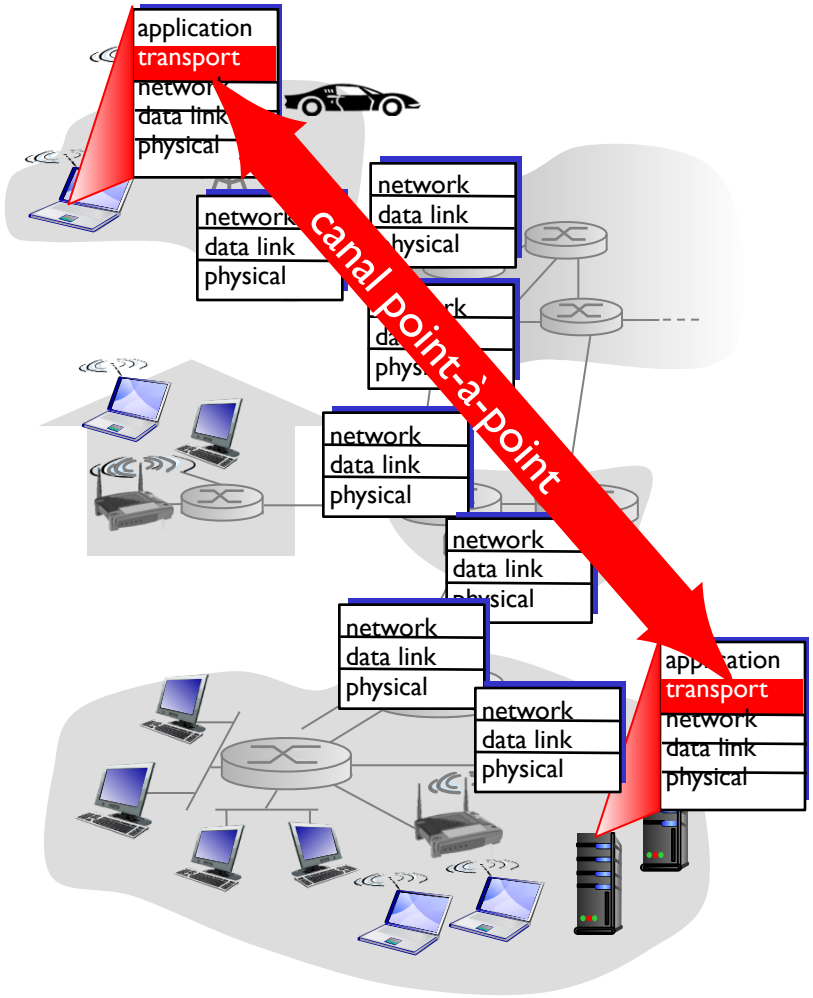


Rappel couche transport vs. réseau

- *couche réseau:* canal logique entre machines (« *hosts* »)
- *couche transport:* canal logique entre processus
  - repose sur les services de la couche réseau
  - fournit des garanties supplémentaire

Protocoles de la couche transport

- fiable, livraison dans l'ordre (TCP)
  - controle de congestion
  - contrôle de flux
  - établissement d'une connection
- non-fiable, livraison non ordonnée (UDP)
  - extension basique d'IP
- non-fournis :
  - garanties sur la latence
  - garanties sur la bande passante



(De)Multiplexage

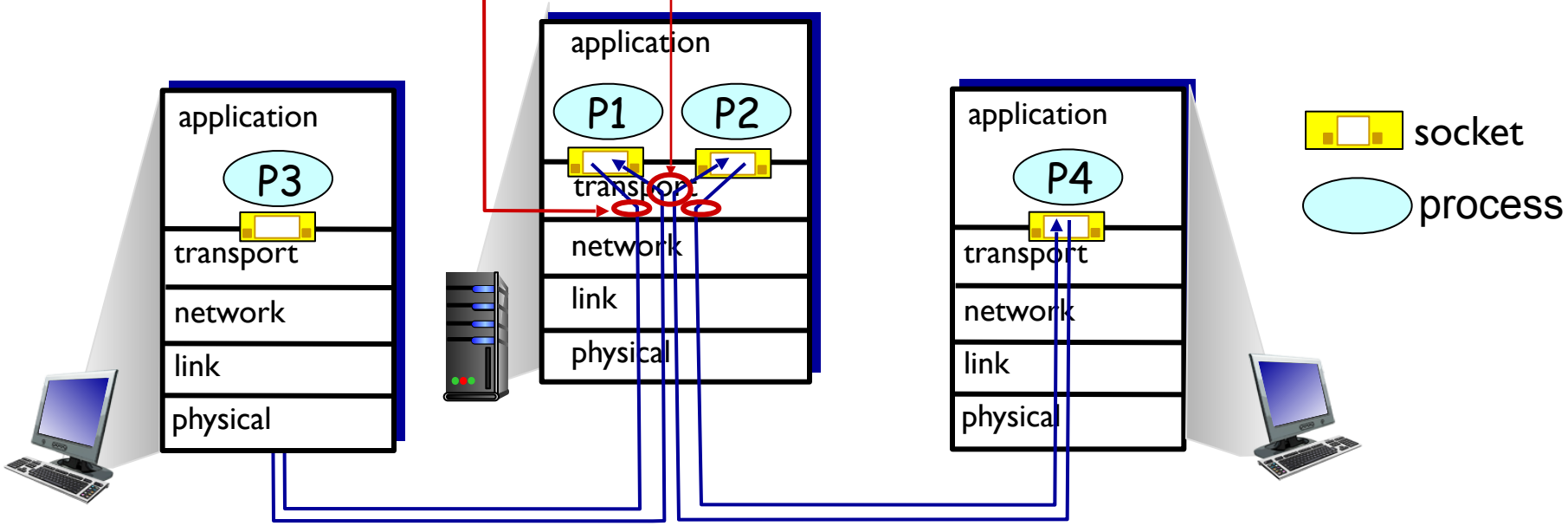
Multiplexage/demultiplexage

émetteur : multiplexage

- gestion de données de plusieurs sockets
- ajout en-tête transport

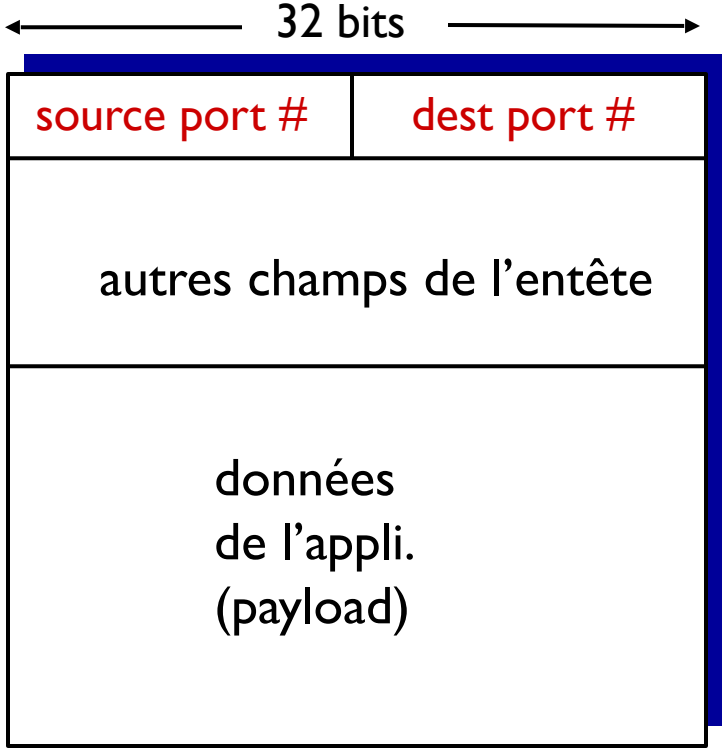
récepteur : démultiplexage

utiliser l'en-tête pour livrer les segments reçus à la bonne socket



Démultiplexage

- Lors d'un réception d'un paquet IP
  - chaque paquet contient l'**adresse IP** de l'émetteur et du récepteur
  - chaque paquet contient un **segment** (unité de données de la couche transport)
  - chaque segment contient le **port** de l'émetteur et du récepteur
- L'hôte utilise les *adresses IP et les numéros de port* pour livrer le segment à la socket appropriée

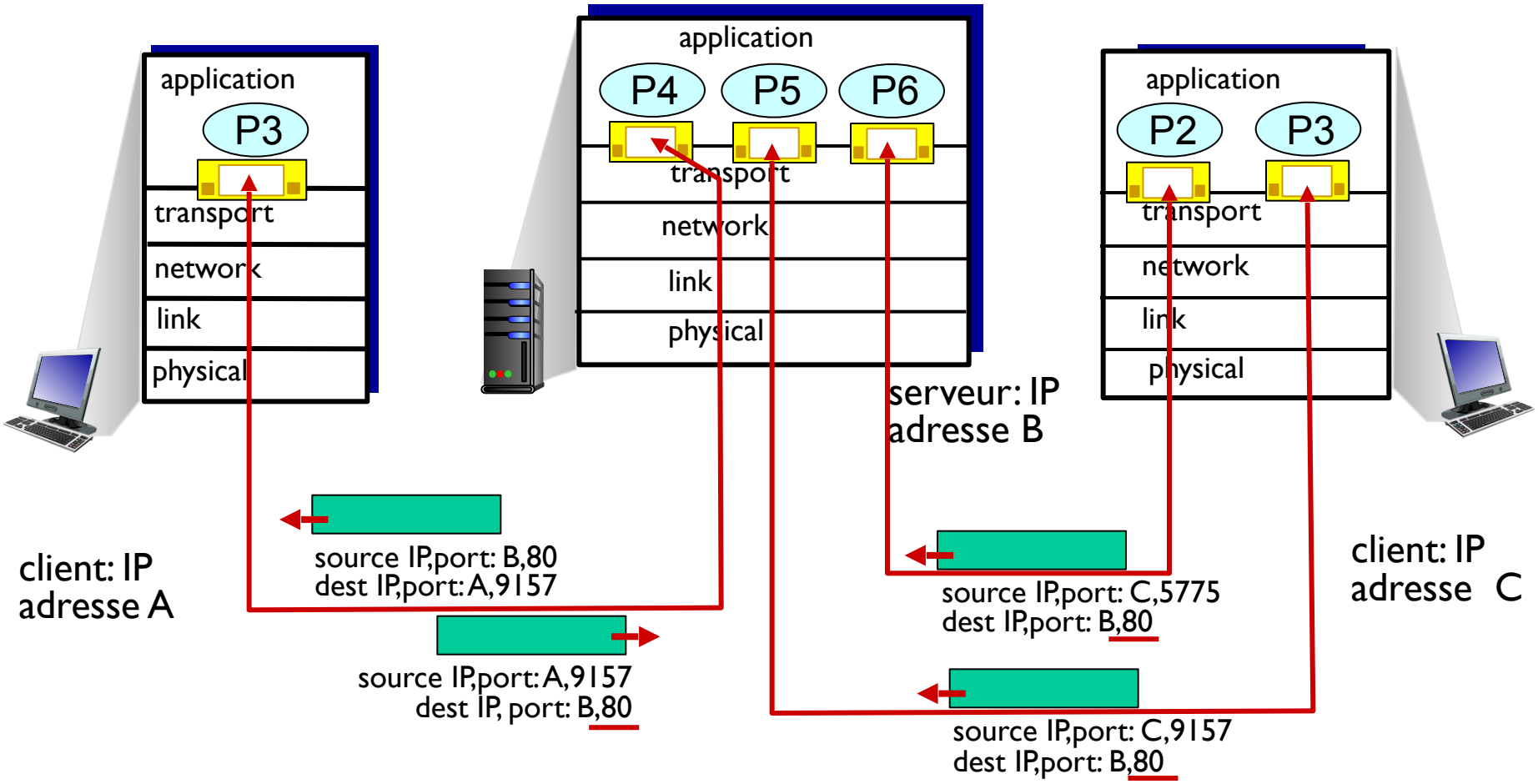


format d'un segment TCP/UDP

Démultiplexage TCP

- Socket TCP identifiée par
  - adresse IP émetteur
  - numéro port émetteur
  - adresse IP récepteur
  - numéro port récepteur
- Démux.: récepteur utilise ces quatre valeurs pour livrer le segment à la bonne socket
- Sur un serveur, plusieurs sockets peuvent être ouvertes simultanément
- Ex : serveur web.
  - une socket pour chaque client
  - voire une socket pour chaque requête

Démultiplexage TCP : exemple



3 segments à destination de l'adresse IP B, port 80 sont livrés à des sockets **différentes**

Detection d'erreurs de transmission

Somme de contrôle (checksum)

But: détecter les erreurs (i.e., bits modifiés) lors de la transmission des segments

émetteur:

- contenu du segment (header inclus) comme seq. de mots de 16bits
- checksum : addition (complément à un de la somme) du contenu
- checksum mis dans un champ de l'entête

récepteur:

- calcule le checksum du segment reçu
- vérifie si checksum calculé = champ checksum de l'entête :
  - NON - erreur détectée
  - OUI - pas d'erreur détectée *mais … ?*

Somme de contrôle : exemple

exemple: addition de 2 entiers de 16-bits

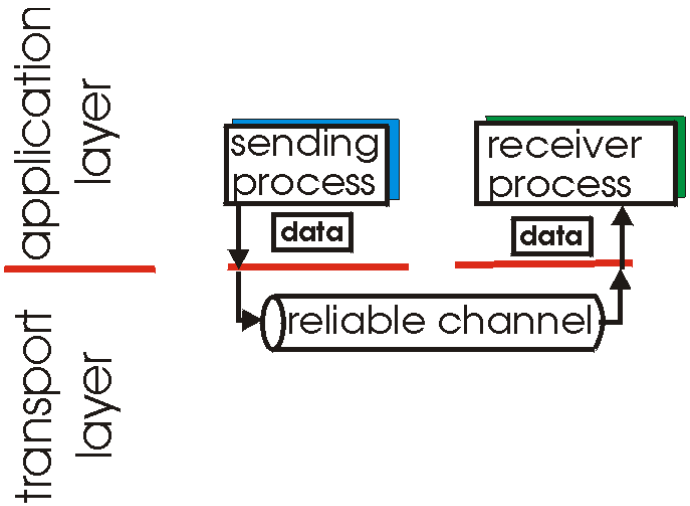
	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
somme	1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Note: dans le cas d'une retenue, elle est ajoutée au résultat final

Transfert fiable de données
(rdt : reliable data transfer)

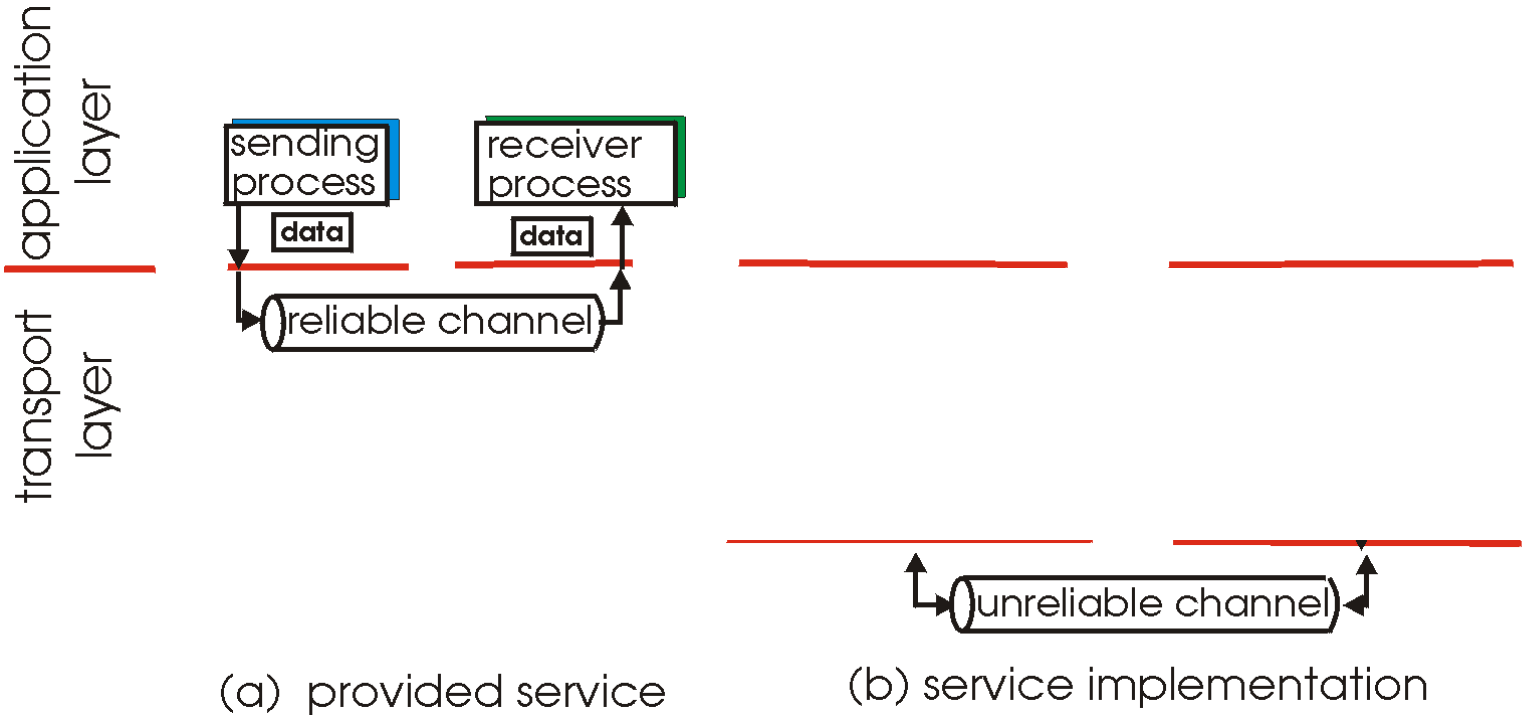
Transfert de données *fiable*

- But : établir un canal **fiable** rentre émetteur et récepteur
- A partir du canal **non-fiable** de la couche inférieure



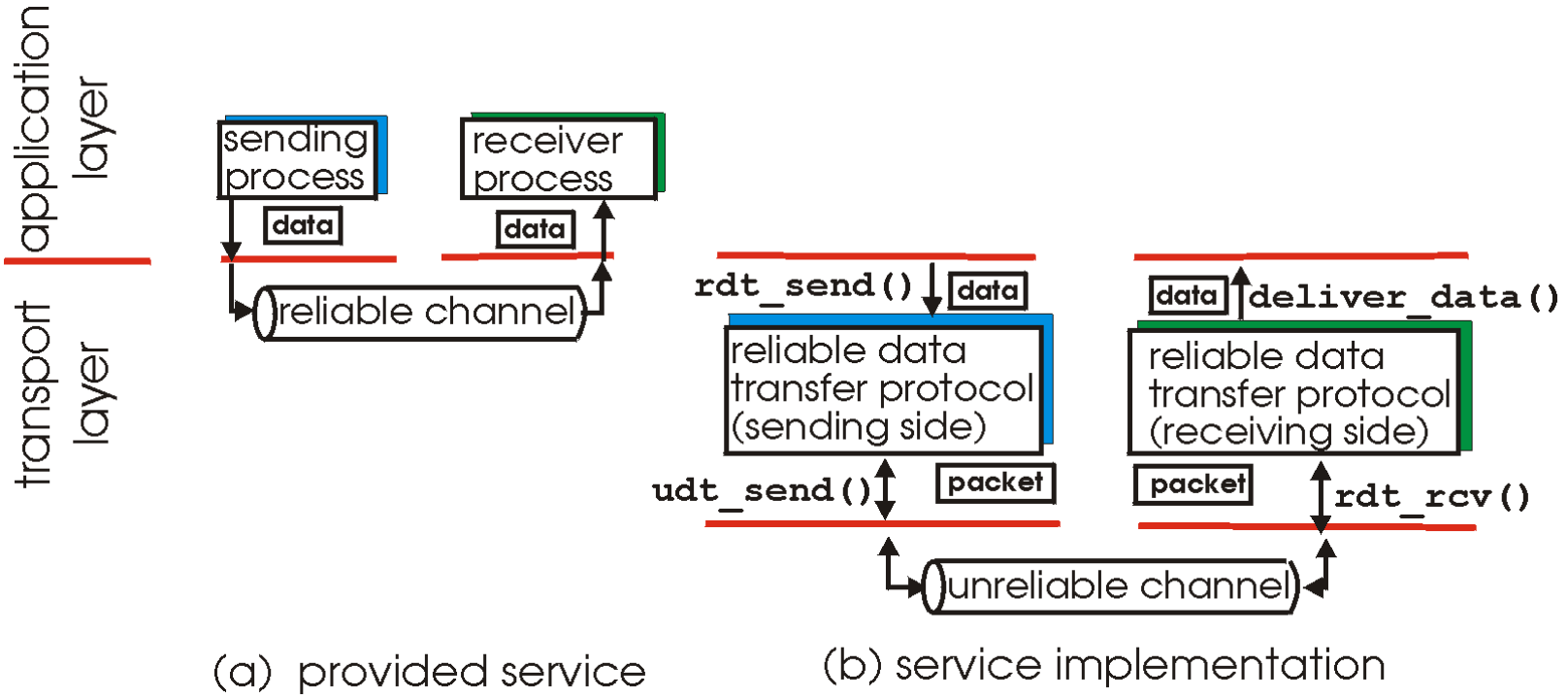
Transfert de données *fiable*

- But : établir un canal **fiable** rentre émetteur et récepteur
- A partir du canal **non-fiable** de la couche inférieure



Transfert de données *fiable*

- But : établir un canal **fiable** rentre émetteur et récepteur
- A partir du canal **non-fiable** de la couche inférieure

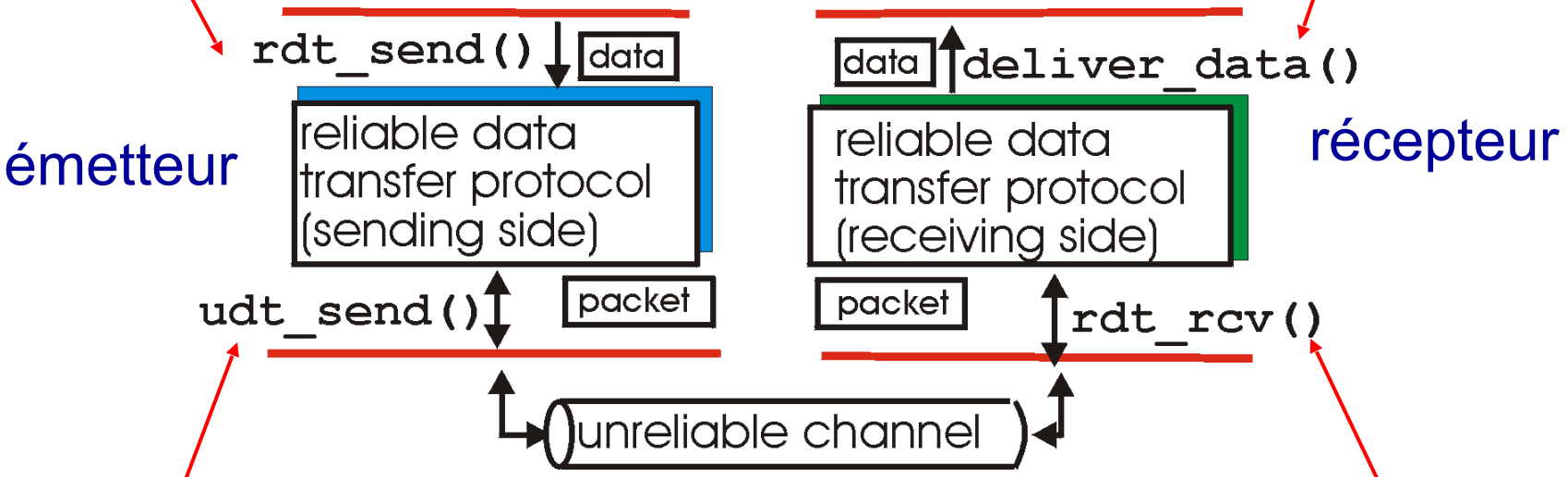


- rdt_ *reliable data transfert*
- udt_ *unreliable data transfert*

Transfert de données fiable: primitives

`rdt_send()` : appelé depuis la couche appli. données à livrer à la couche appli. du récepteur

`deliver_data()` : appelé pour livrer les données à la couche appli.



`udt_send()` : appelé pour transférer des paquet sur le canal non-fiable

`rdt_rcv()` : appelé quand des paquets arrivent coté récepteur

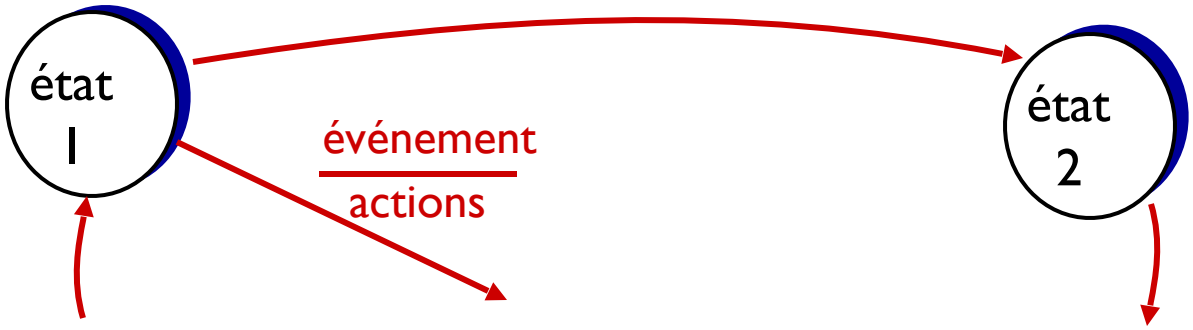
Construction incrémentale

dans la suite:

- développement d'un protocole de transfert de donnée fiable : protocole **rdt**
- transfer uni-directionnel (émetteur → récepteur)
  - mais échanges bidirectionnel de messages de contrôle
- spécification sous forme d'automate fini déterministe (AFD)

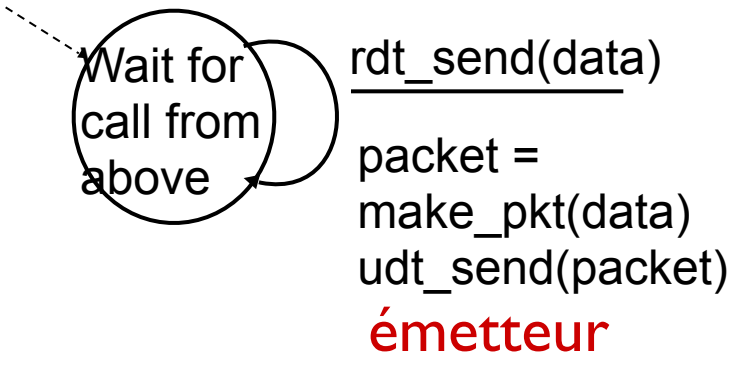
état: état suivant déterminé par le prochain évènement

évènement : arrivée d'un message, appel de la couche appli., etc.

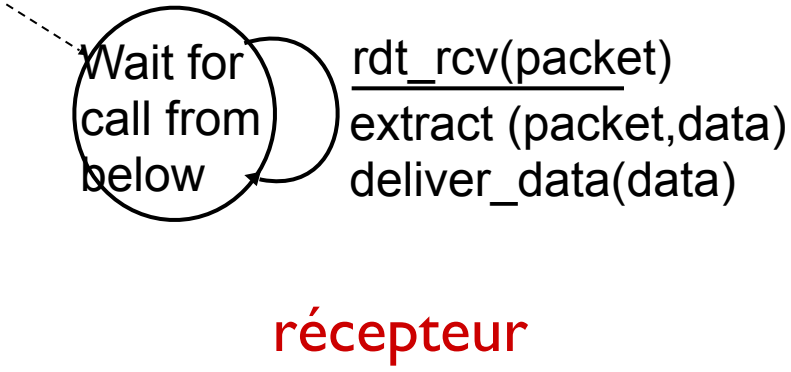


rdt1.0: transfert fiable à partir d'un canal fiable

- canal sous-jacent fiable
  - pas d'erreur sur les bits transmis
  - pas de perte de paquets



émetteur



récepteur

rdt2.0: canal avec erreurs

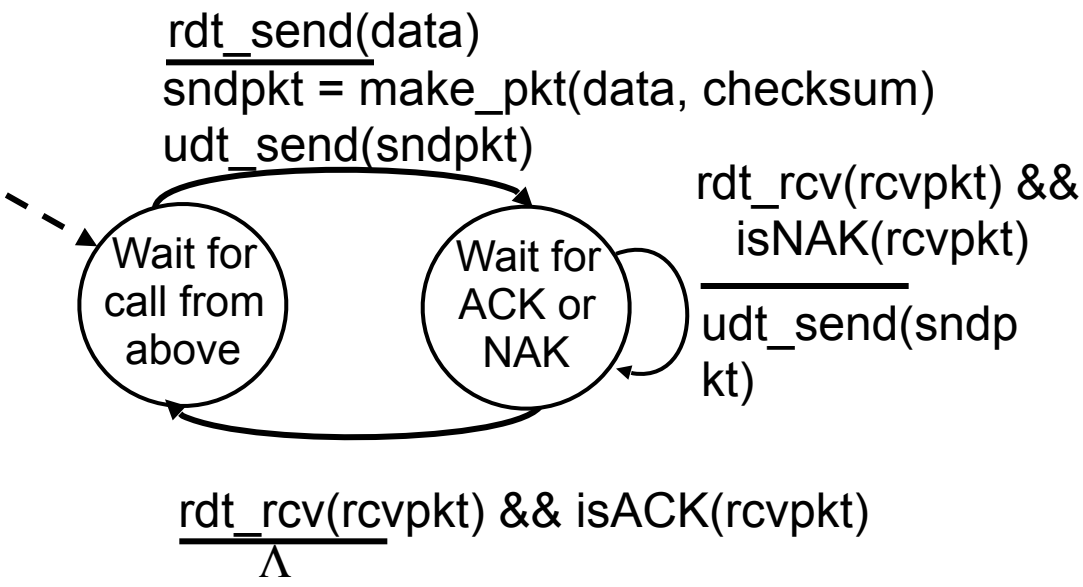
- bits peuvent être modifiés lors de leur transmission sur le canal sous-jacent
  - détection des erreurs à partir du checksum

Rétablissement après erreur dans une conversation ?

rdt2.0: canal avec erreurs

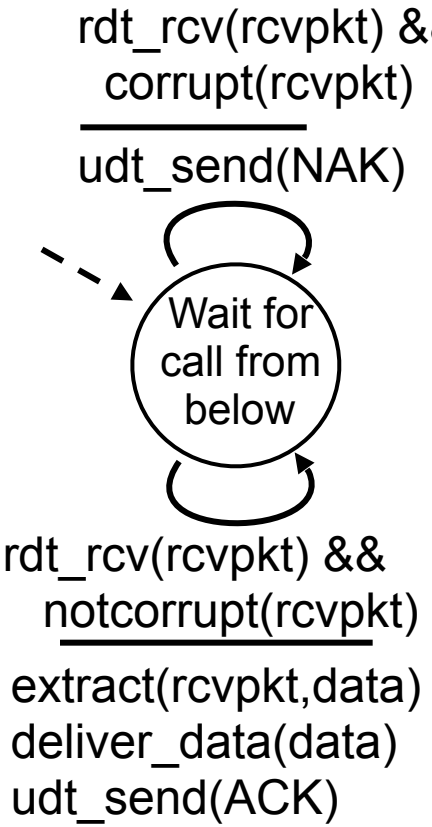
- bits transmis peuvent être modifiés par le canal sous-jacent
  - détection d'erreurs à l'aide du checksum
- question : se remettre d'une erreur ?
  - *accusé de réception (ACKs):* mssg. par lequel récepteur indique à l'émetteur que le paquet reçu est OK
  - *accusé de réception négatif (NAKs):* mssg. indiquant que le paquet reçu a des erreurs
  - retransmission des paquets lors de réception de NAK
- nouveaux mécanismes dans `rdt2.0:`
  - détection d'erreurs
  - feedback: messages de contrôle (ACK,NAK) récepteur →émetteur

rdt2.0: spécification

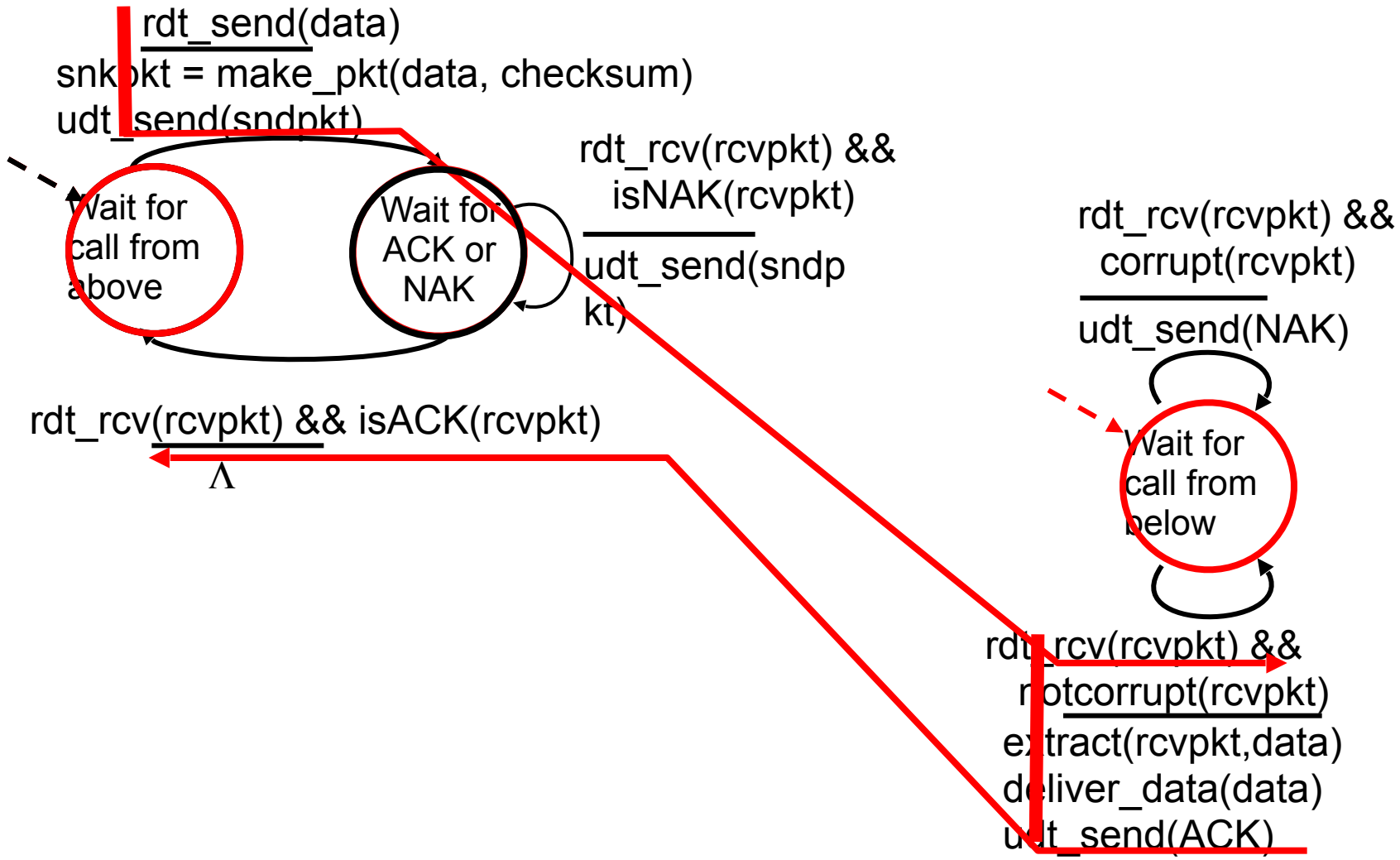


émetteur

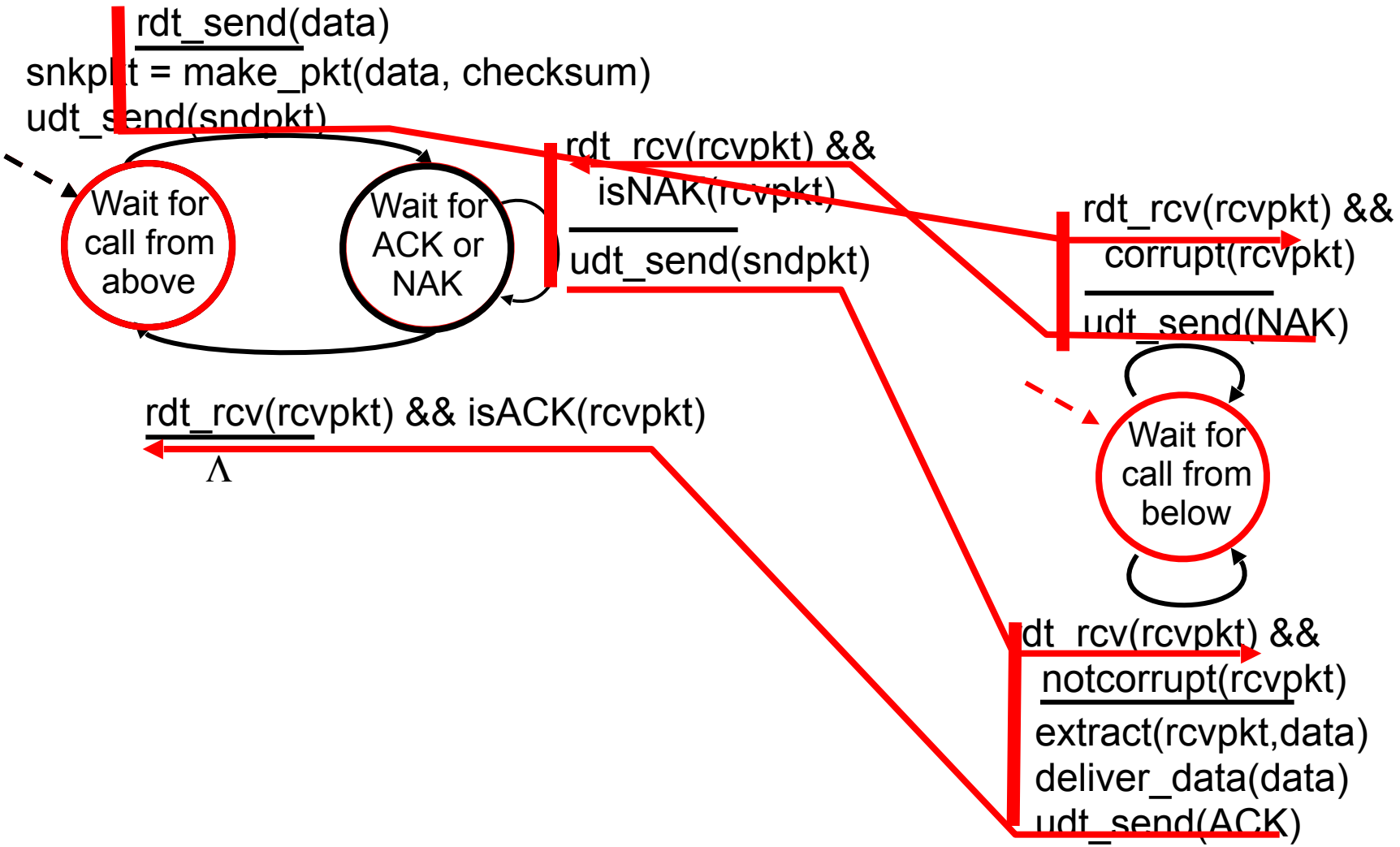
récepteur



rdt2.0: scénario sans erreur de transmission



rdt2.0: scénario avec erreur



rdt2.0 bug !

Erreur lors de la transmission d'un ACK/ NAK ?

- émetteur ne connaît pas ce qui s'est passé coté récepteur
- retransmission du paquet : mauvaise idée, dupliqua possible

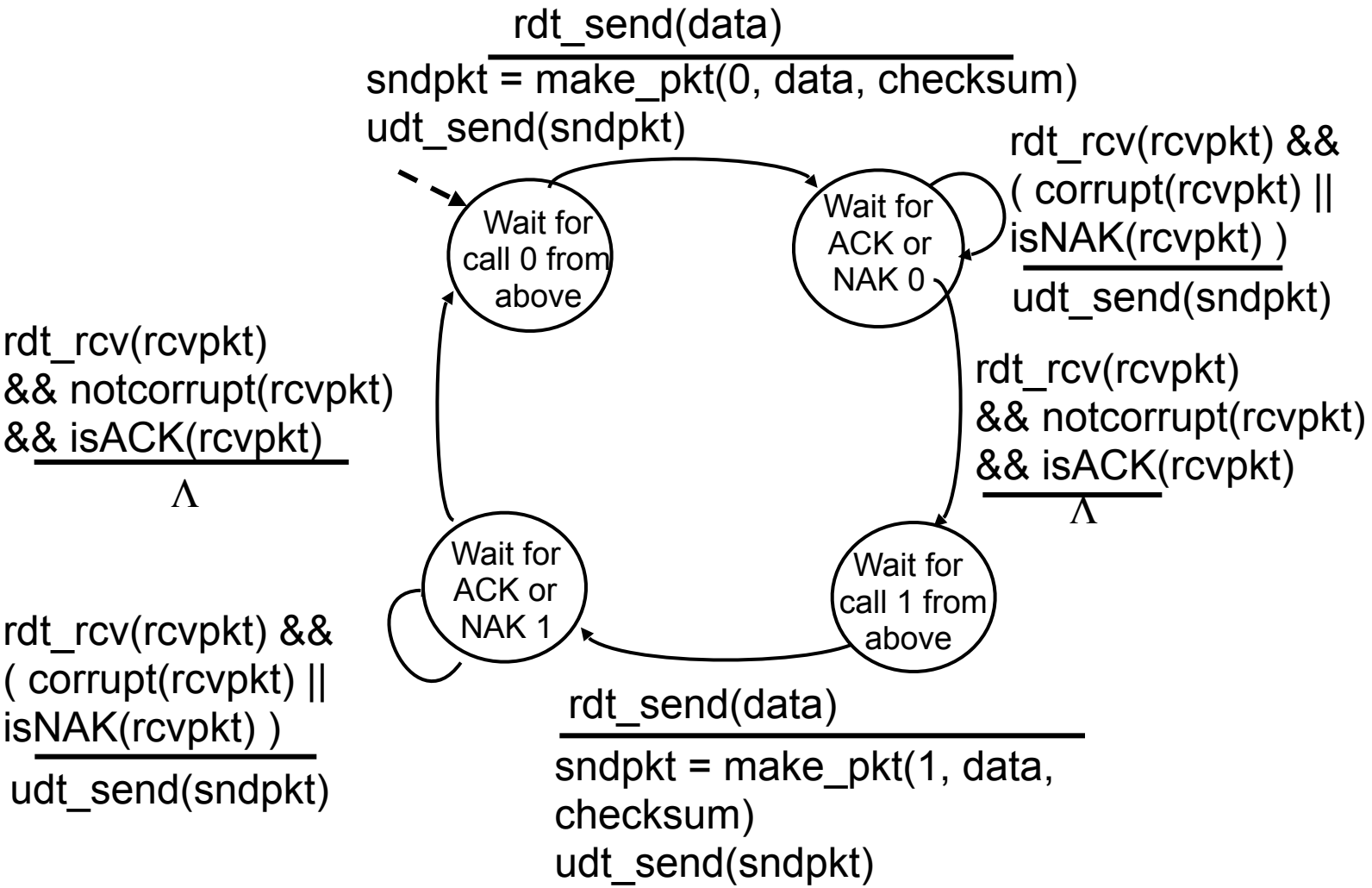
gestion des duplicata:

- retransmission du paquet si ACK/NAK erroné
- chaque paquet a un *numéro de séquence attribué par l'émetteur*
- récepteur ne livre pas les paquets dupliqués

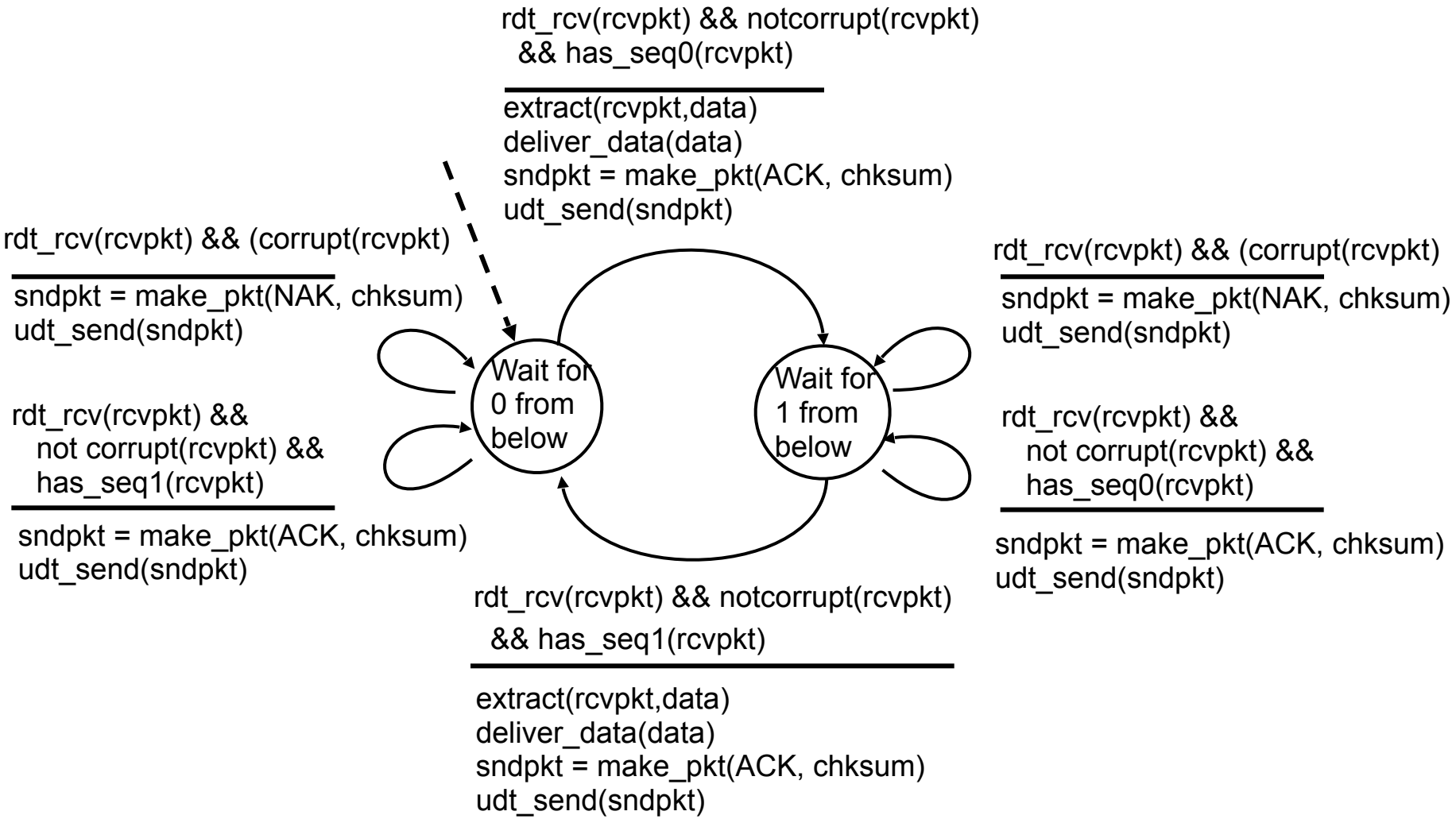
arrêt et attente

émetteur envoie un paquet, puis attend son acquittement.

rdt2.1: émetteur, gestion ACK/NAKs avec erreur



rdt2.1: récepteur, gestion ACK/NAKs avec erreur



rdt2.1: discussion

émetteur:

- paquet avec seq. #
- deux seq. # (0 et 1) suffisent. Pourquoi ?
- verifier si ACK/NAK erroné
- deux fois plus d'états
  - il faut se rappeler si le paquet attendu a le seq# 0 ou 1

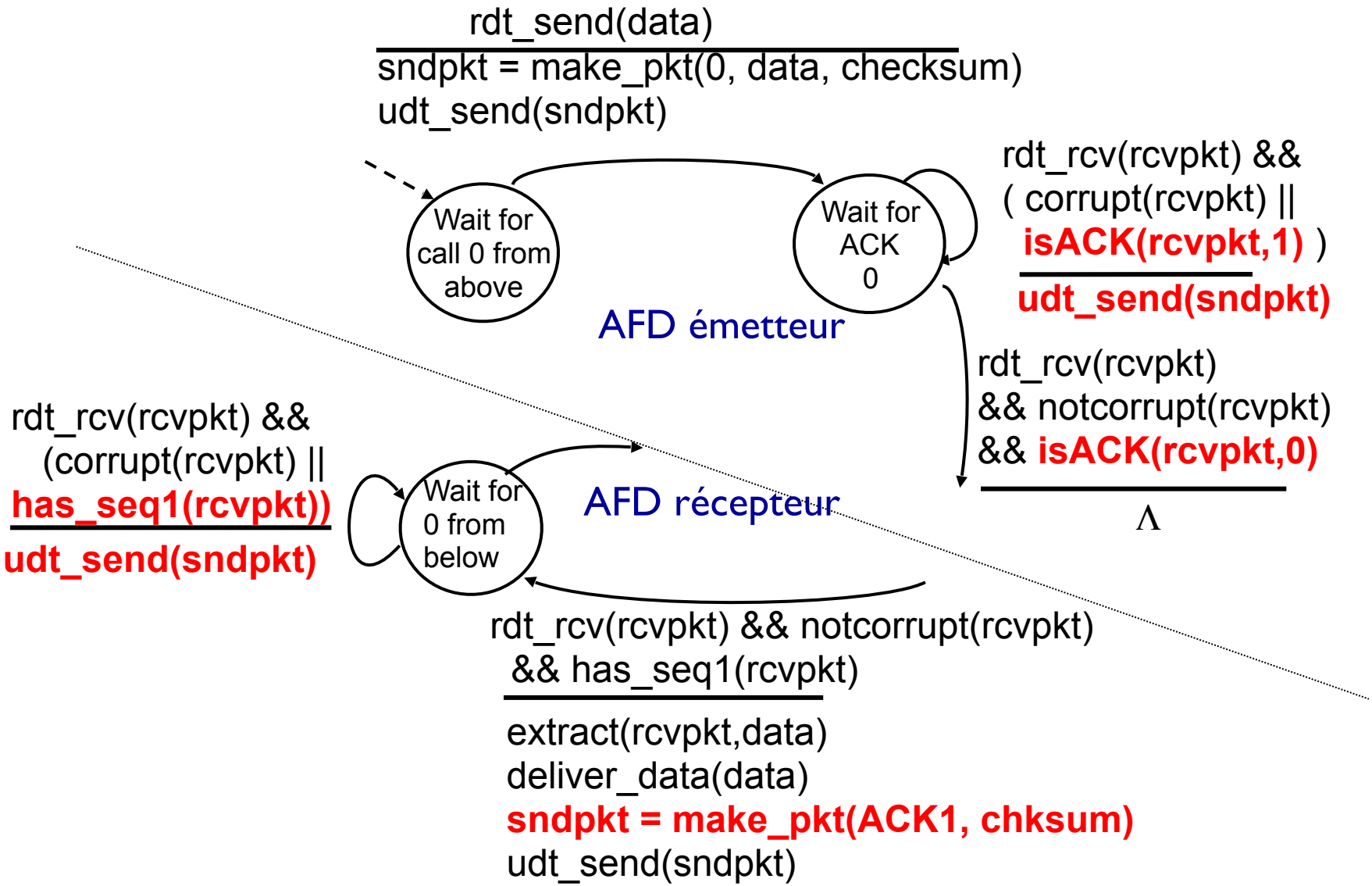
récepteur:

- verifier si un paquet reçu est un dupliqua
  - état courant indique si le seq# attendu est 0 ou 1
- note: récepteur *ne peut pas* savoir si il'y a eu erreur lors de la transmission de son précédent ACK/NAK

rdt2.2: suppression des NAKs

- Même fonctionnalités que rdt2.1
- *N'utilise que des ACKs*
- A la place des NAKs, envoi de ACK pour le dernier paquet sans erreur
  - ACK doit donc inclure le seq# du paquet qu'il acquitte
- Réception ACK dupliqué coté émetteur remplace la réception d'un NAK : l'émetteur retransmet le paquet courant

rdt2.2: émetteur, récepteur (extraits)



rdt3.0: canaux avec erreurs *et* pertes

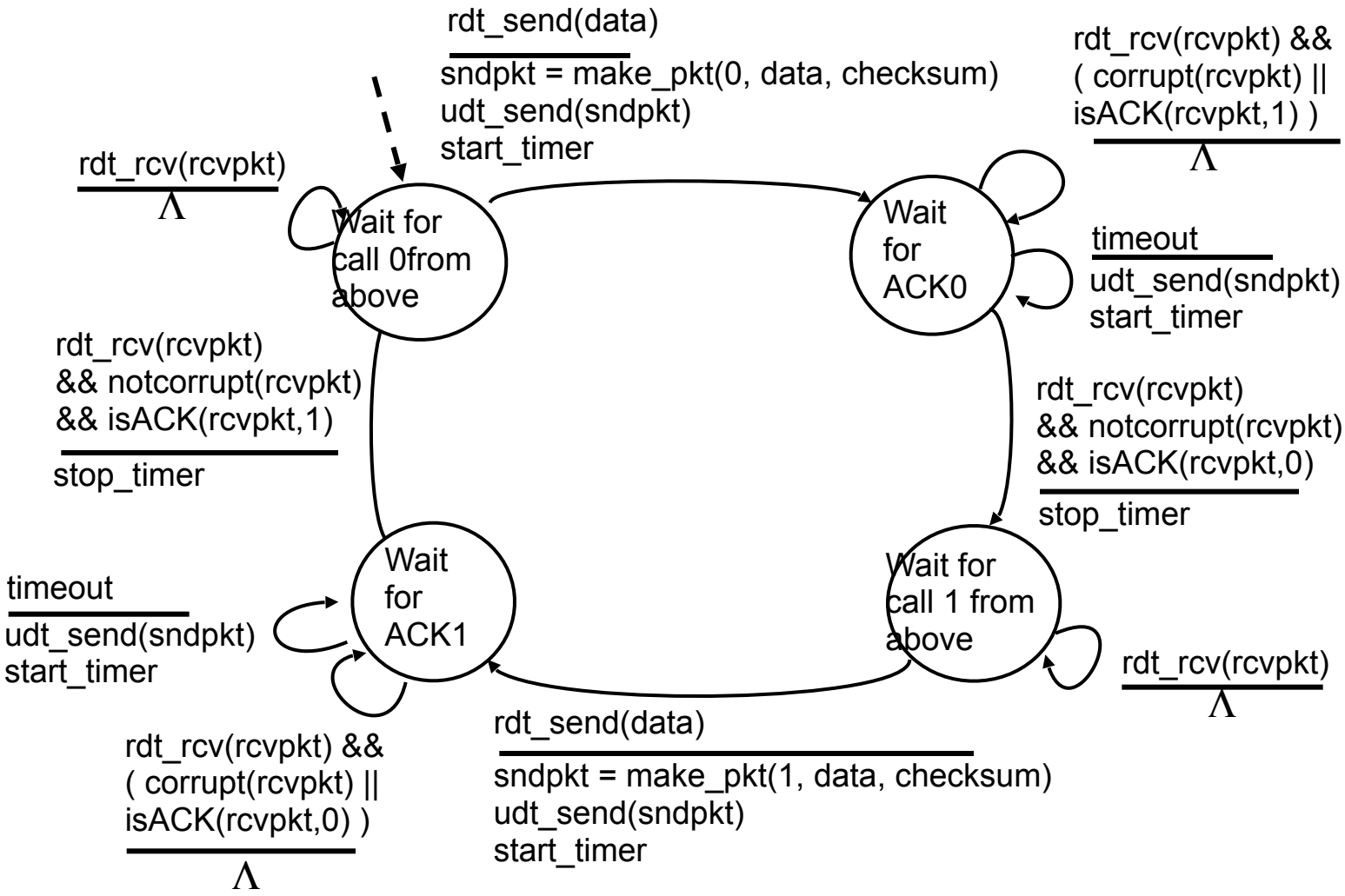
nouvelle hypothèse: canal sous-jacent peut aussi perdre des paquets (donnée ou ACKs)

- checksum, seq. #, ACKs, retransmissions …. insuffisants

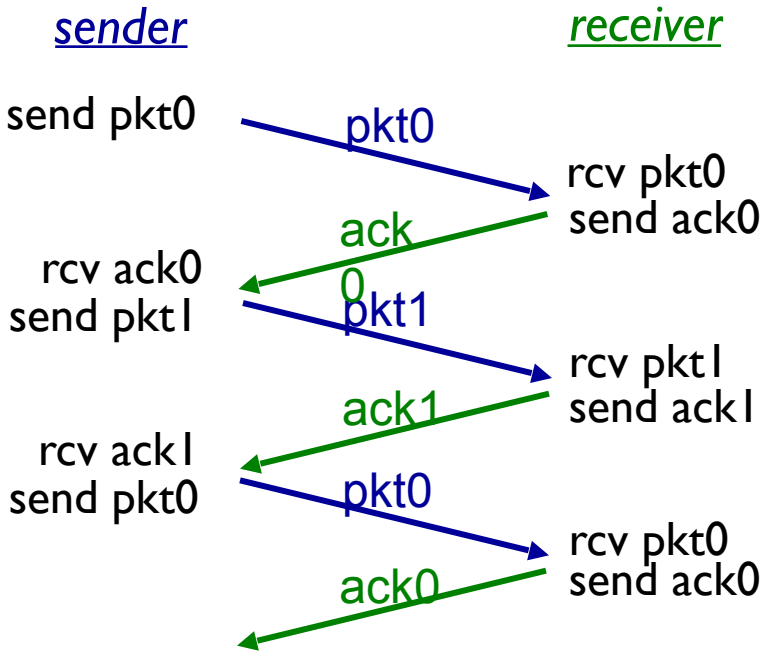
approche: attente ACK pour un « certain temps » avant retransmission

- si ACK perdu → retransmission
- si ACK ou paquet retardé paquet retransmis est un dupliqua, mais ceci est géré par les seq.#
  - ACK doit continuer seq# du packet acquitté
- nécessite un temporisateur (timer)

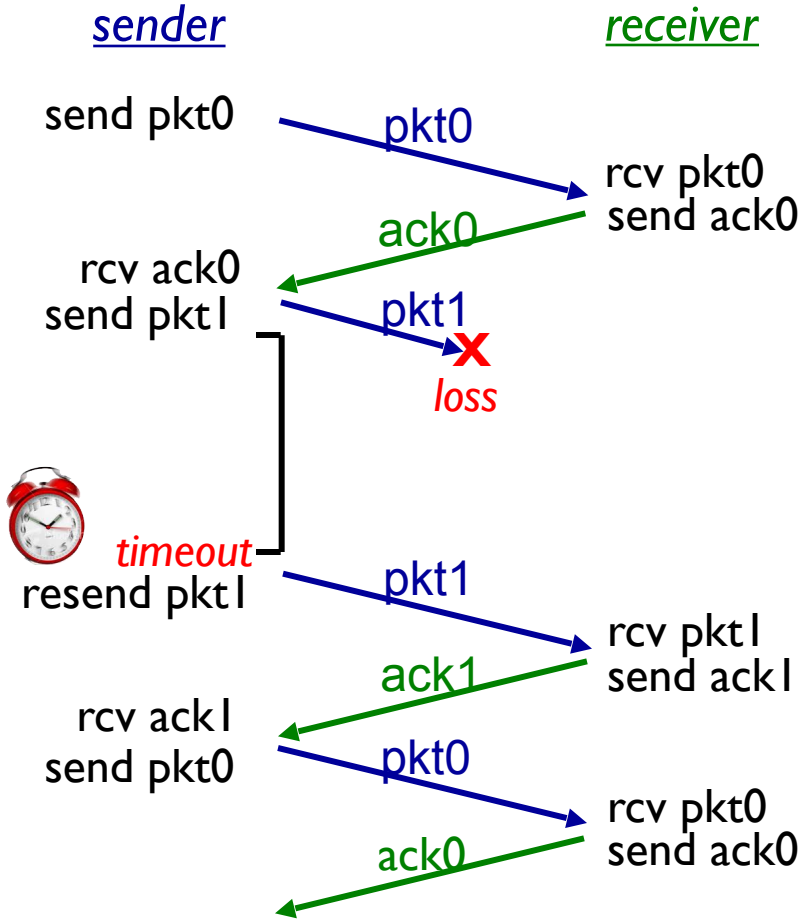
rdt3.0 émetteur



rdt3.0 exemples

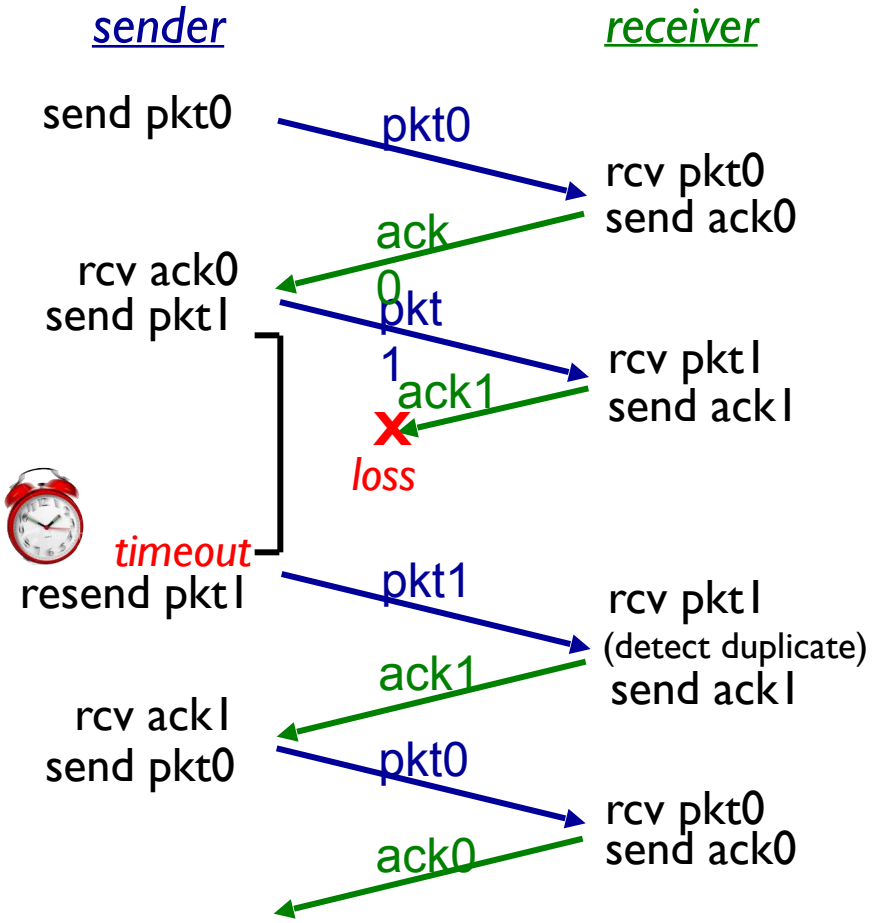


(a) pas de perte

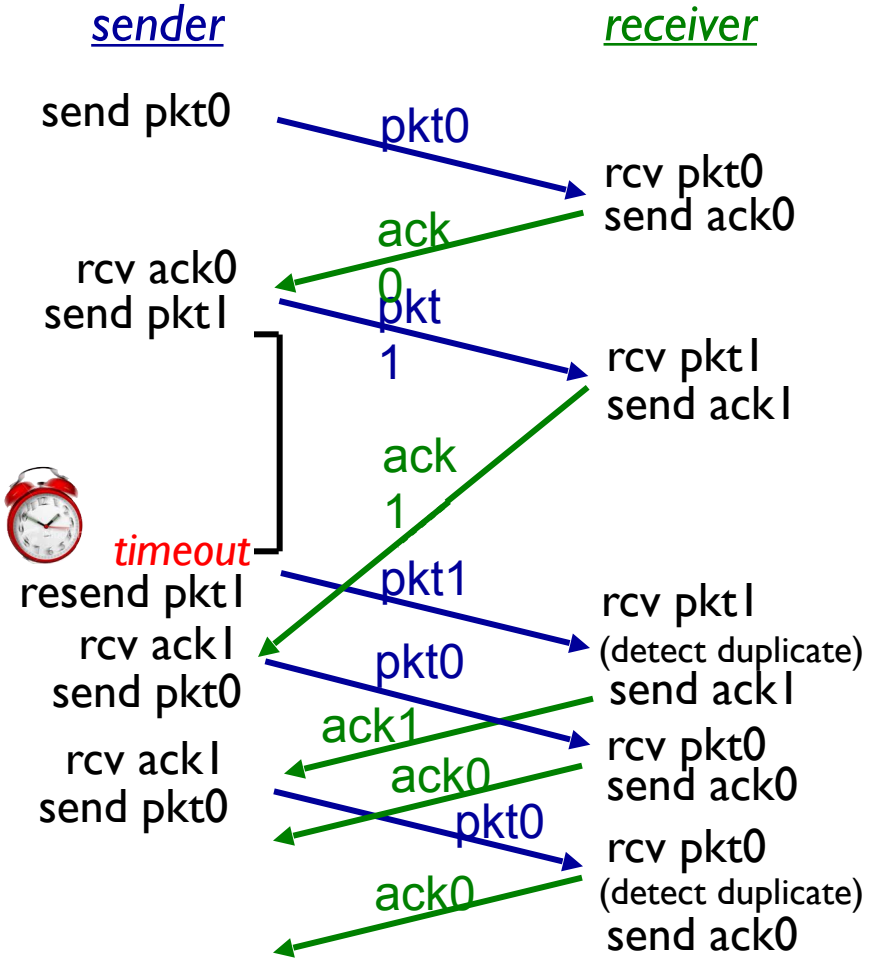


(b) perte d'un paquet

rdt3.0 exemples



(c) perte d'un ACK



(d) temps attente insuffisant/ ACK retardé

rdt3.0 : performances

- rdt3.0 est correct, mais ses performances sont mauvaises
- ex: lien1 Gb/s, latence 15 ms, 8000 bits/packet:

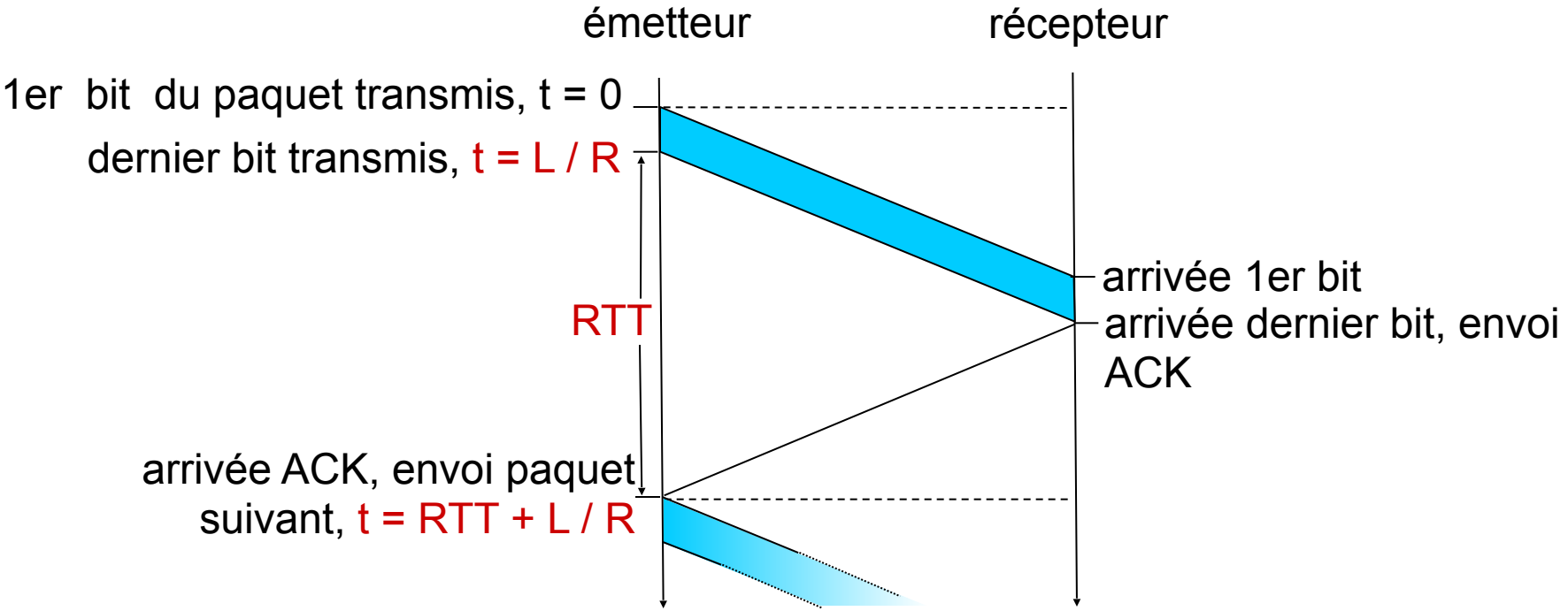
$$D_{trans} = \frac{L}{R} = \frac{8000\ bits}{10^9\ bits/sec} = 8\ microsecs$$

  - U_{sender}: *utilisation* – fraction du temps passé à envoyer

$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

    - avec RTT=30 msec, paquer 1KB toutes les 30 msec : débit effectif 33kB/sec, à comparer avec 1 Gbps du lien
- Le protocole limite fortement l'utilisation des resources physiques!

rdt3.0: arrêt-puis-attente

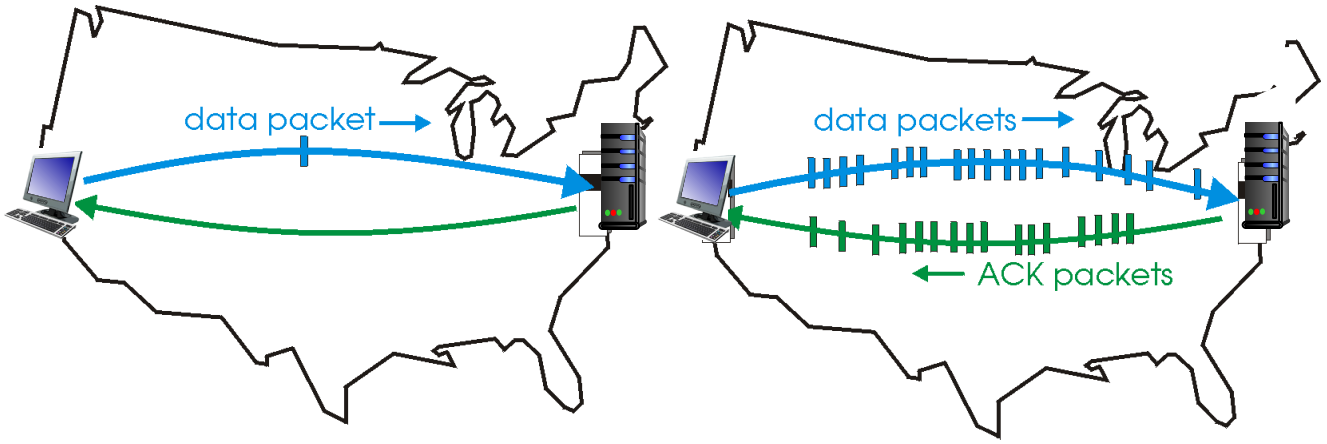


$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

Pipelining

pipelining: plusieurs paquets en transit, accusé de réception non encore reçus

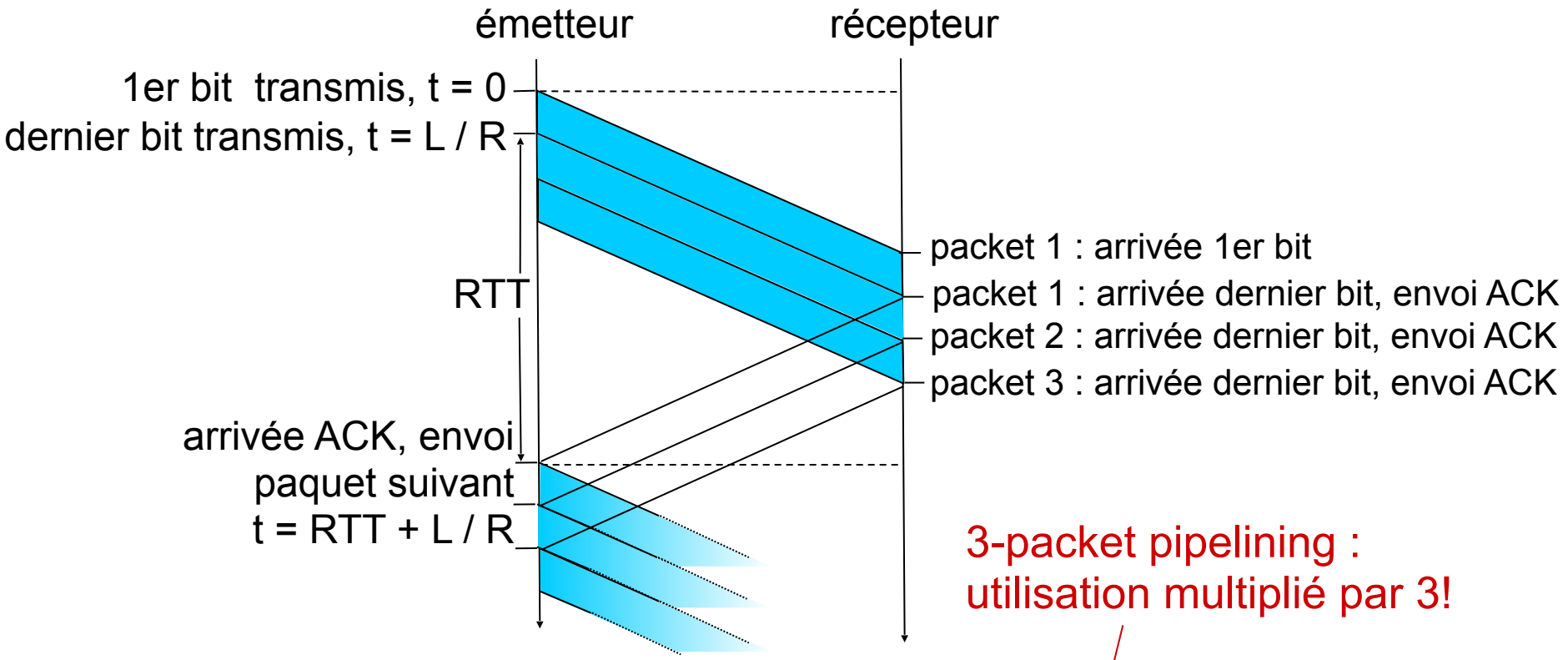
- nécessite un plus grand intervalle de numéros de seq.
- buffers coté émetteur et/ou récepteur



(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

Pipelining: utilisation du réseau



3-packet pipelining :
utilisation multiplié par 3!

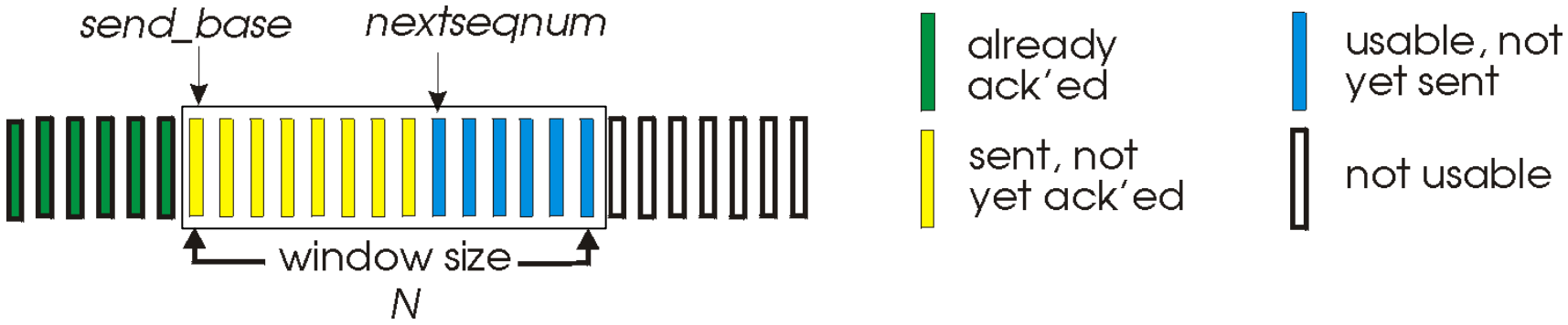
$$U_{sender} = \frac{3L / R}{RTT + L / R} = \frac{.0024}{30.008} = 0.00081$$

Pipelining : aperçu

- Au plus **N** paquets dans le pipeline (ie, au plus N paquets non acquittés émetteur → récepteur)
- Paquets numérotés (numéro de séquences)
- Récepteur envoi des accusés de réception *cumulatifs*
  - *Paquet non acquitté si il y a un « trou » entre les numéros de séquence*
- Temporisateur correspondant *au paquet le plus vieux non acquitté*
  - Quand le temporisateur expire, ***tous les paquets non acquittés*** sont retransmis

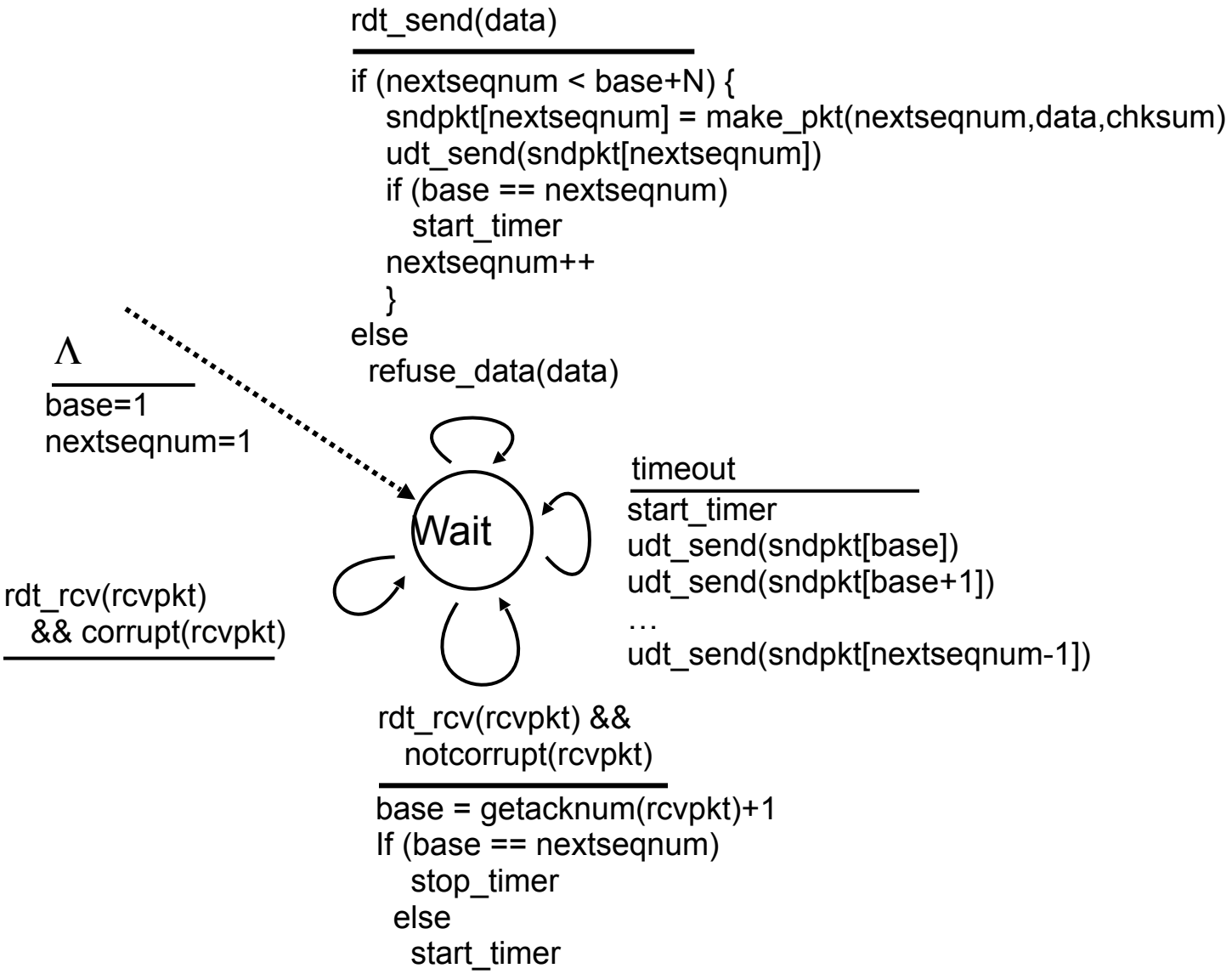
Pipelining : émetteur

- seq # sur *k* bits, dans l'entête du paquet
- « ***fenêtre*** » d'au plus N paquets consécutifs non-acquittés

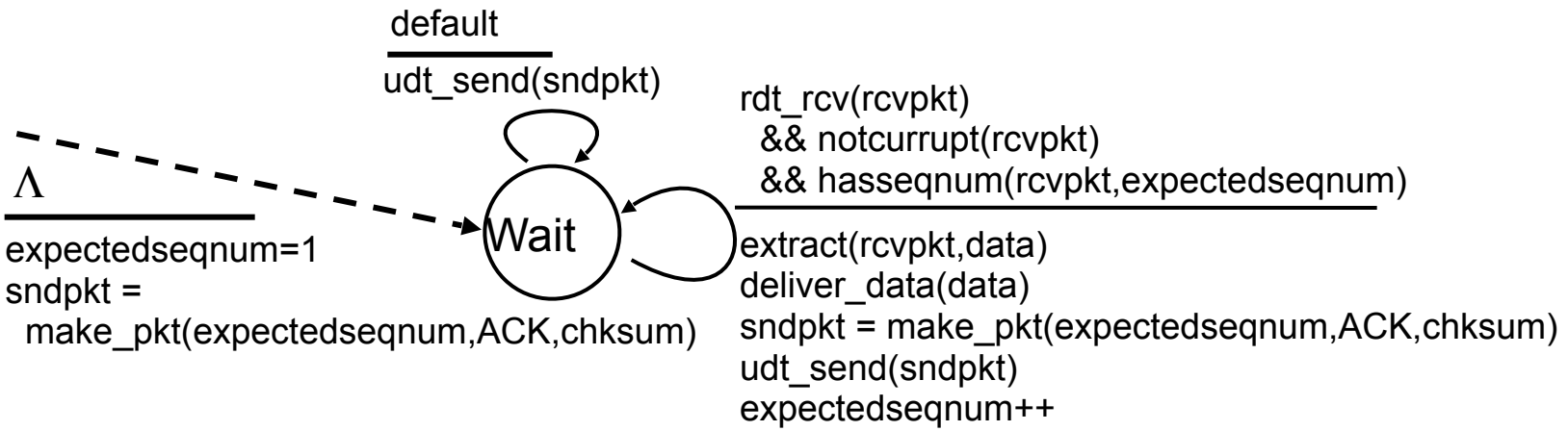


- **ACK(n)**: Accusé de reception de tous les paquets jusqu'au n-ième (inclus) *"ACK cumulatif"*
  - reception possible d'accusés dupliqués
- **temporisateur** pour le paquet en transit le plus vieux
- *timeout(n):* retransmettre paquet #n ainsi que tous les paquets de la fenêtre avec seq# > n

Pipelining : automate émetteur



pipelining: automate récepteur

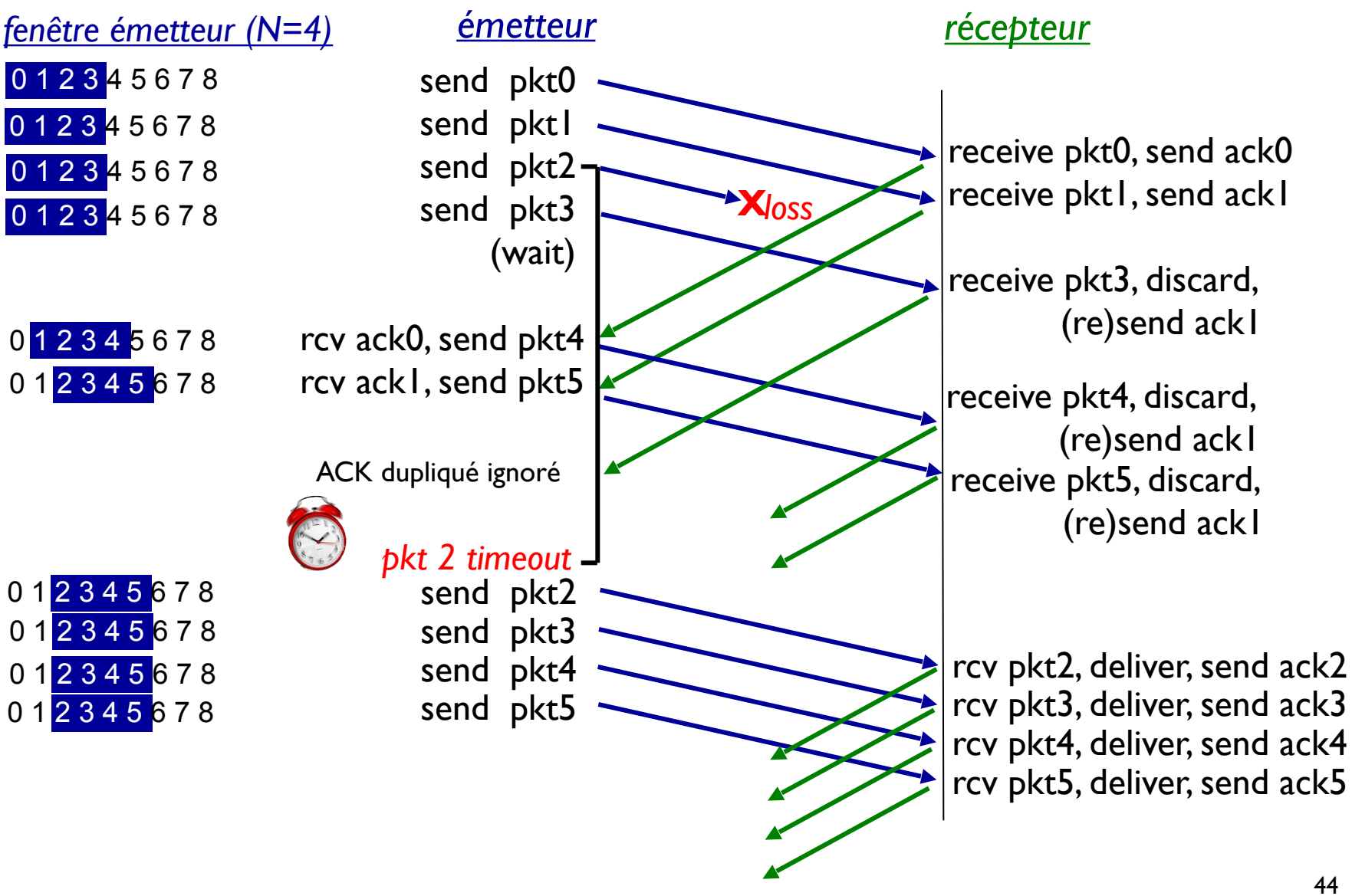


émission ACK: uniquement après reception d'un paquet sans erreur *et dont tous les prédécesseurs ont été reçus*

- géneration possible de ACKs dupliqués
- se rappeler (uniquement) de **`expectedseqnum`**

- paquet en désordre (« trou » dans les seq#):
  - le jeter ! : *pas de buffer en réception!*
  - remission d'un ACK avec le seq# du paquet le plus récent reçu dans l'ordre (cf. exemple)

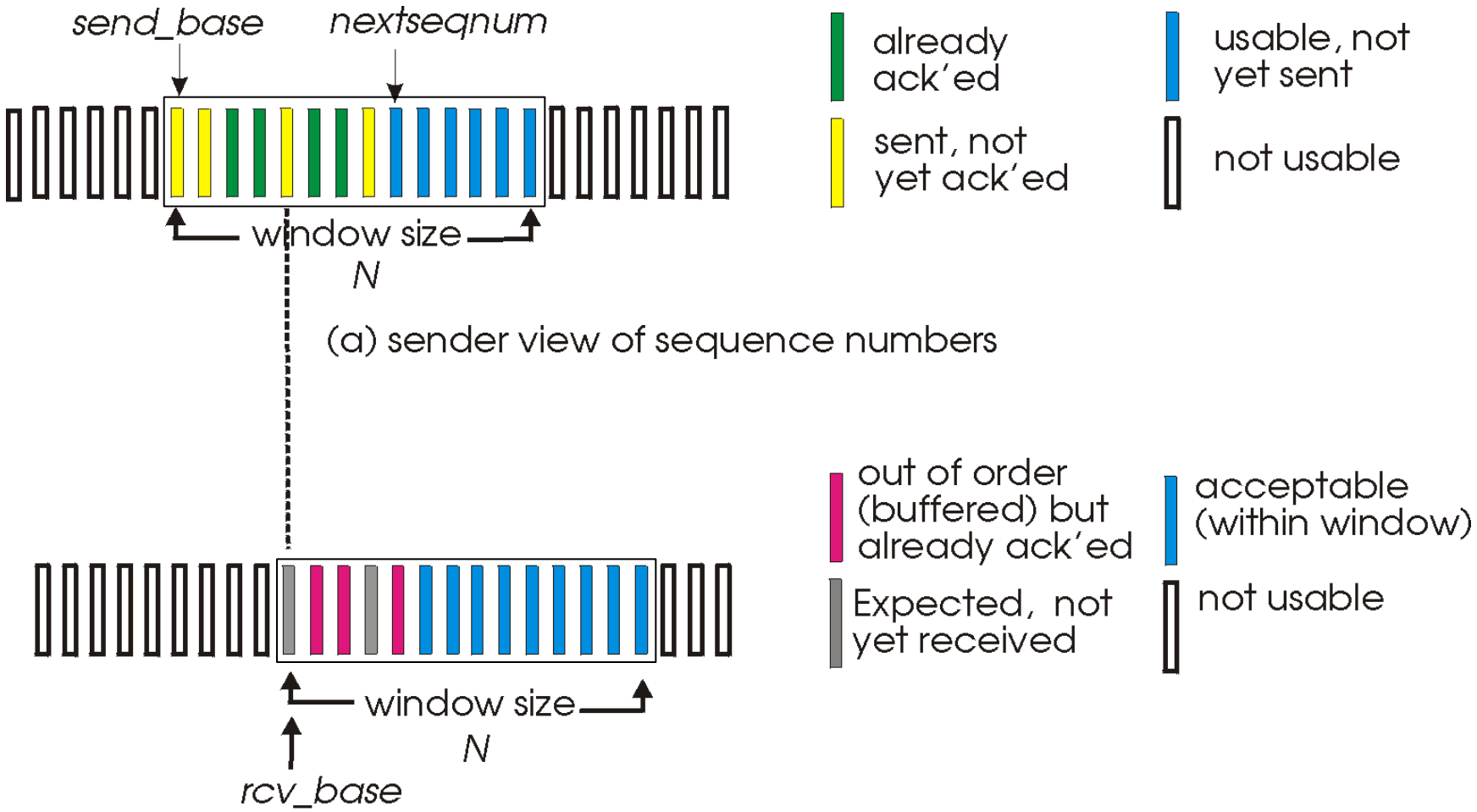
Exemple


fenêtre émetteur (N=4)
émetteur
récepteur
0 1 2 3 4 5 6 7 8
send pkt0
send pkt1
send pkt2
send pkt3
(wait)
receive pkt0, send ack0
receive pkt1, send ack1
loss
receive pkt3, discard,
(re)send ack1
rcv ack0, send pkt4
rcv ack1, send pkt5
receive pkt4, discard,
(re)send ack1
receive pkt5, discard,
(re)send ack1
ACK dupliqué ignoré
pkt 2 timeout
send pkt2
send pkt3
send pkt4
send pkt5
rcv pkt2, deliver, send ack2
rcv pkt3, deliver, send ack3
rcv pkt4, deliver, send ack4
rcv pkt5, deliver, send ack5

Alternative : répétition sélective

- récepteur : accuse réception de chaque paquet reçu sans erreur *individuellement*
  - au besoin, mise en tampon des paquets, pour livraison ultérieure dans l'ordre à la couche supérieure
- émetteur : reémission des paquets pour lesquels ACK non reçus
  - temporisateur pour *chaque paquet* émis
- émetteur : fenêtre
  - *N* numéros de sequence consécutifs
  - Limite le nombre de paquets émis mais non-encore acquittés

Répétition sélective : fenêtre



Répétition sélective

émetteur

rdt_send(data):

- si le seq# suivant dans la fenêtre, créer et émettre un paquet

timeout(n):

- réemettre paquet n, redémarrer le timer

ACK(n) in [sendbase,sendbase+N]:

- marqué pkt n comme reçu
- si n = plus petit seq# des paquets non-acquittés, déplacer la fenêtre au prochain non acquitté

récepteur

pkt n in [rcvbase, rcvbase+N-1]

- envoyer ACK(n)
- out-of-order: buffer
- in-order: livrer les paquets non encore livré dans l'ordre, déplacer la fenêtre au prochain seq# non encore reçu

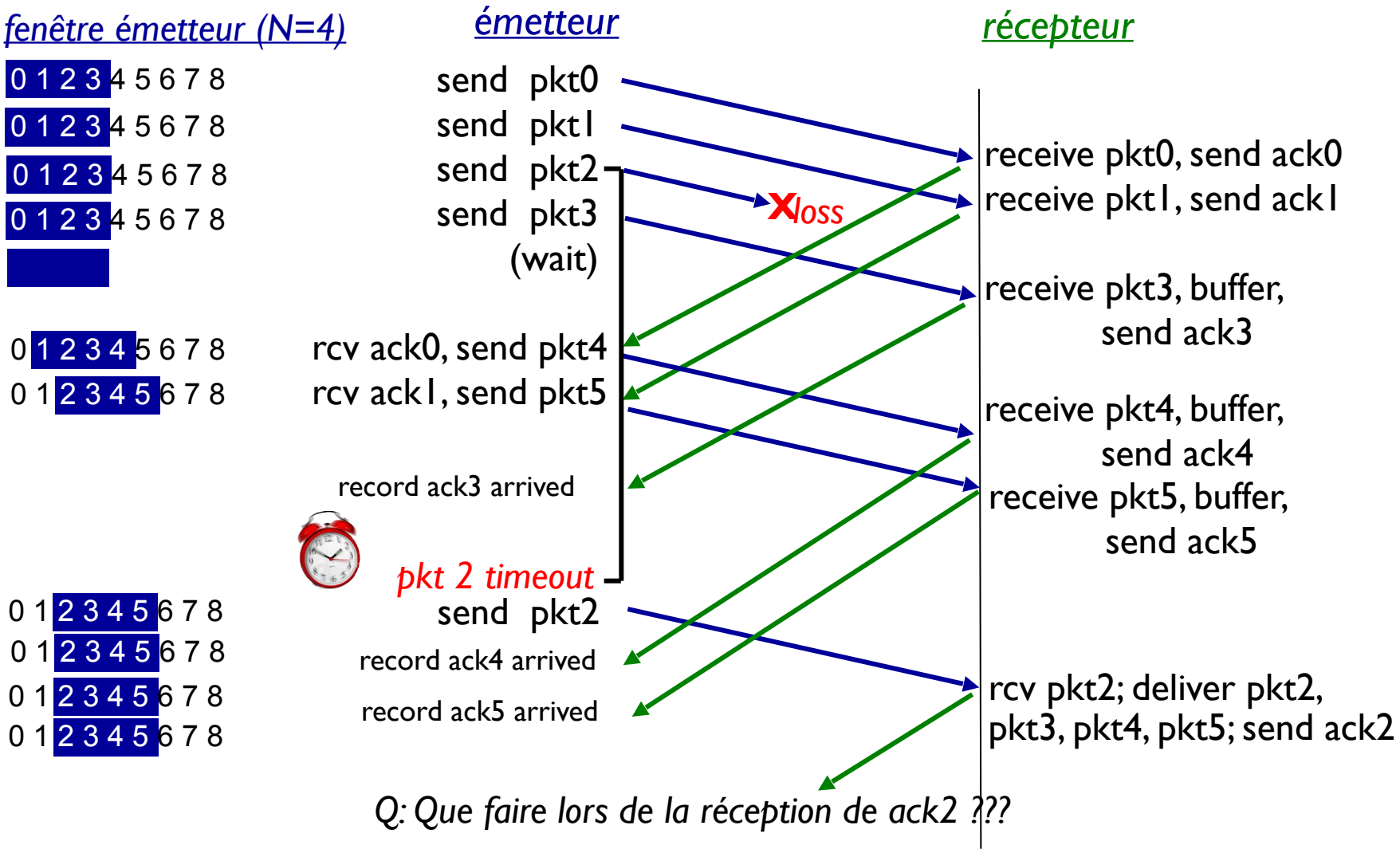
pkt n in [rcvbase-N,rcvbase-1]

- envoyer ACK(n)

défaut:

- ignorer

Répétition sélective : exemple

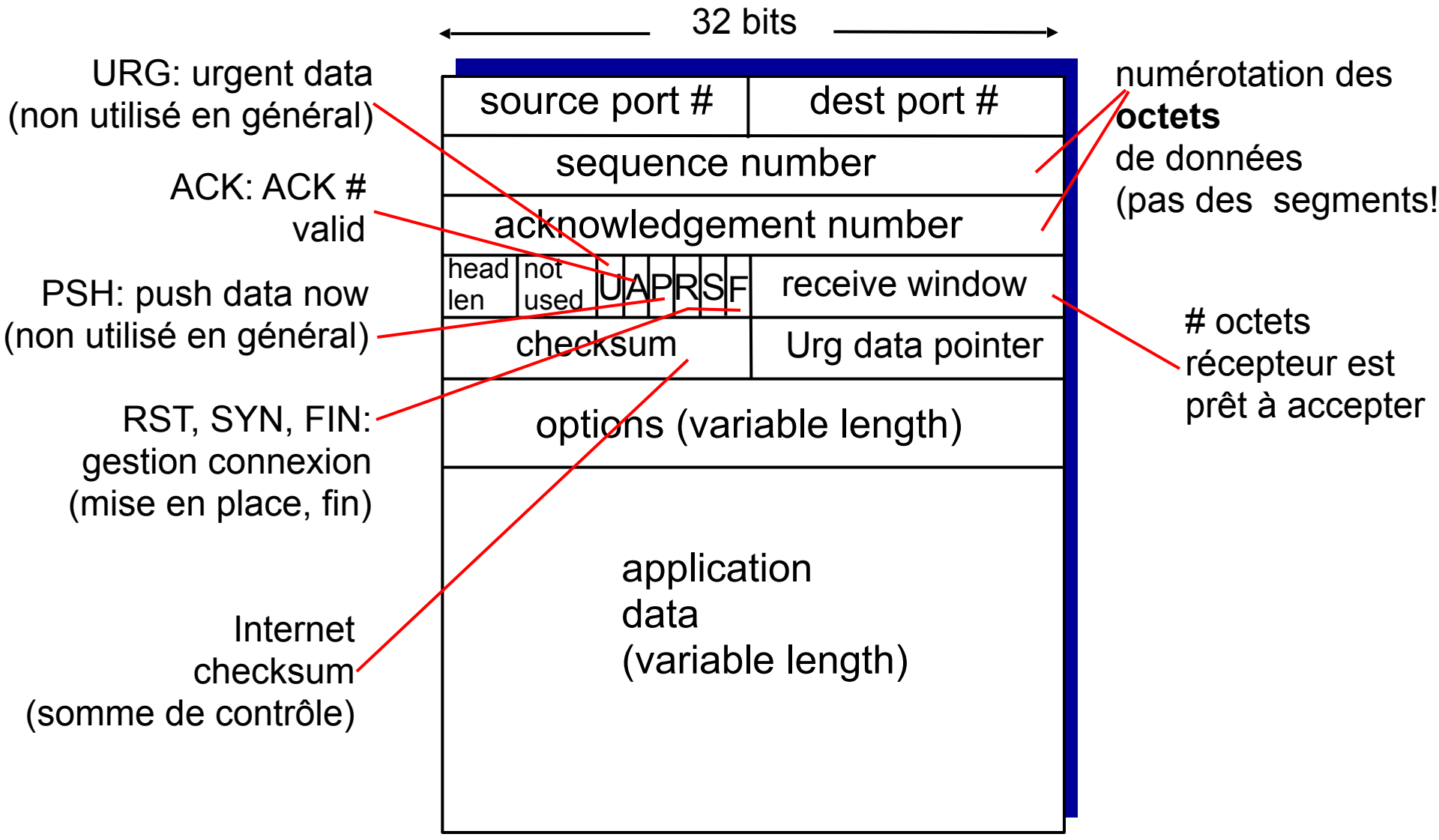


Protocole TCP

TCP: Vue d'ensemble RFCs: 793, 1122, 1323, 2018, 2581

- point-à-point:
  - un émetteur, un récepteur
- flux d'octet, fiable, dans l'ordre
- pipelining:
  - contrôle de congestion et de flot pour ajuster la taille de la fenêtre
- full duplex :
  - flux de données bi-directionel dans une même connection
  - MSS: maximum segment size
- orienté connexion:
  - « handshaking » (échange de messages de contrôle) initialisation de l'émetteur et du récepteur avant transfert des données
- flux controlé:
  - fréquence d'emission compatible avec capacité de traitement du récepteur

segment TCP

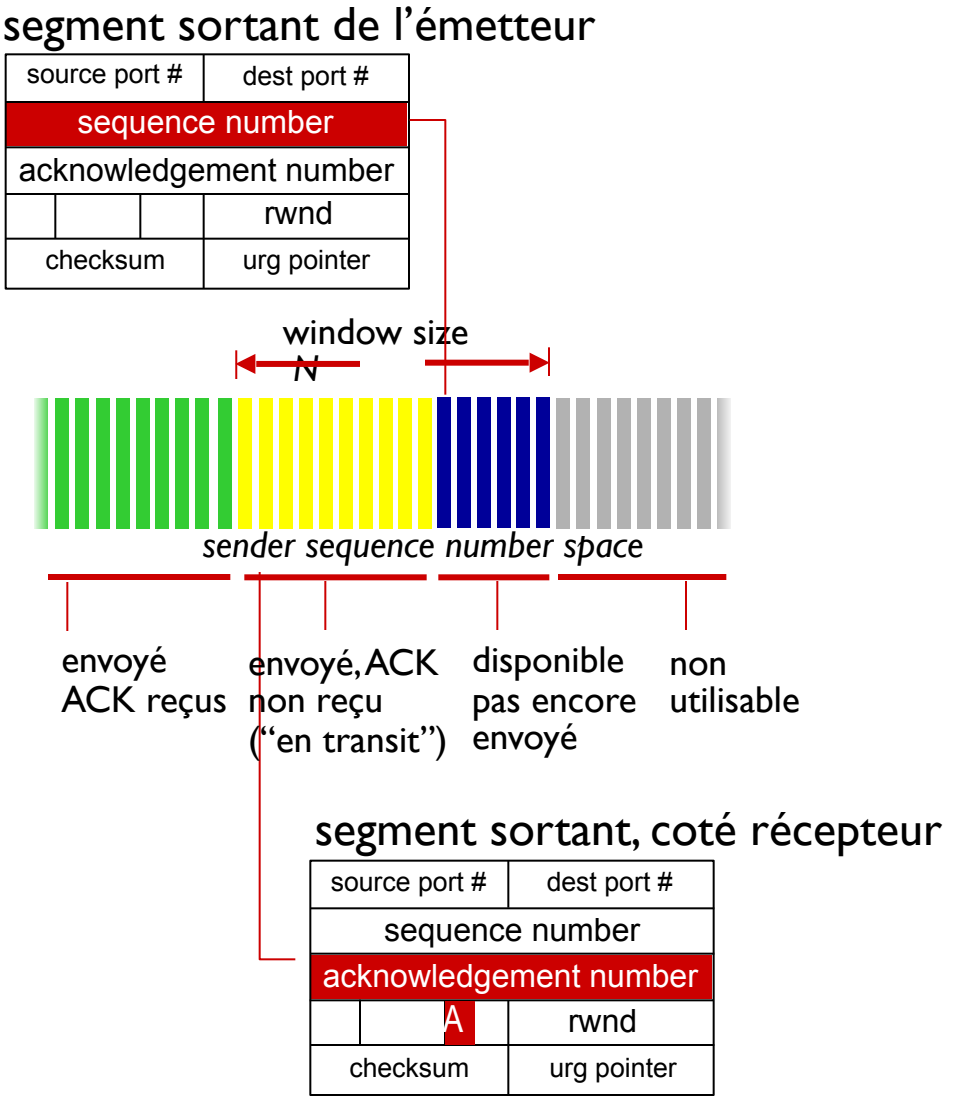

32 bits
URG: urgent data
(non utilisé en général)
ACK: ACK #
valid
PSH: push data now
(non utilisé en général)
RST, SYN, FIN:
gestion connexion
(mise en place, fin)
Internet
checksum
(somme de contrôle)
source port #
dest port #
sequence number
acknowledgement number
head len
not used
U
A
P
R
S
F
receive window
checksum
Urg data pointer
options (variable length)
application
data
(variable length)
numérotation des
octets
de données
(pas des segments!
octets
récepteur est
prêt à accepter

Numéros de séquence, ACKs

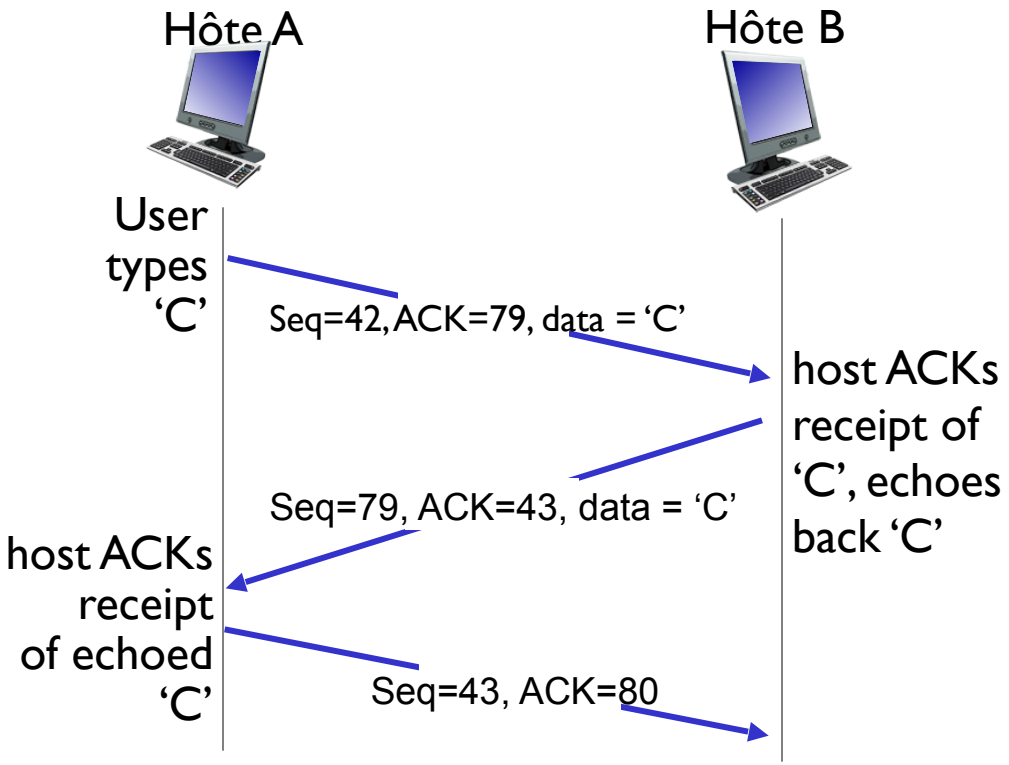
numéro de séquence :
- numéro du premier octet des données du segment

accusé de réception:
- seq# du prochain octet attendu
- accusé cumulatif

Q: Que fait le récepteur des segments reçus dans désordre ?
- R: non spécifié, laissé au choix de l'implémentation



Numéros de séquence, ACKs



Exemple : telnet

Round trip time (RTT), timeout

Q: Comment estimer le temps d'aller-retour (RTT)?

- **`SampleRTT`**: mesure le temps écoulé entre l'émission d'un segment et la réception du ACK correspondant
  - ignore les retransmissions
- **`SampleRTT`** varie, parfois fortement
  - **`EstimatedRTT`** : moyenne de plusieurs mesures récentes
  - éviter les variations trop brusques

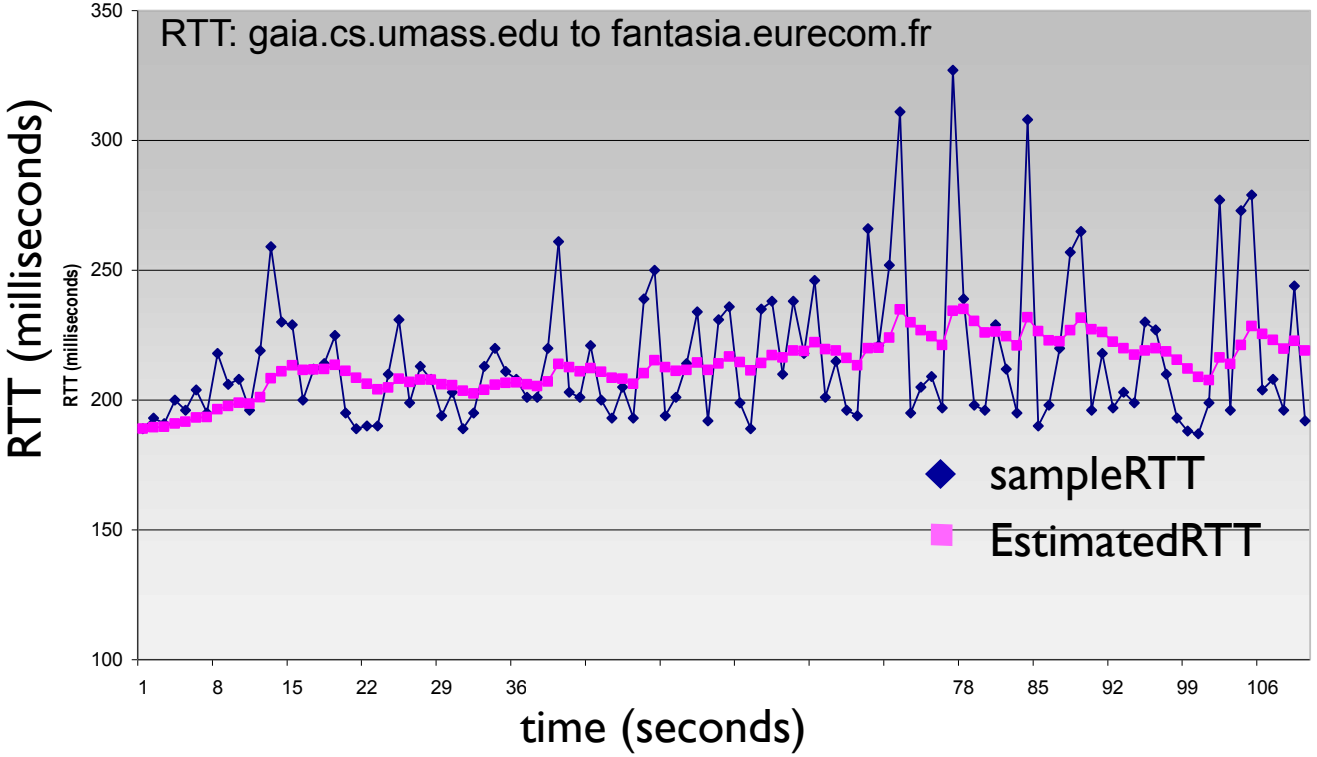
Q: Quel délai d'expiration des temporisateur (timeout) ?

- **`EstimatedRTT`** + marge de sécurité
  - éviter les retransmission inutiles
  - … mais limiter délai avant rémission d'un paquet erroné

Round trip time

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

- « Moyenne mobile exponentielle »
- L'influence des mesures passées décroît exponentiellement
- Typiquement : $\alpha = 0.125$



Round trip time, timeout

- timeout interval:
  - **`EstimatedRTT`** plus « marge de sécurité »
  - marge de sécurité dépend des variations de **`EstimatedRTT`**
- Estimation de la déviation de SampleRTT par rapport à EstimatedRTT:

$$\texttt{DevRTT} = (1-\beta)*\texttt{DevRTT} + \beta*|\texttt{SampleRTT}-\texttt{EstimatedRTT}|$$

(typiquement, $\beta = 0.25$)

$$\texttt{TimeoutInterval} = \texttt{EstimatedRTT} + 4*\texttt{DevRTT}$$

estimated RTT

« marge de sécurité »

TCP : transfert fiable de données

TCP : transfert fiable de données

- TCP implémente un transfert fiable de données au dessus d'IP
  - pipelining
  - accusés de réception cumulatifs
  - temporisateur unique
- Les retransmissions sont déclenchés par :
  - expiration du temporisateur
  - réception d'ACKs dupliqués

Dans un premier temps, version simplifiée de l'émetteur
  - acks dupliqués sont ignorés
  - pas de contrôle de flot, ni de contrôle de congestion

Emetteur : évènements

data reçues de la couche appli :

- creation d'un segment avec seq#
- seq# est le numéro du premier octet de données du segment dans le flux de d'octets
- démarrer le temporisateur si celui-ci est arrêté
  - temporisateur correspond au plus vieux segment non acquitté
  - temps avant expiration: `TimeOutInterval`

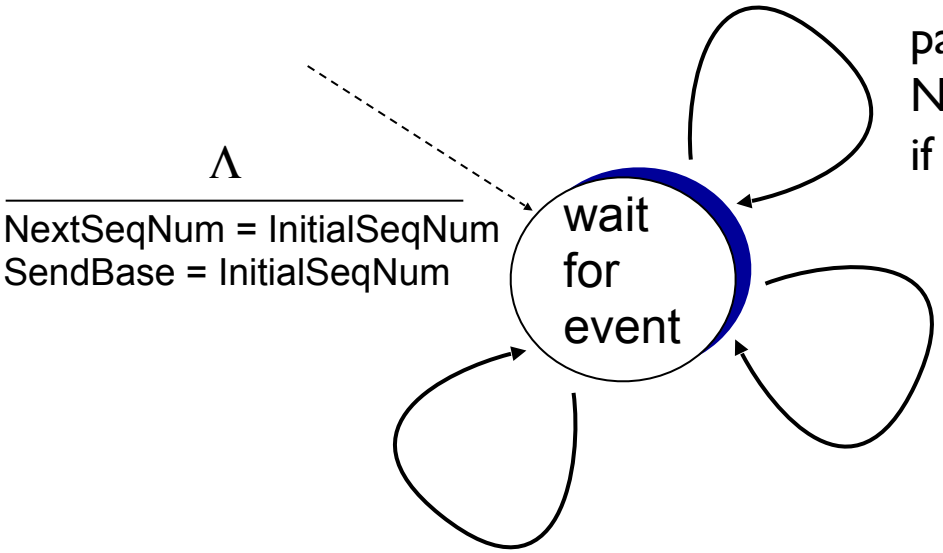
expiration du temporisateur:

- reémission du segment correspondant
- redémarrer le temporisateur

réception ack

- si ack acquitte un segment non encore acquitté
  - mettre à jour connaissance des segments acquittés
  - redémarrer temporisateur s'il existe encore des segments non acquittés

Automate émetteur (simplifié)



data received from application above

create segment, seq. #: NextSeqNum
pass segment to IP (i.e., "send")
NextSeqNum = NextSeqNum + length(data)
if (timer currently not running)
    start timer

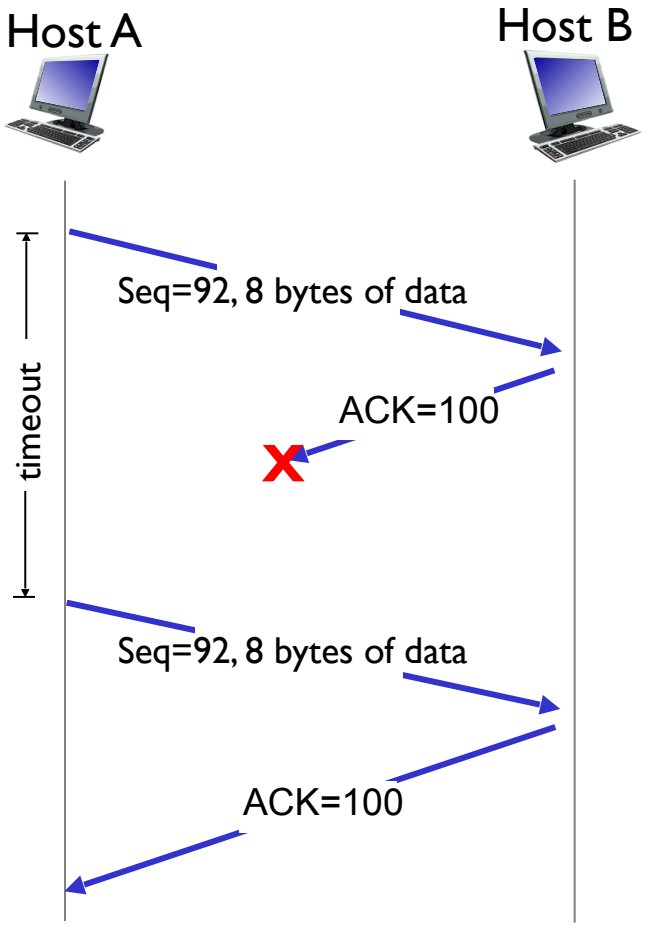
timeout

retransmit not-yet-acked segment
        with smallest seq. #
start timer

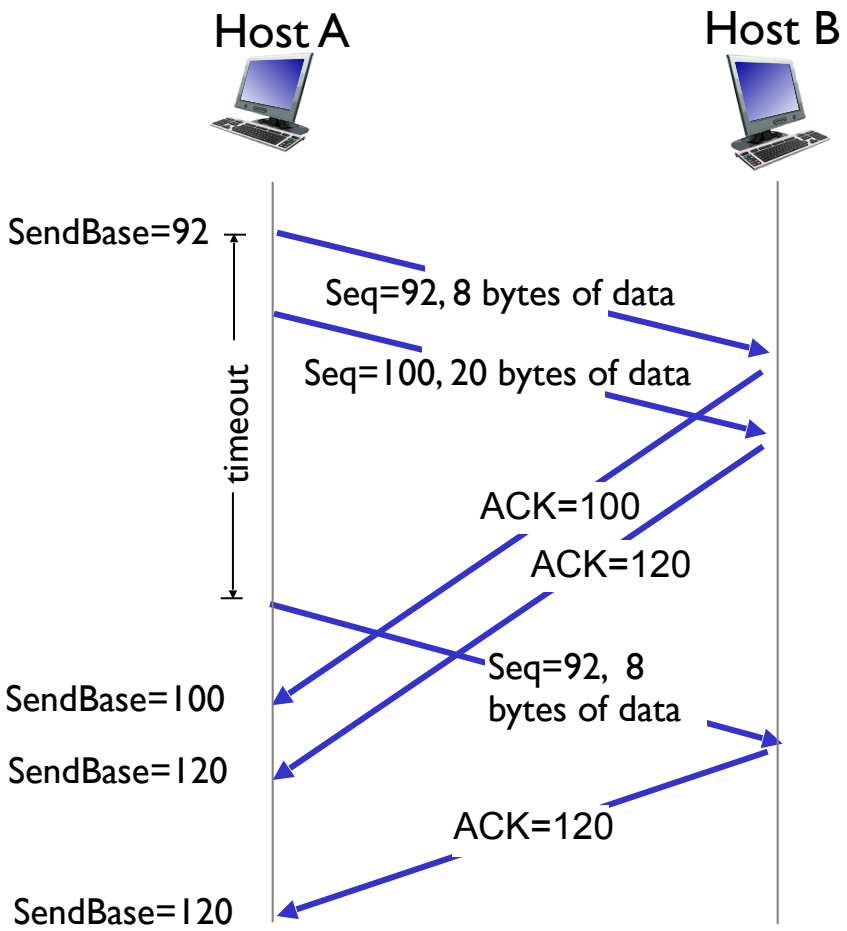
ACK received, with ACK field value y

```
if (y > SendBase) {
    SendBase = y
    /* SendBase–1: last cumulatively ACKed byte */
    if (there are currently not-yet-acked segments)
        start timer
      else stop timer
     }
```

TCP: Exemples de retransmissions

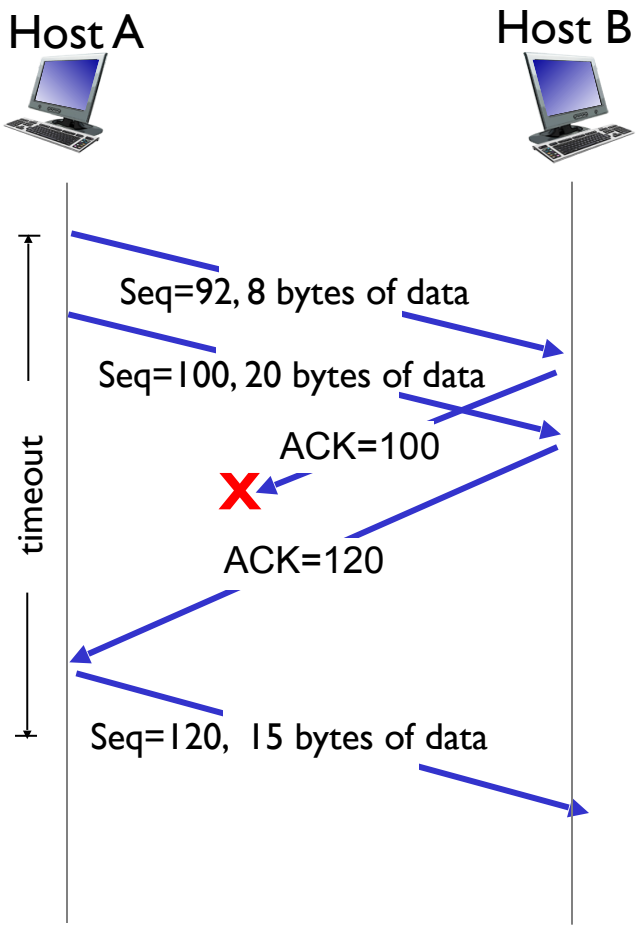


lost ACK scenario



premature timeout

TCP: Exemples de retransmissions



cumulative ACK

Récepteur : envoi de ACK [RFC 1122, RFC 2581]

évènements (récepteur)	*action (récepteur)*
arrivée d'un segment dans l'ordre avec seq# = expected_seq (les octets 0,..,expected_seq ont été reçus)	retardé l'envoi de ACK (500ms max) au cas où un nouveau segment est reçu. Si pas de nouveau segment, envoi ACK
arrivée d'un segment dans l'ordre, un autre segment a un ACK retardé	envoi immédiat d'un ACK cumulatif pour ces deux segments
arrivée d'un segment en désordre (seq# > expected_seq)	envoi immédiat d'un *ACK dupliqué* indiquant seq# attendu
arrivée d'un segment dans l'ordre avec seq# = expected_seq, un ou plusieurs segments avec seq# > expected_seq déjà reçu	envoi immédiat d'un ACK pour le nouveau segment reçu

TCP fast retransmit

- délai temporisateur souvent plutôt long :
  - délai important avant la retransmission d'un paquet perdu
- détection des paquets perdus à partir de ACKs dupliqués
  - émetteur envoie typiquement des segments à la suite les uns des autres
  - si un segment est perdu, plusieurs ACKs dupliqués seront générés

TCP fast retransmit

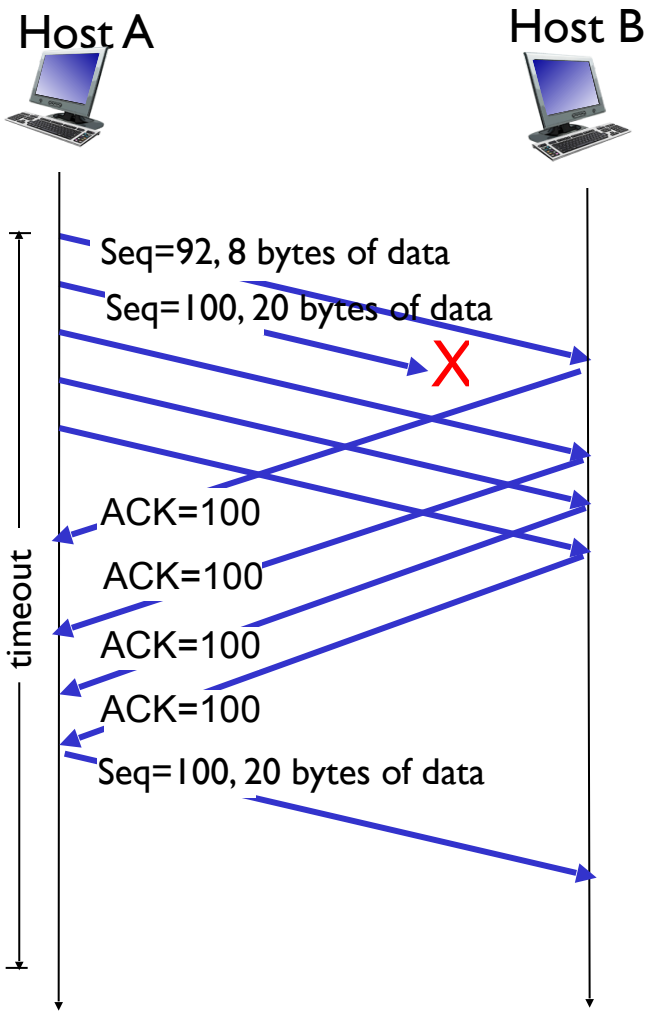
lorsque l'émetteur reçoit 3 ACKs pour le même segment

("triple duplicate ACKs"),

-> retransmission du segment non acquitté avec le plus petit seq#

- Ce segment est probablement perdu, ne pas attendre le temporisateur

TCP fast retransmit



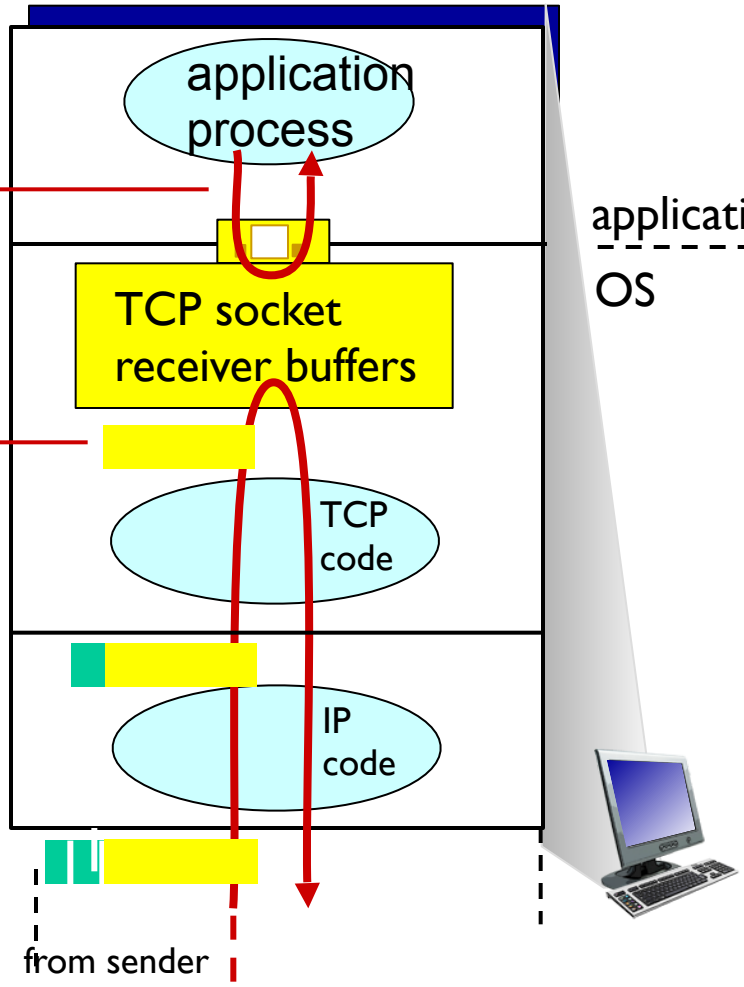
"fast retransmit" après reception d'un ACK en triple exemplaire

TCP : contrôle de flot

TCP contrôle de flot

données retirées des buffers TCP par l'application ….

… plus lentement que leur livraison par le récepteur TCP et/ou leur envoi par l'émetteur TCP



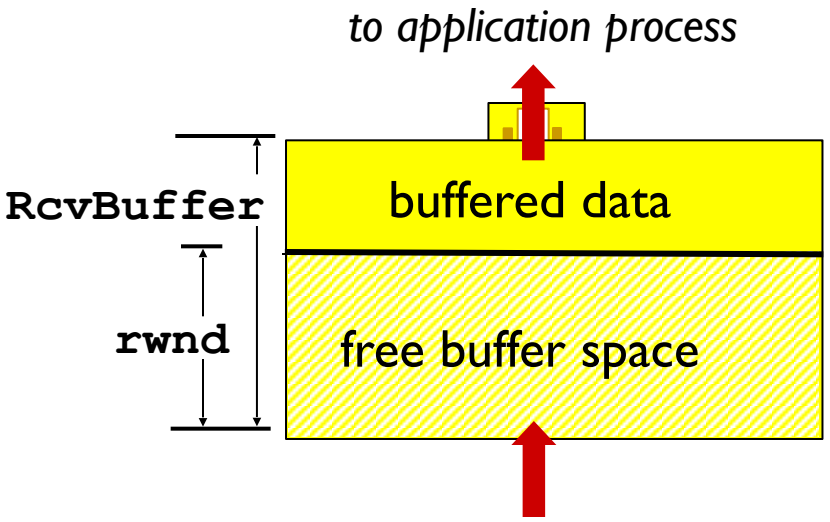
pile réseaux récepteur

contrôle de flot

récepteur contrôle l'émetteur, pour que celui-ci ne sature son buffer en envoyant trop rapidement des segments

TCP contrôle de flot

- récepteur annonce l'espace disponible dans son buffer → valeur **`rwnd`** dans l'entête des segments envoyés à l'émetteur
  - taille du buffer **`RcvBuffer`** (defaut 4096 octets) option des sockets
  - ajusté aussi par l'OS
- émetteur limite le nombre de segment en transit (non-acquitté) à la dernière valeur de **`rwnd`**
- évite saturation (overflow) du buffer du récepteur



mise en buffer coté récepteur

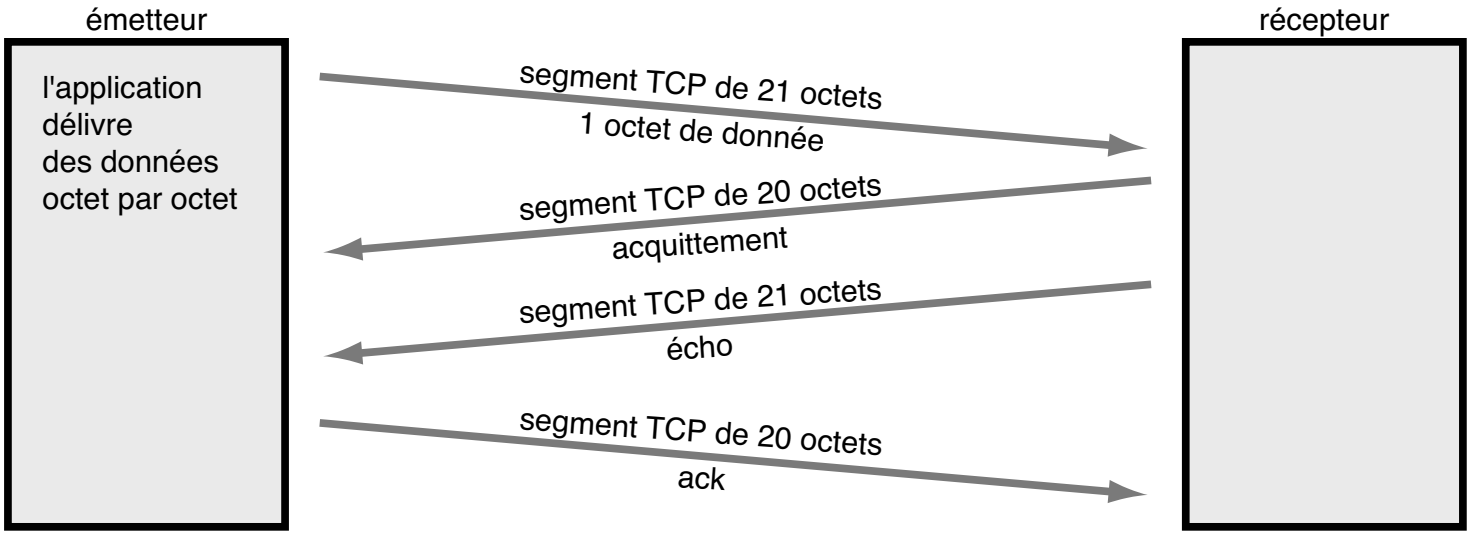
Politique de transmission

- En cas de fenêtre de réception taille nulle, envoi possible de segments pour :
  - stopper une app. distante (donnée URGENTE)
  - demander au récepteur seq# du prochain octet attendu/taille de la fenêtre

- Politique d'envoi ACK/données libre :
  - émetteur *n'est pas tenu* d'envoyer immédiatement les données de l'app.
  - récepteur *n'est pas tenu* d'acquitter au plus vite

Politique de transmission

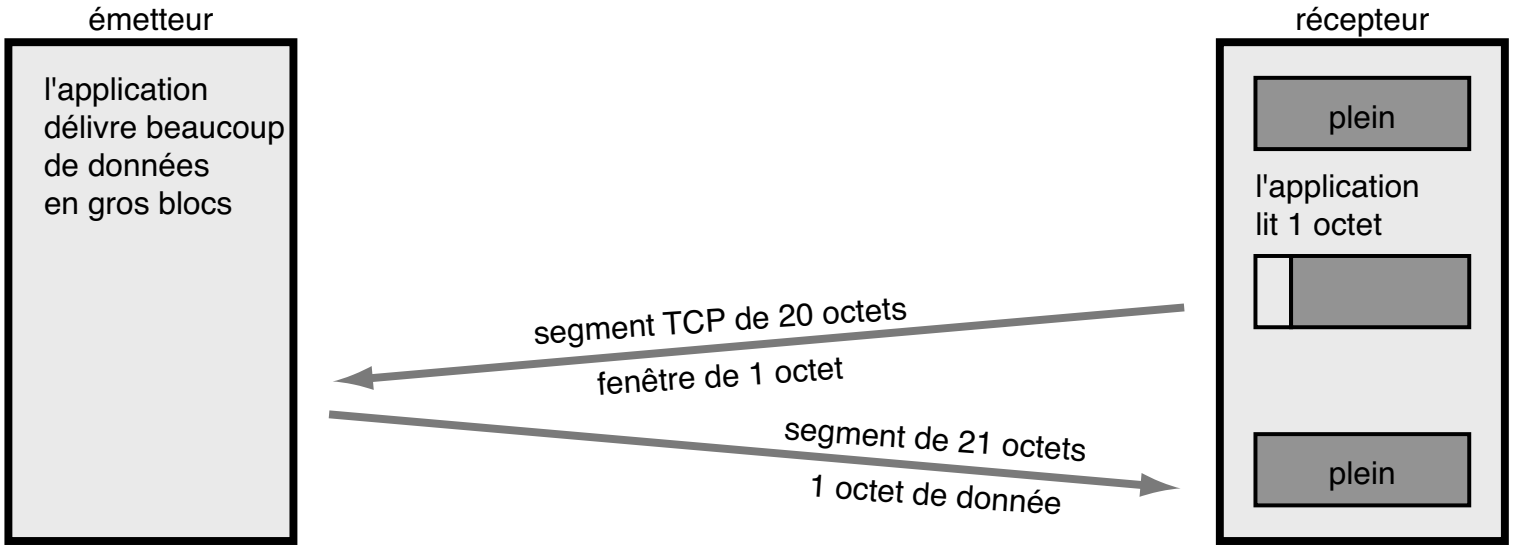
- App. émet les données octet par octet (ex: telnet)



- Algorithm de **Naggle**
  - envoi premier octet
  - stockage des suivants dans un buffer
  - envoi du buffer :
    - lorsqu'il est plein **ou** après réception du ACK du précédent segment

Politique de transmission

- Silly window syndrome



- Solution de **Clark**
  - Le récepteur ne doit pas annoncer une nouvelle taille de fenêtre tant que

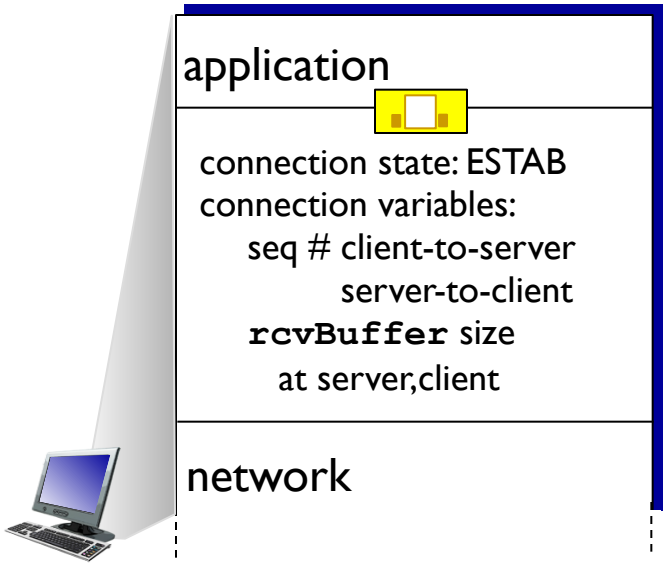
    taille fenêtre < min(MSS, (taille buffer)/2)

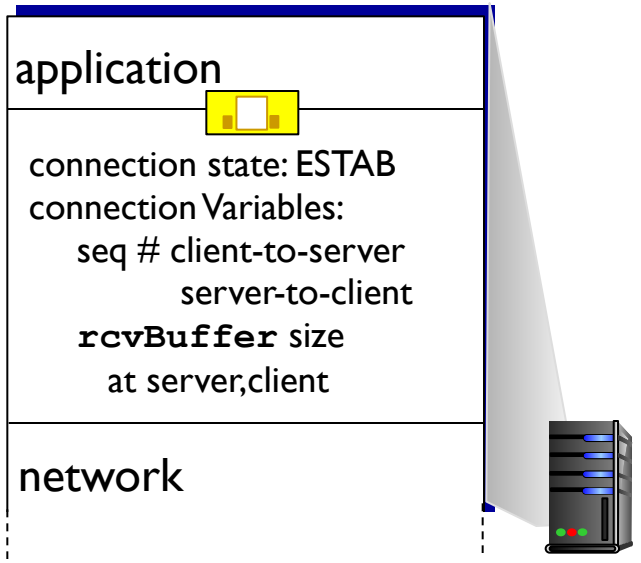
TCP : établissement/fermeture de connexions

Etablissement d'une connection TCP

avant d'échanger des données, « handshake » entre émetteur et récepteur:

- s'accordent pour établir une connection
- s'accordent sur les paramètres (taille des buffers, seq# initial, etc.)



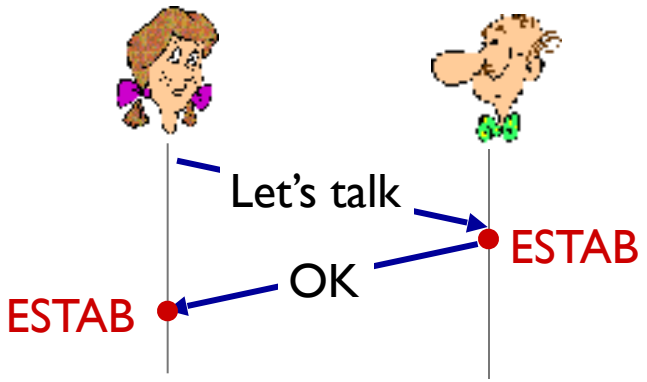


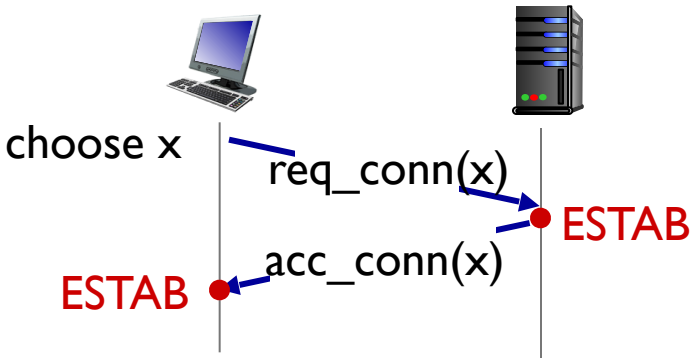

```
Socket clientSocket =
  newSocket("hostname","port
   number");
```

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

Etablissement d'une connection

2-way handshake:

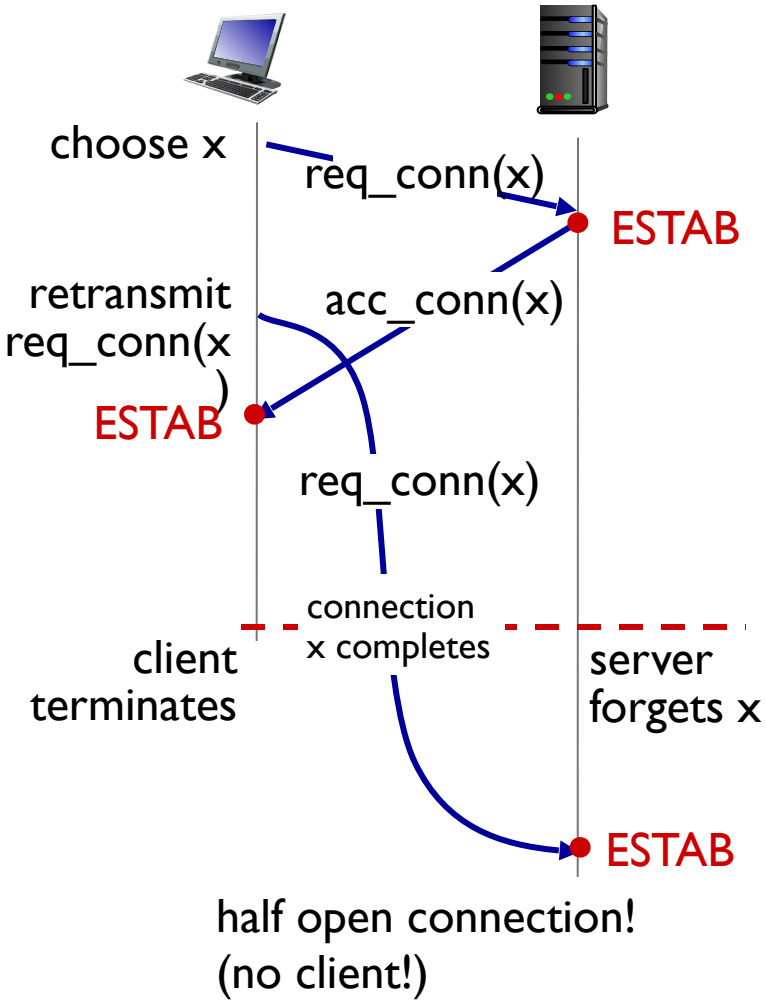


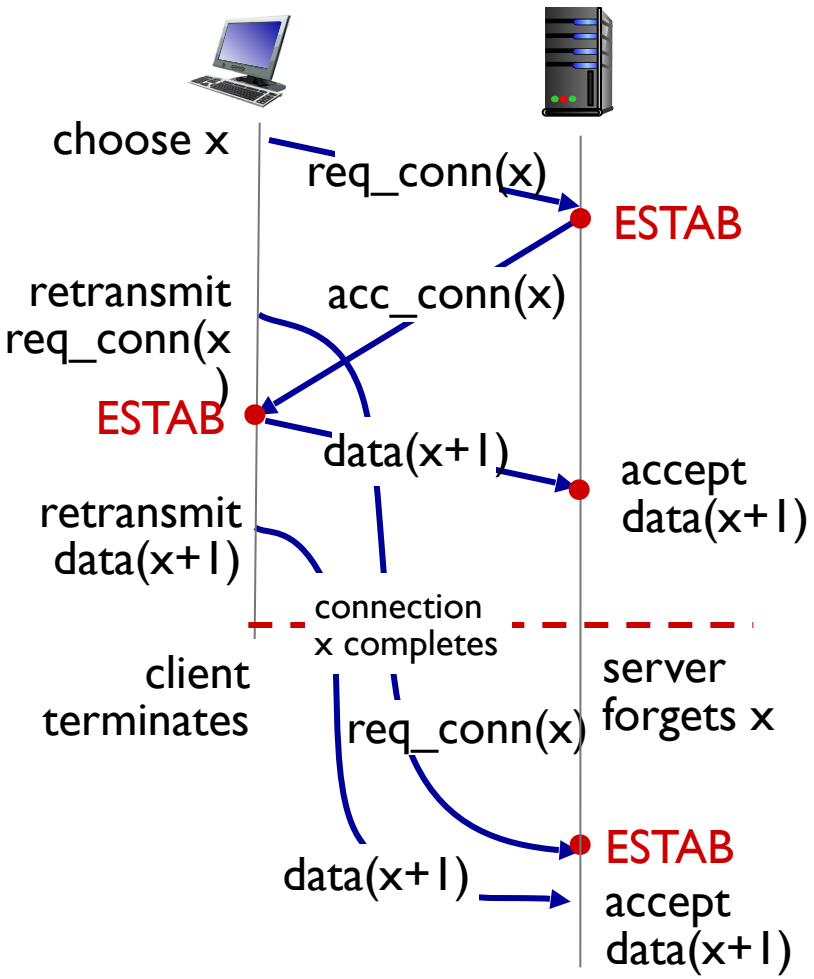


Q: Est-ce que le protocole « 2-way handshake » est correct sur un canal non fiable ?

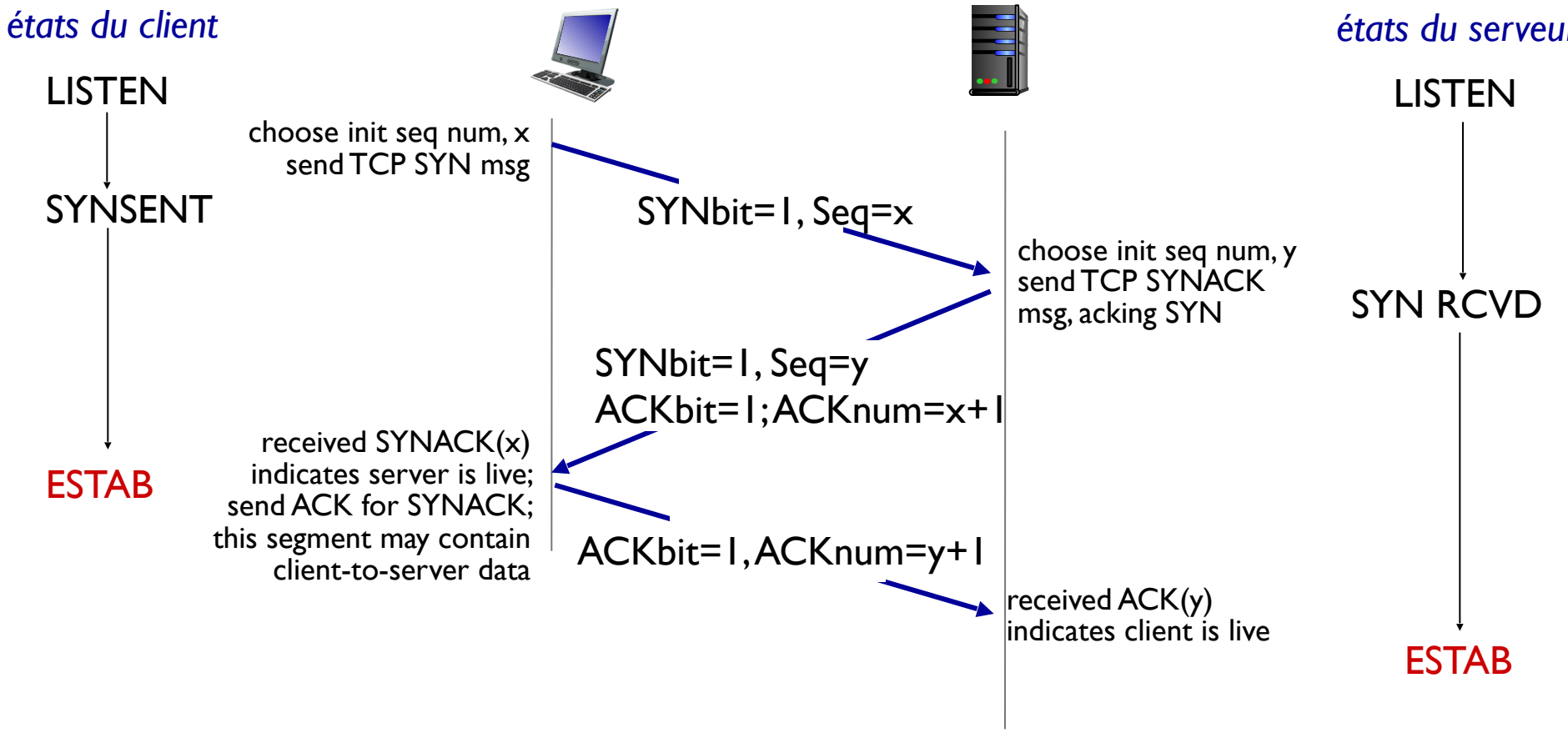
- délai variable
- retransmission de messages (e.g. req_conn(x)) en cas de perte
- messages reçus en désordre
- état interlocuteur inconnu

2-way handshake : exemples avec fautes

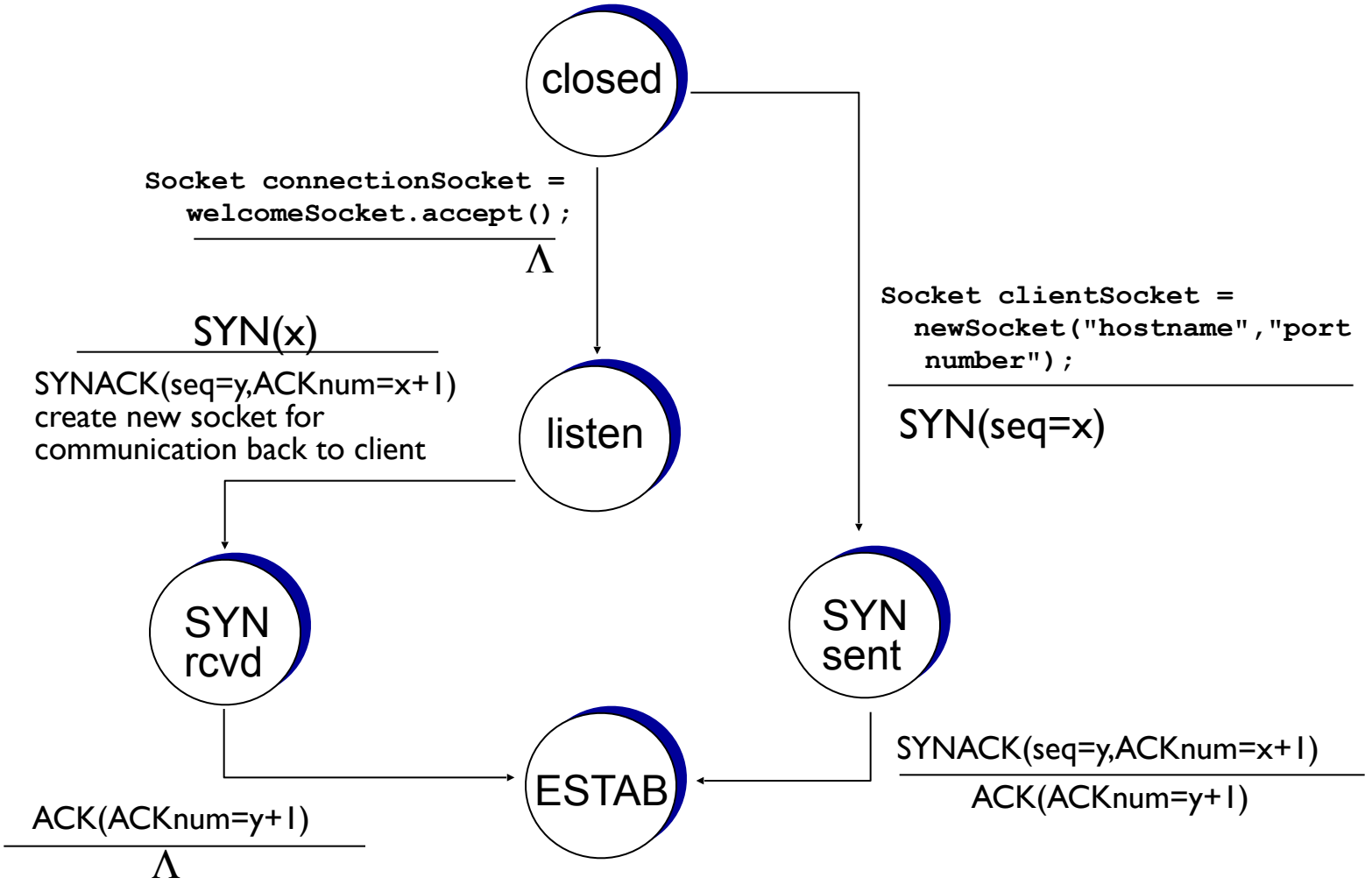




TCP 3-way handshake



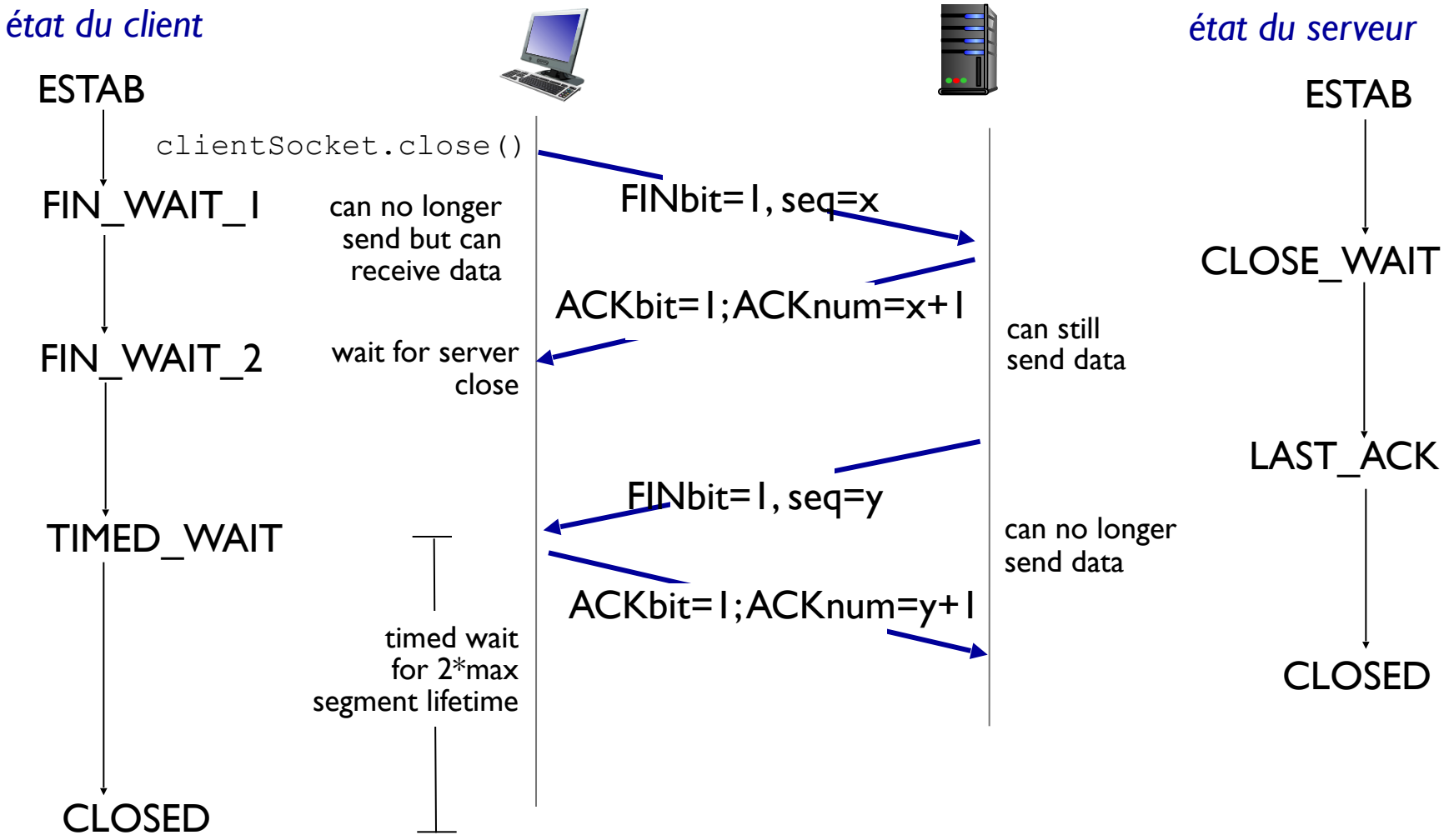
TCP 3-way handshake: Automate



TCP: Fermeture d'une connexion

- le client et le serveur server ferme leur coté de la connexion
  - envoi d'un segment TCP avec bit FIN = 1
- réponse à un segment FIN avec ACK
  - après réception de FIN, segment ACK peut être combiné avec son propre FIN
- échanges simultanés de FIN supportés

TCP : fermeture de connexion



TCP : contrôle de congestion

Contrôle de la congestion

congestion:

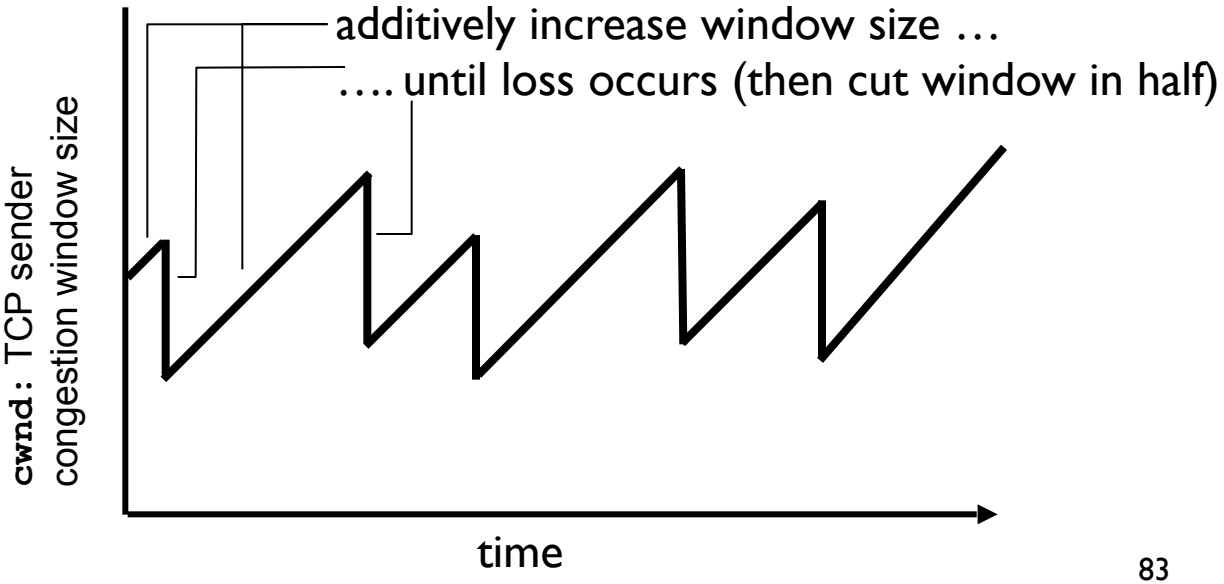
- Informellement: "de trop nombreuses sources émettent trop rapidement des données pour qu'elles puissent être transférées par le réseau"
- Diffère du contrôle de flot !
- Symptômes :
  - Perte de paquet (buffers des routeurs saturés)
  - Délais importants (file d'attente dans les routeurs)

Congestion

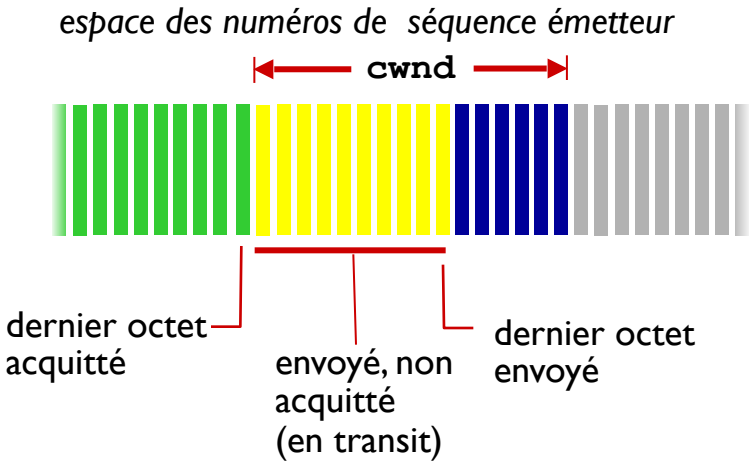
Contrôle de congestion TCP : croissance additive, décroissance multiplicative

- *principe:* émetteur augmente la vitesse de transmission (taille de la fenêtre), testant la bande passante utilisable, jusqu'à ce qu'une perte se produise
  - *augmentation additive:* incrément **`cwnd`** by 1 MSS chaque RTT jusqu'à ce qu'une perte soir détectée
  - *décroissance multiplicative:* diviser **`cwnd`** par 2 après perte

comportement en dents de scie



Contrôle de congestion TCP



vitesse de transmission

- *approximativement:* envoi de cwnd octets, attente de ACK pendant RTT, puis envoi d'octets supplémentaire

$$\text{débit} \approx \frac{\text{cwnd}}{\text{RTT}} \text{ octets/sec}$$

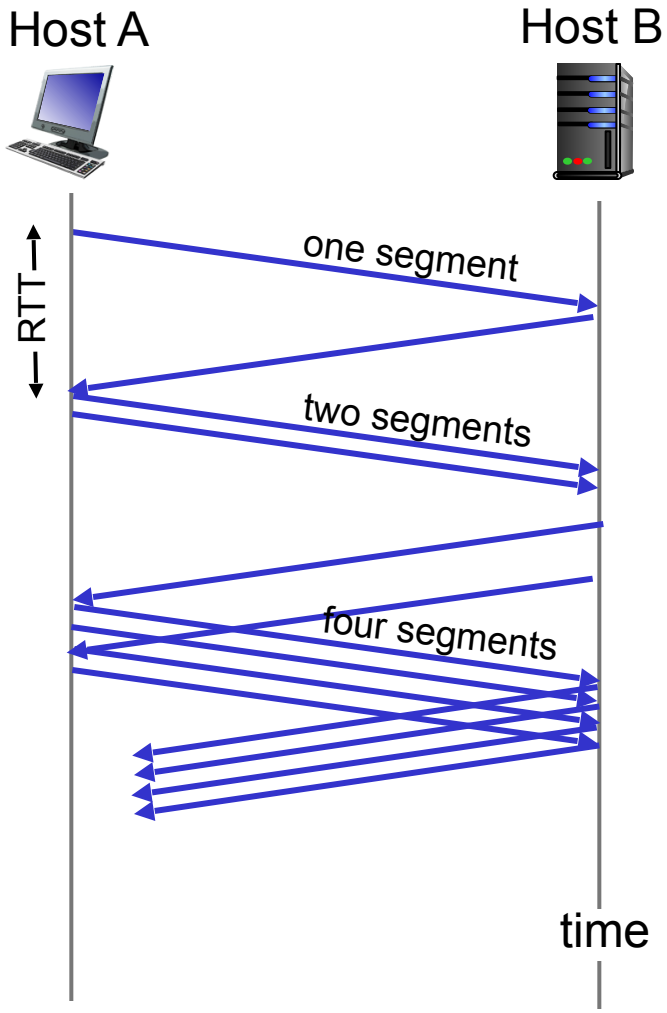
- Limite émetteur:

  LastByteSent-LastByteAcked $\leq$ **cwnd**

- **cwnd** est dynamique fonction de l'estimation de la congestion du réseau

TCP Slow Start

- Au début de la connexion, augmentation exponentielle du débit tant que pas de perte
  - initialement **`cwnd`** = 1 MSS
  - double **`cwnd`** chaque RTT
  - -> incrementation de **`cwnd`** à chaque réception de ACK
- *résumé:* vitesse de transmission faible initialement, mais augmente exponentiellement



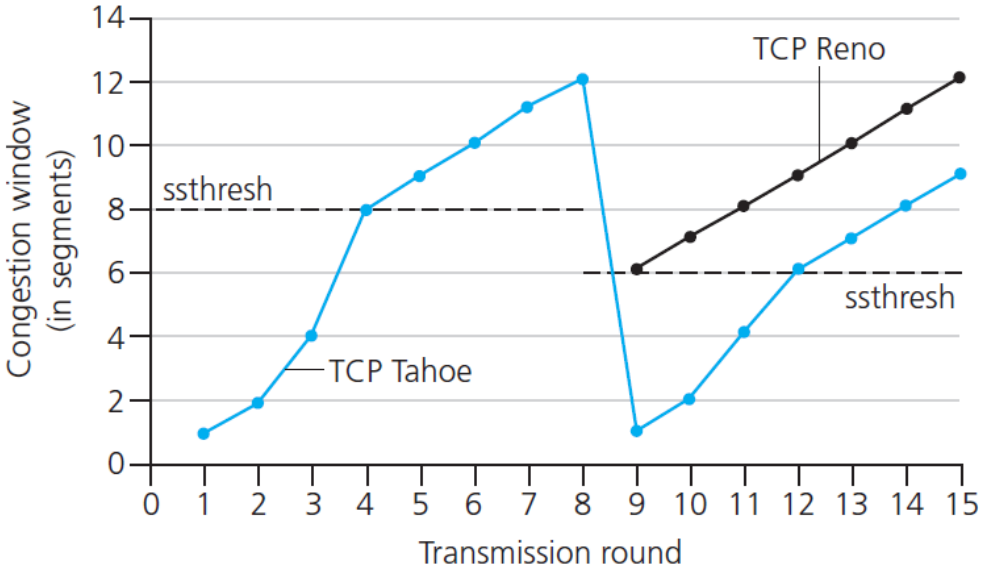
TCP: détection et réaction aux pertes

- perte détecté par expiration du temporisateur :
  - **cwnd** remis à 1 MSS;
  - fenêtre passe ensuite en croissance exponentielle (comme dans slow start) jusqu'au seuil, puis croissance linéaire
- perte détecté par 3 ACKs dupliqués : TCP RENO
  - réseau capable de transférer des segments
  - **cwnd** est divisé par 2, puis croissance linéaire
- TCP Tahoe **cwnd** toujours remis à 1 MSS

TCP: passage de slow start à CA

Q: Quand doit-on passer de croissance exponentielle à linéaire

A: Quand **`cwnd`** atteint 1/2 de sa valeur au moment de la détection de perte

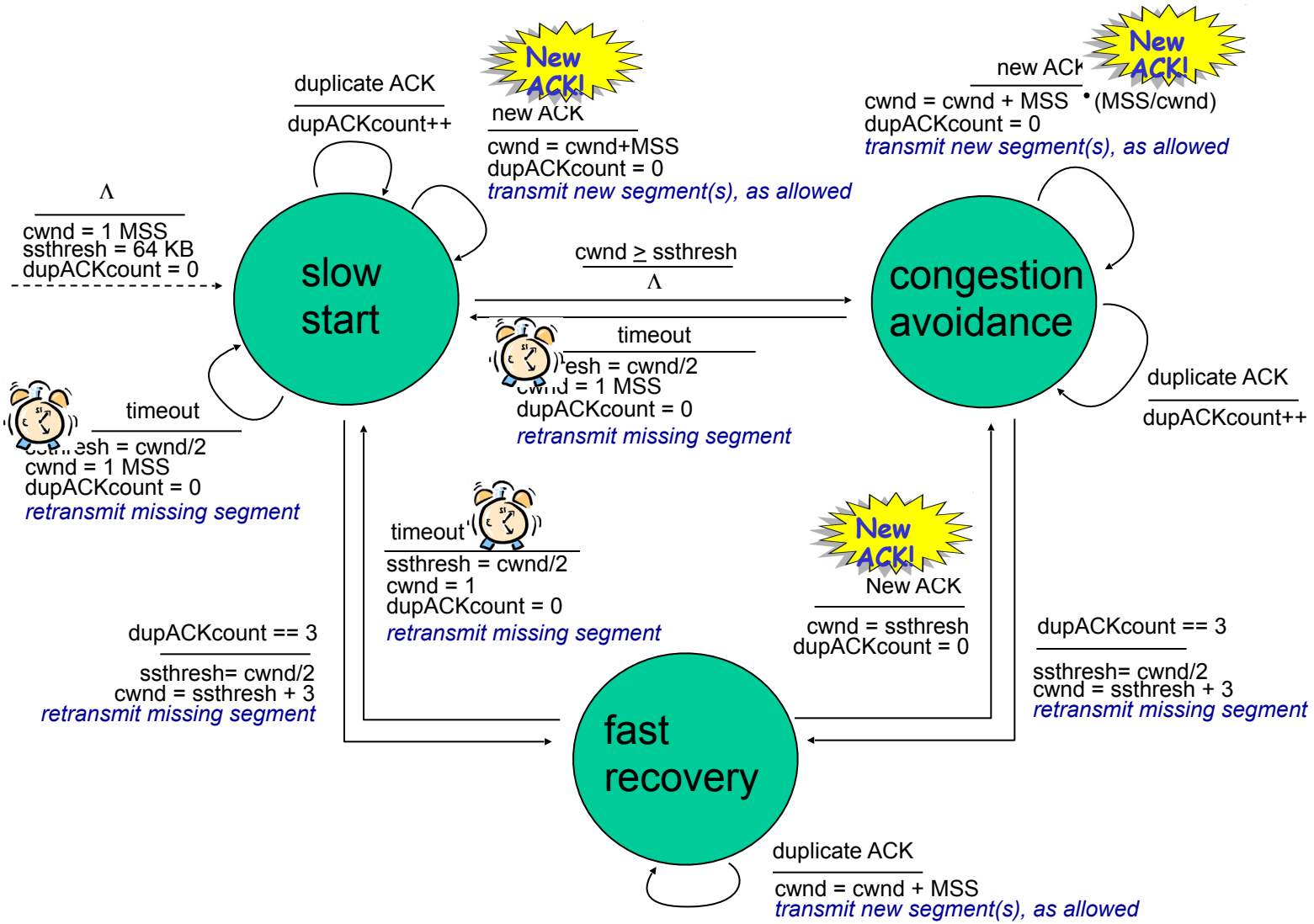


Implementation:

- variable **`ssthresh`**
- lors d'une perte, **`ssthresh`** est mis à **`cwnd`**/2

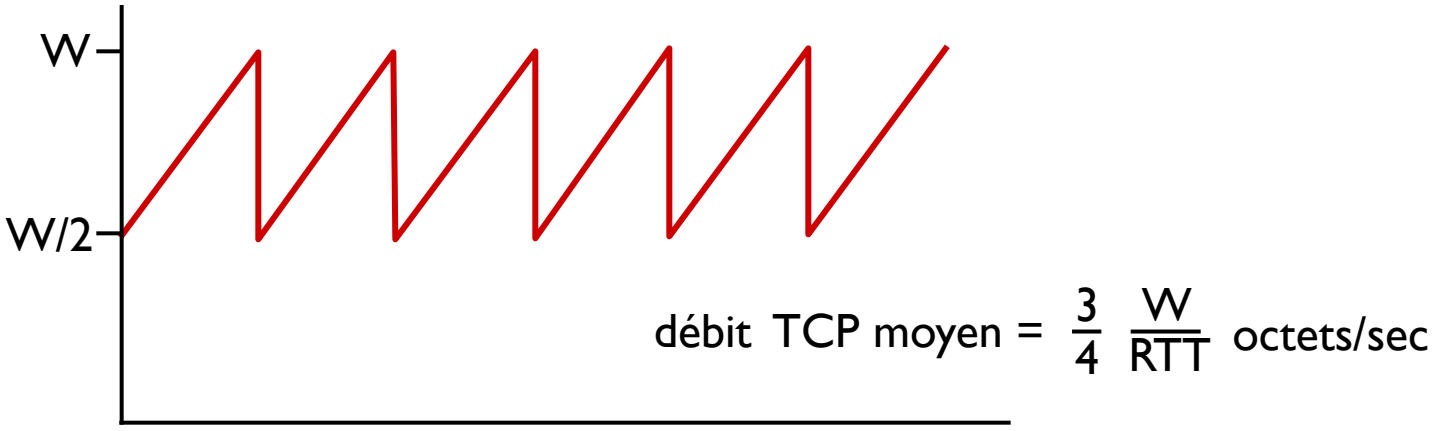
* CA : congestion avoidance

Summary: TCP Congestion Control



débit TCP

- débit TCP moyen comme fonction de la taille de la fenêtre et de RTT?
  - en ignorant slow start et en supposant que la couche appli. a toujours des données à envoyer
- W: taille de la fenêtre (en octet) quand une perte a lieu
  - nombre d'octets moyen en transit = ¾ W
  - débit moyen = 3/4W per RTT



TCP et réseaux haut débit

- exemple: segment 1500 octet, RTT100ms, on veut débit 10 Gbps
- nécessaire : W = 83 333 segments en transit
- débit en fonction la probabilité L de perte de segment [Mathis 1997]:

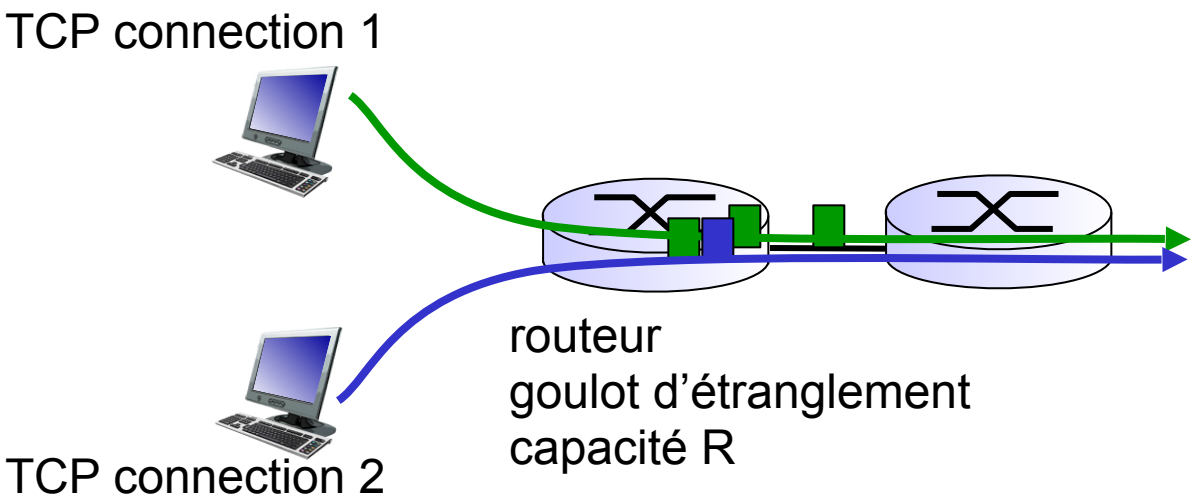
$$\text{débit TCP} = \frac{1.22 \cdot MSS}{RTT\sqrt{L}}$$

➜ pour obtenir 10 Gbps de débit, on doit avoir $L = 2 \cdot 10^{-10}$ – *taux de perte très faible !*

- nouvelles version de TCP adapté aux réseaux haut débit

équité et TCP

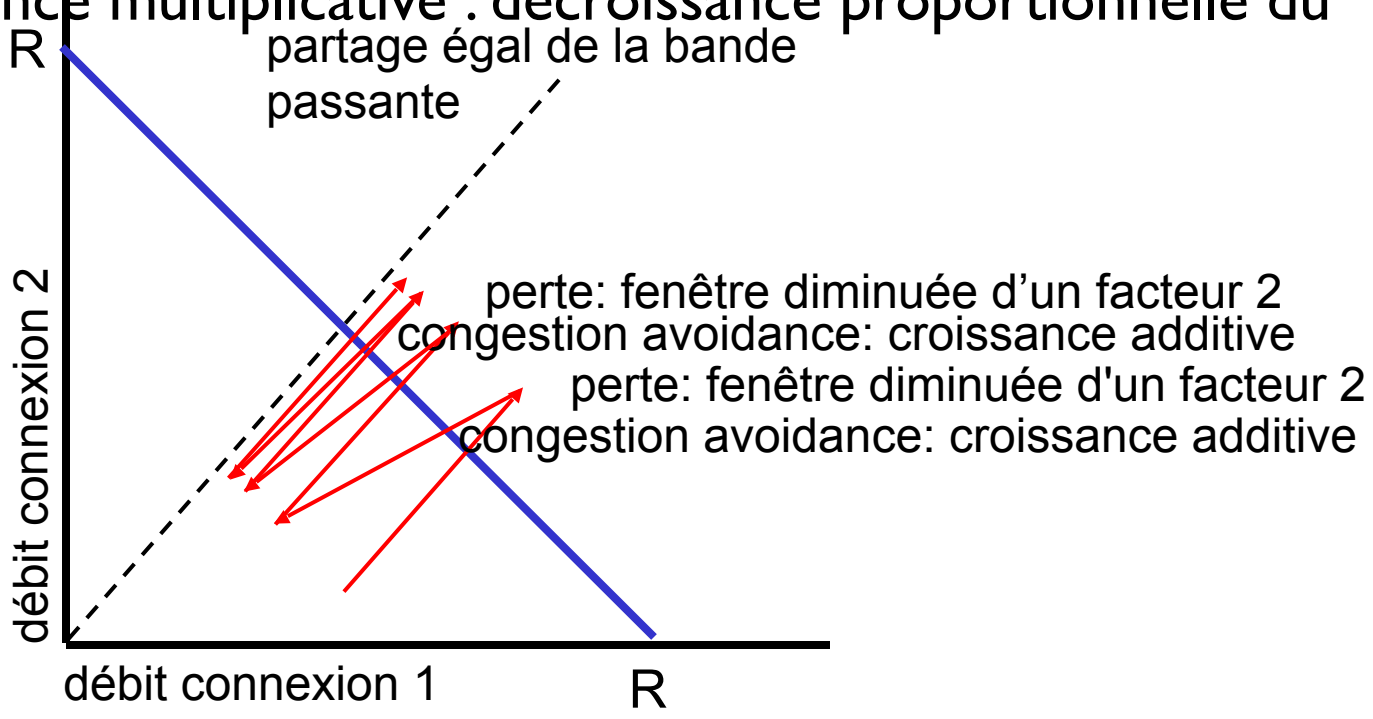
objectif : si K sessions TCP partage le même lien de bade passante R, chacune devrait obtenir un débit moyen de R/K



Pourquoi TCP est équitable ?

Deux sessions parallèle, en competition pour la bande passant:

- augmentation additive : pente de l'évolution du débit
- décroissance multiplicative : décroissance proportionnelle du débit



équité (cont')

équité et UDP

- Apps multimedia souvent n'utilisent pas TCP
  - évite que leur débit soit contrôlé par TCP
- Reposent sur UDP:
  - envoi d'audio/video at vitesse constante, tolère des pertes de paquet

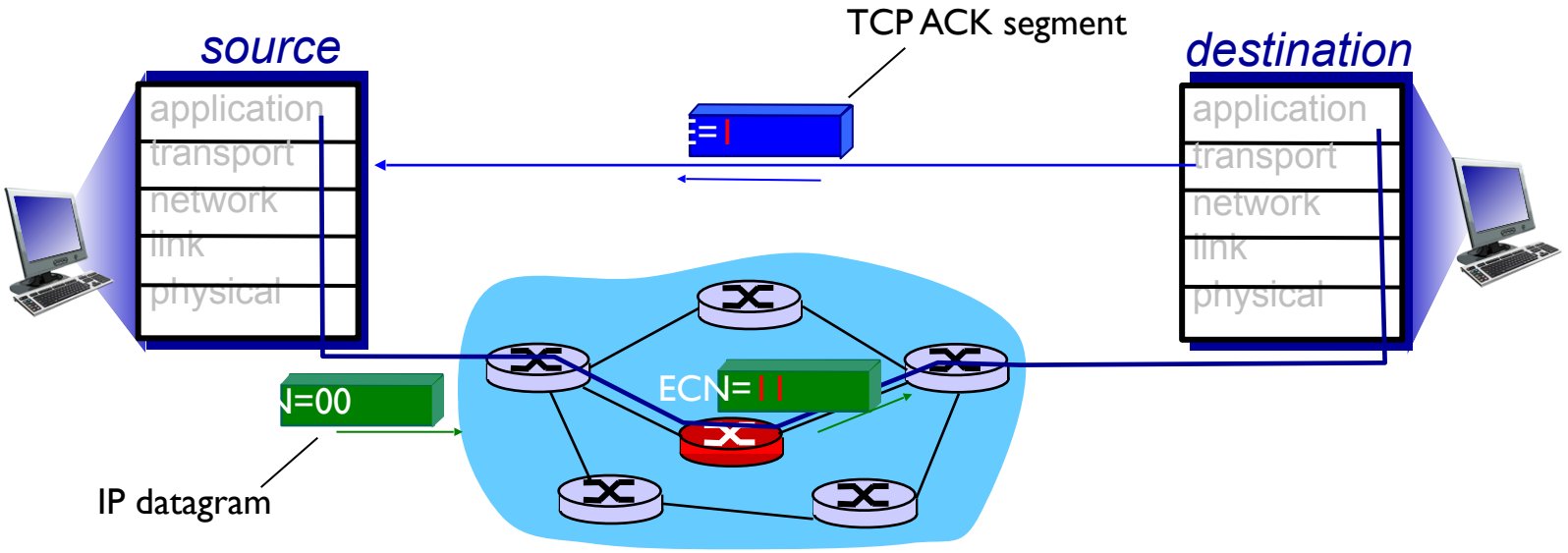
équité et connexions TCP parallèles

- plusieurs connexions parallèles ouvertes par une même app.
- ex: navigateurs web
- e.g., lien de capacité R avec 9 connexions existantes:
  - autre app demande 1 connexion TCP → débit R/10
  - autre app demade 11 connexions TCPs → débit R/2

Explicit Congestion Notification (ECN)

Contrôle de congestion assisté par le réseau

- des bits de l'entête IP header modifiés *par les routeurs* pour indiquer la congestion
- ces indications sont conservées jusqu'à la machine hôte
- récepteur peut envoyer ces informations à l'émetteur via les segments ACK



Résumé

- Principes mise en oeuvre par la couche transport :
  - multiplexage
  - transfert de données fiable (sans perte et dans l'ordre)
  - contrôle de flot
  - contrôle de congestion
- Implementation dans Internet
  - protocole TCP