Les réseaux

2

Couche Transport

D’après le livre :

Analyse structurée des réseaux

Jim Kurose, Keith Ross
Pearson Education

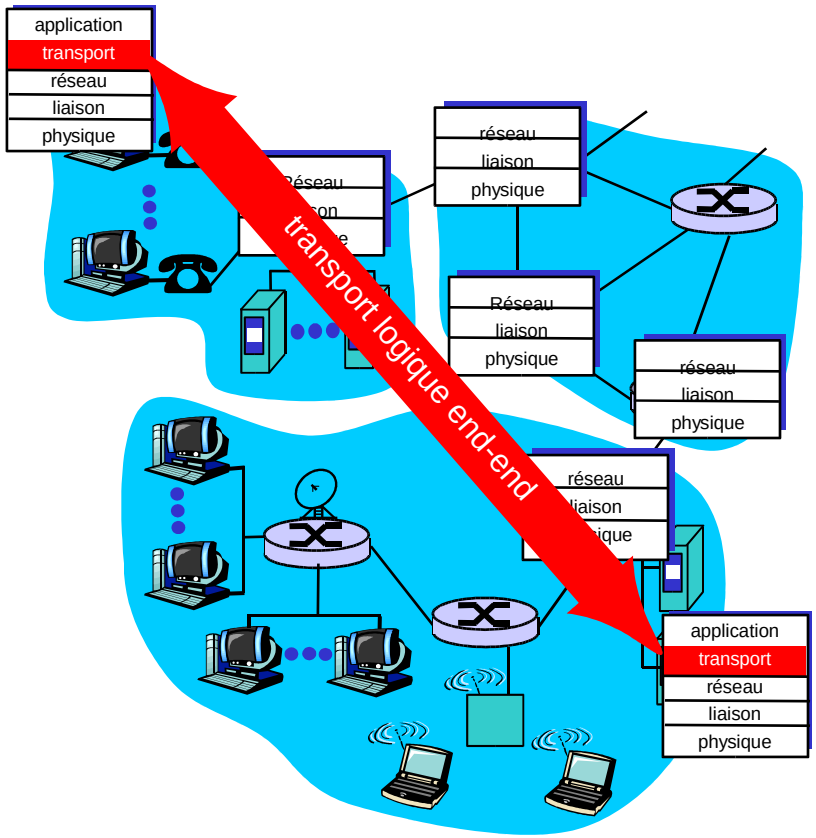
Adaptation : AbdelAli ED-DBALI (AbdelAli.Ed-Dbali@lifo.univ-orleans.fr)

Couche Transport : buts du chapitre

- Comprendre les principes derrière les services de la couche transport :
  - multiplexage/démultiplexage
  - Transfert fiable de données
  - Contrôle de flux
  - Contrôle de congestion
- Étudier les protocoles Internet de la couche transport :
  - UDP : transport en mode non connecté
  - TCP : transport en mode non connecté
  - Contrôle de congestion dans TCP

Transport : services et protocoles

- Fourni la *communication logique* entre les processus tournant sur différents systèmes
- Les protocoles de transport tournent sur les postes terminaux
  - Émetteur : cinder messages des applis dans des segments, les passer à la couche réseau
  - Récepteur : réassemble les segments en messages, les passer à la couche applis
- Plus d'un protocole de transport disponibles aux applis
  - Internet: TCP et UDP



Couche transport vs. Couche réseau

- *Couche réseau :*
  communication logique entre machines
- *Couche transport :*
  communication logique entre processus
  - *retransmet* et *met en valeur* les services de la couche réseau

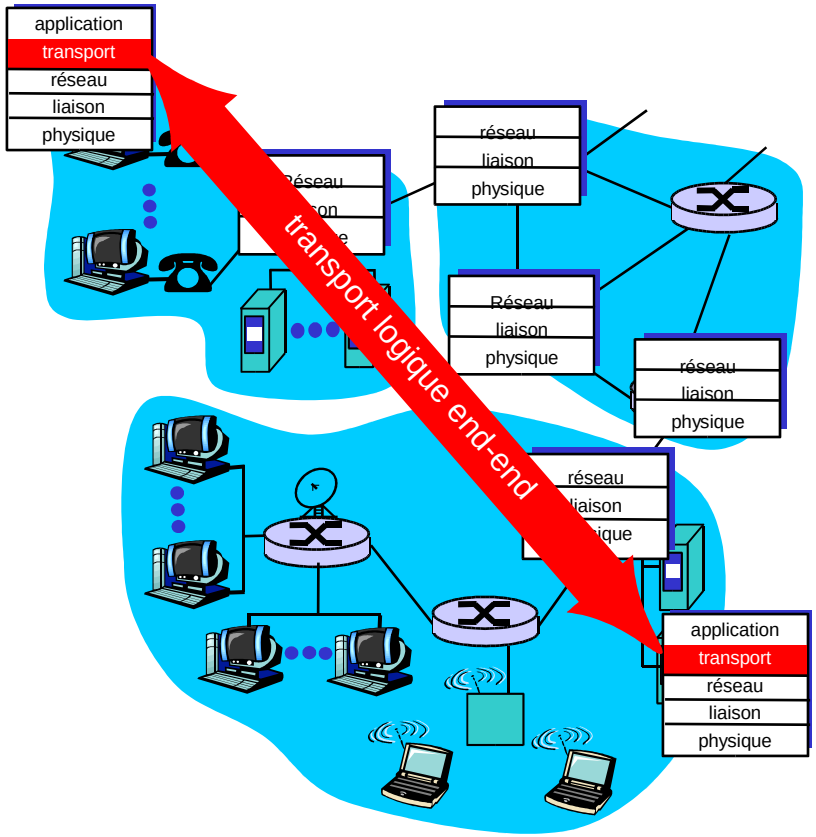
Analogie :

12 enfants envoient des lettres à 12 enfants

- processus = enfants
- messages d'applis = lettres dans les enveloppes
- machines = maisons
- protocole de transport = Anne et Bill
- protocole couche réseau = service de poste

Protocoles de la couche transport dans l'Internet

- Acheminement *fiable* et *ordonné* (TCP)
  - contrôle de congestion
  - contrôle de flux
  - établir la connexion
- Acheminement *non fiable* et *non ordonné* : UDP
  - Pas d'extension significative du “best-effort” de l'IP
- Services indisponibles :
  - garanties des délais
  - garanties de bandes



Multiplexage/démultiplexage

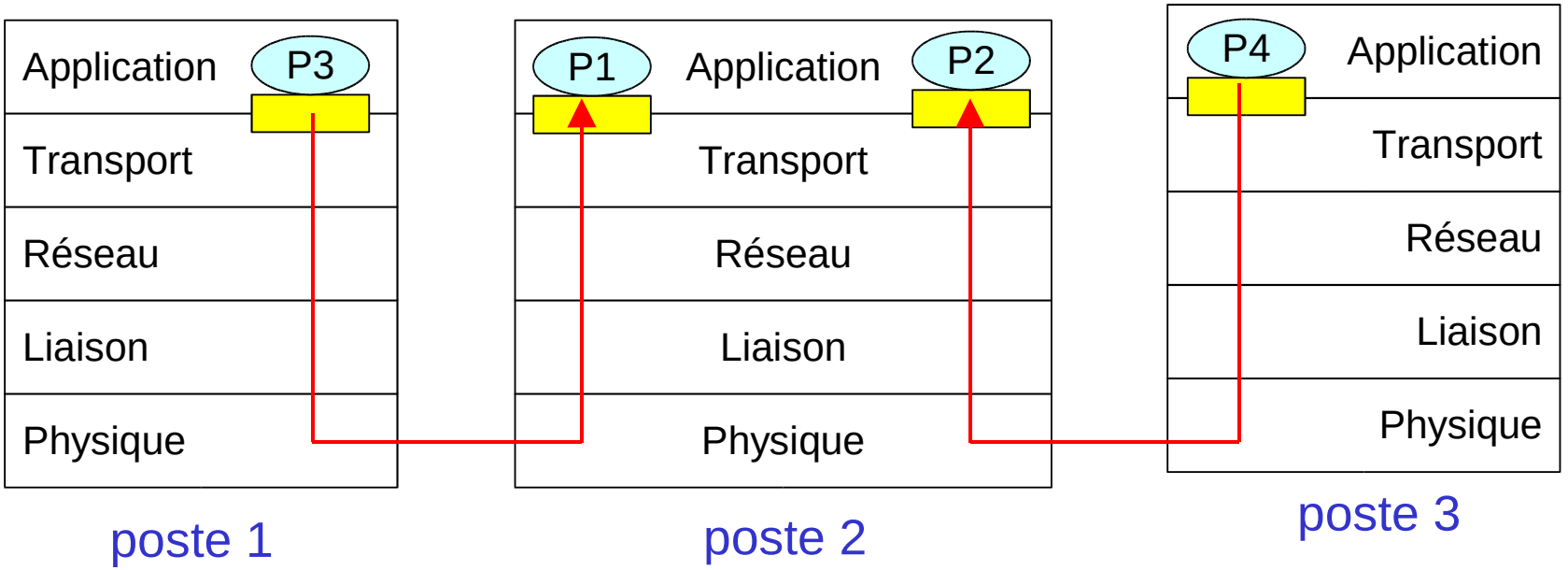
Démultiplexage à la récpt

Acheminement des segments reçus à la bonne socket

Multiplexage à l'émission

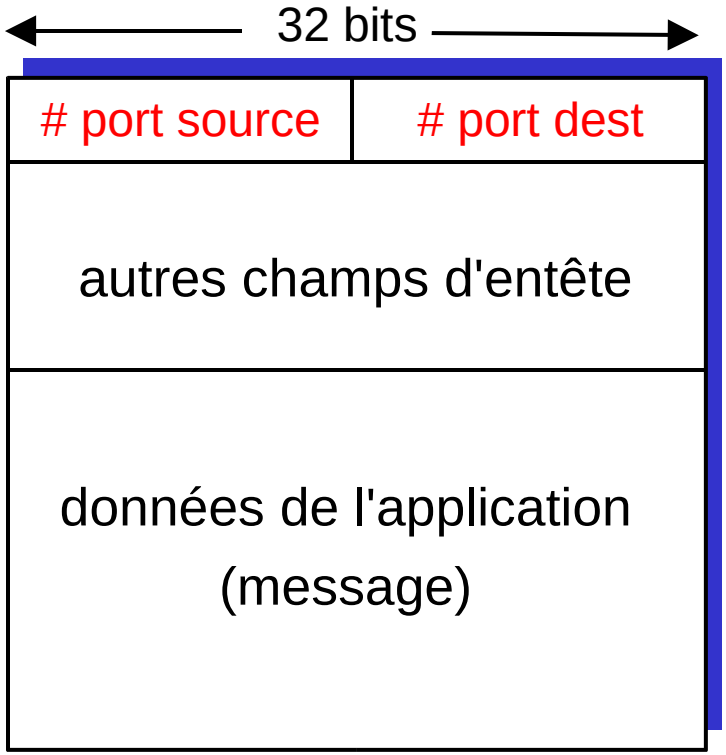
Rassemblement des données de plusieurs sockets, enveloppement des données avec les entêtes (utilisées pour démultiplexage)

= socket = processus



Démultiplexage : comment ça marche

- Le poste reçoit les datagrammes IP
  - Chaque datagramme a une adresse IP source, une adresse IP destination
  - Chaque datagramme transporte 1 segment
  - Chaque segment a un port source, un port destination (rappel : les numéros de port pour les applications spécifiques)
- Le poste utilise les adresses IP & les numéros de port pour délivrer le segment à la bonne socket



format segment TCP/UDP

Démultiplexage en mode non-connecté

- Creation de sockets avec numéros de port :

```
DatagramSocket mySocket1 = new
   DatagramSocket(99111);
DatagramSocket mySocket2 = new
   DatagramSocket(99222);
```

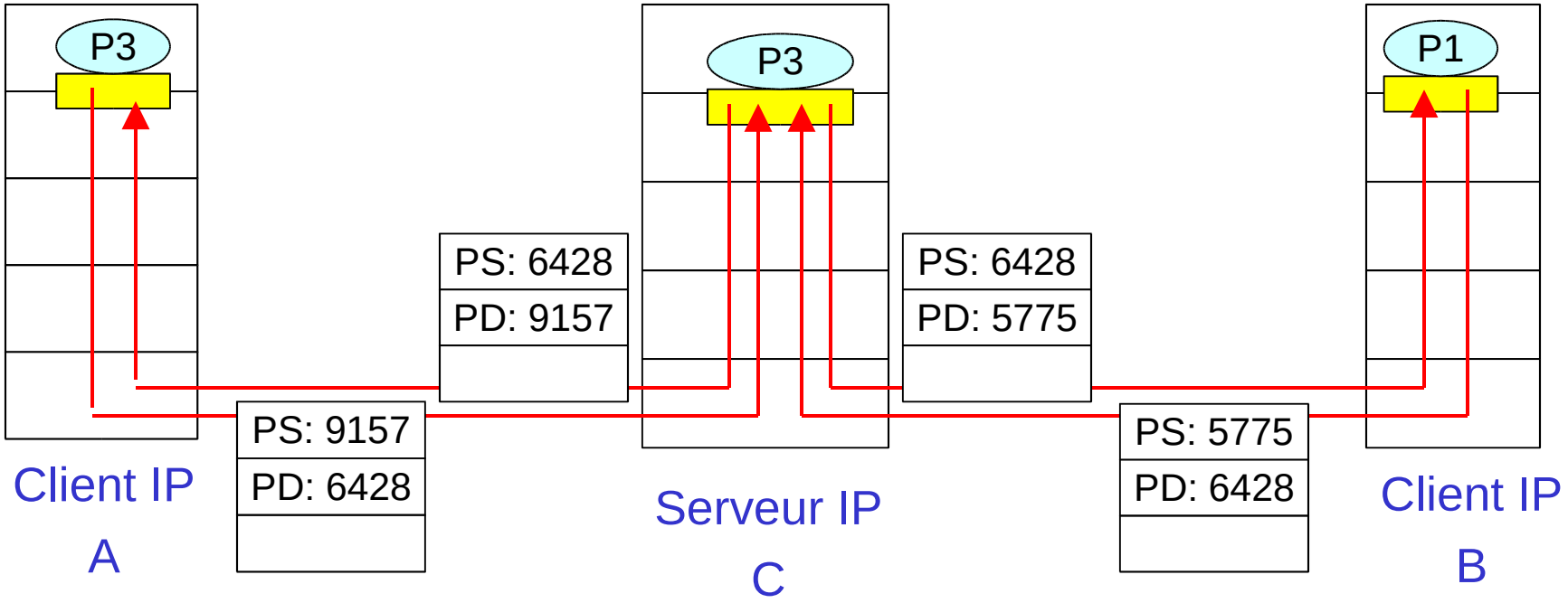
- Socket UDP identifiée par le couple :

(adresse IP dest, numéro port dest)

- Quand le poste reçoit les segments UDP :
  - Vérifie le num. de port dest. dans le segment
  - dirige le segment UDP vers la socket ayant ce num. de port
- Les datagrammes IP avec adresses IP source et/ou numéros de port source différents : dirigés vers la même socket

Démultiplexage en mode non-connecté (suite)

```
DatagramSocket serverSocket = new DatagramSocket(6428);
```

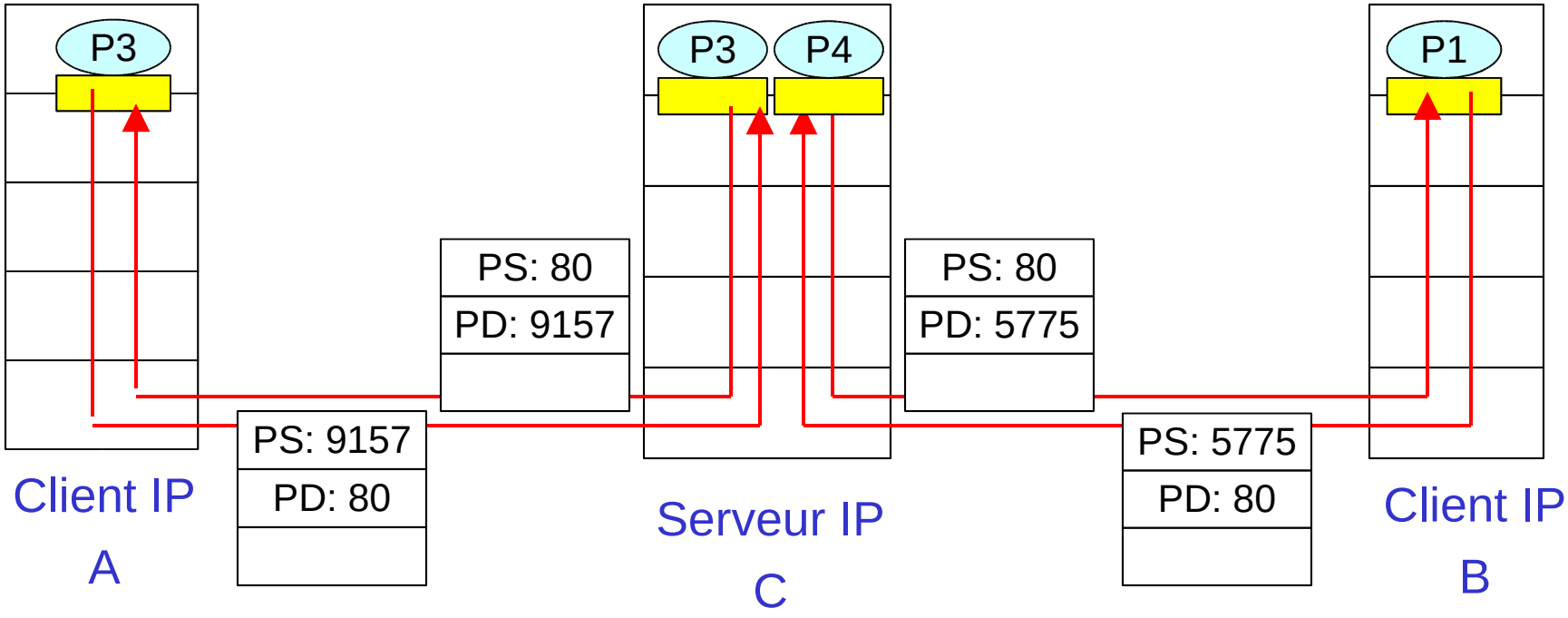


PS permet “adresse de retour”

Démultiplexage en mode connecté

- La socket TCP identifiée par le quadruplet :
  - adresse IP source
  - numéro de port source
  - adresse IP dest
  - numéro de port dest
- Le récepteur utilise les 4 valeurs pour diriger le segment vers la bonne socket
- Le serveur peut supporter des sockets TCP simultanées :
  - Chaque socket identifiée par son propre quadruplet
- Les serveurs Web ont des sockets différentes pour chaque client connecté
  - Les connexions HTTP non-persistantes ont des sockets différentes pour chaque requête

Démultiplexage en mode connecté (suite)



UDP: User Datagram Protocol [RFC 768]

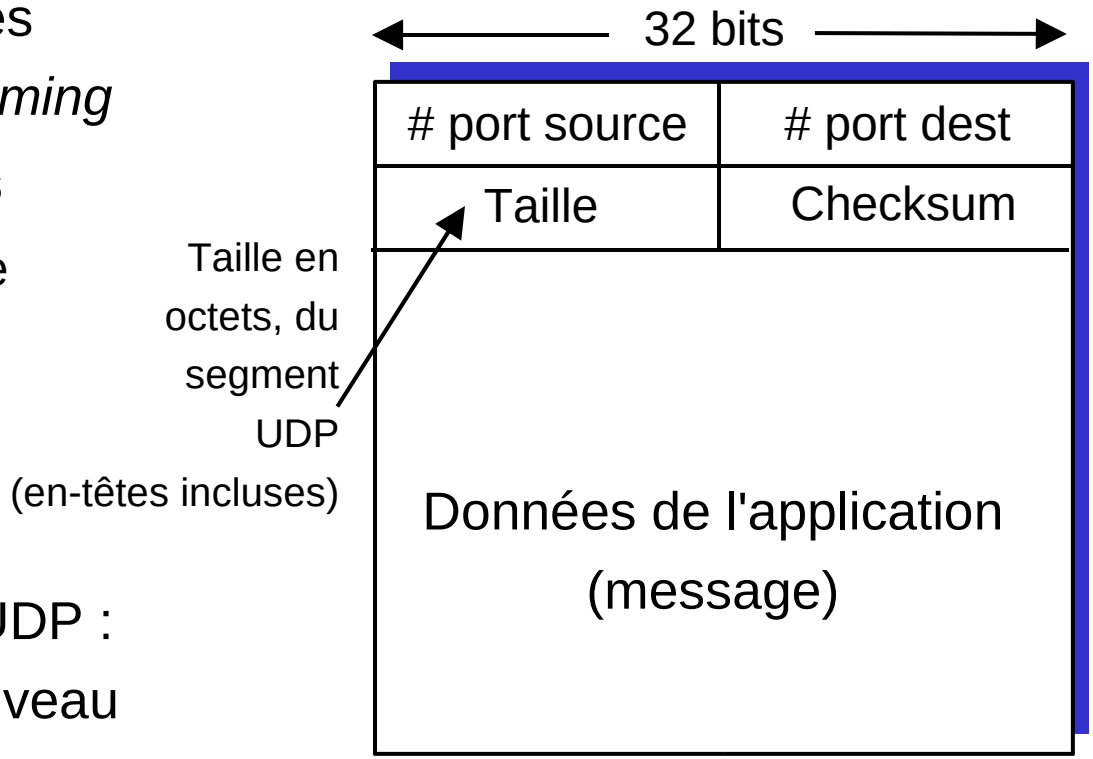
- Protocole Internet de transport *minimaliste*, “os nu”
- Son service “best effort” : les segments UDP peuvent :
  - être perdus
  - délivrés aux applis dans le désordre
- *Mode non-connecté :*
  - Pas de poignée de main entre l'émetteur et le récepteur UDP
  - Chaque segment UDP géré indépendement des autres

Pourquoi UDP ?

- pas d'établissement de connexion (qui peut ajouter un retard)
- simple: pas d'état de connexion à gérer au niveau de l'émetteur et du récepteur
- petits en-têtes des segments
- pas de contrôle de congestion

UDP (suite)

- Utilisé souvent dans les apps *multimedia-streaming*
  - tolérantes aux pertes
  - sensibles à la vitesse
- Autres utilisations :
  - DNS
  - SNMP
- Transfert fiable avec UDP : ajouter la fiabilité au niveau de la couche application
  - gestion des erreurs du transport au niveau de l'application !



Format du segment UDP

checksum UDP

But : détecter les “erreurs” (ex. bits erronés) dans les transmis

Émetteur

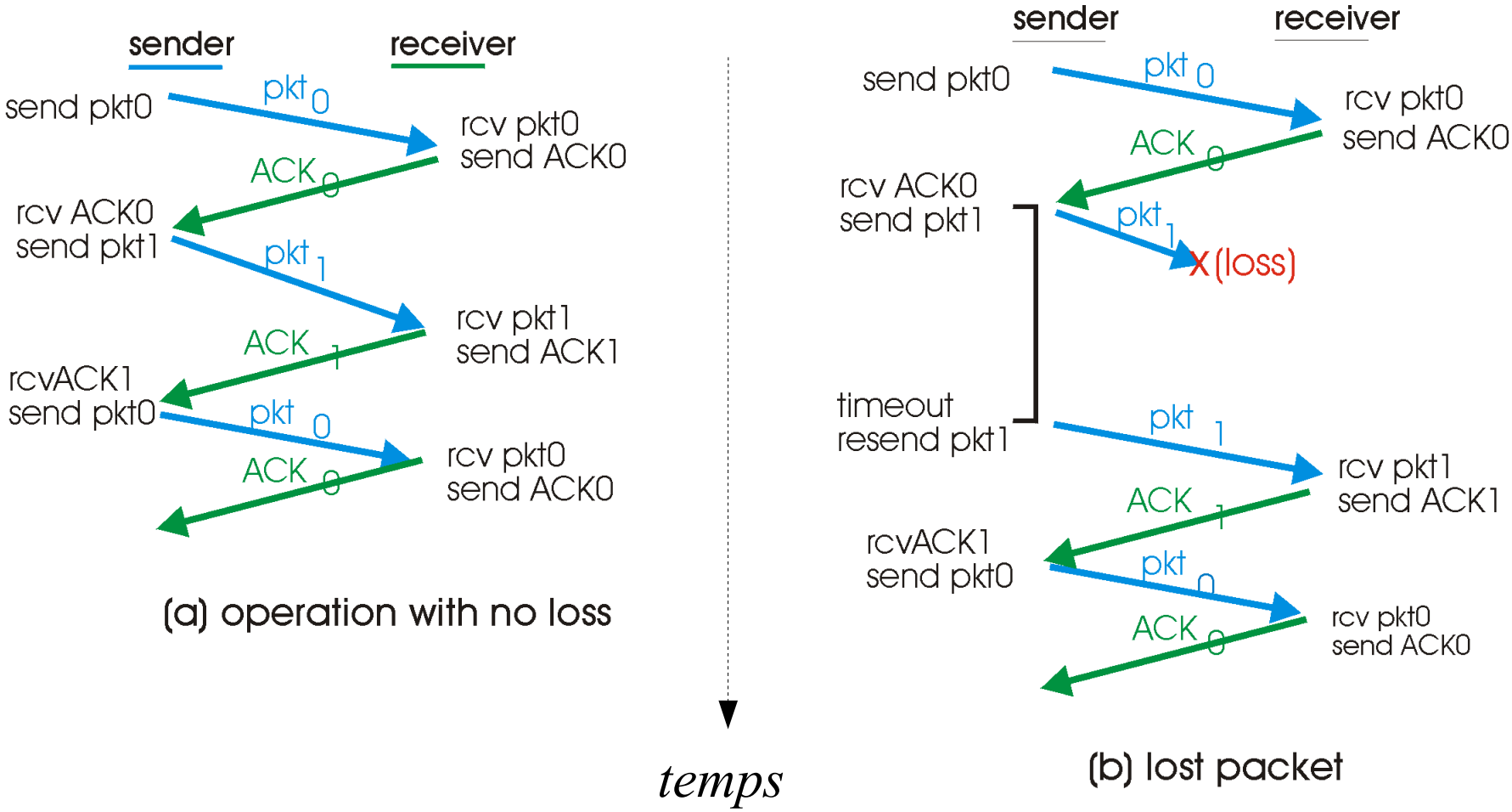
- Traite le contenu du segment comme une séquence d'entiers de 16 bits
- Calcule le *checksum* : addition (complément à 1) du contenu du segment
- Met la valeur calculée dans le champ *checksum* du segment UDP

Récepteur

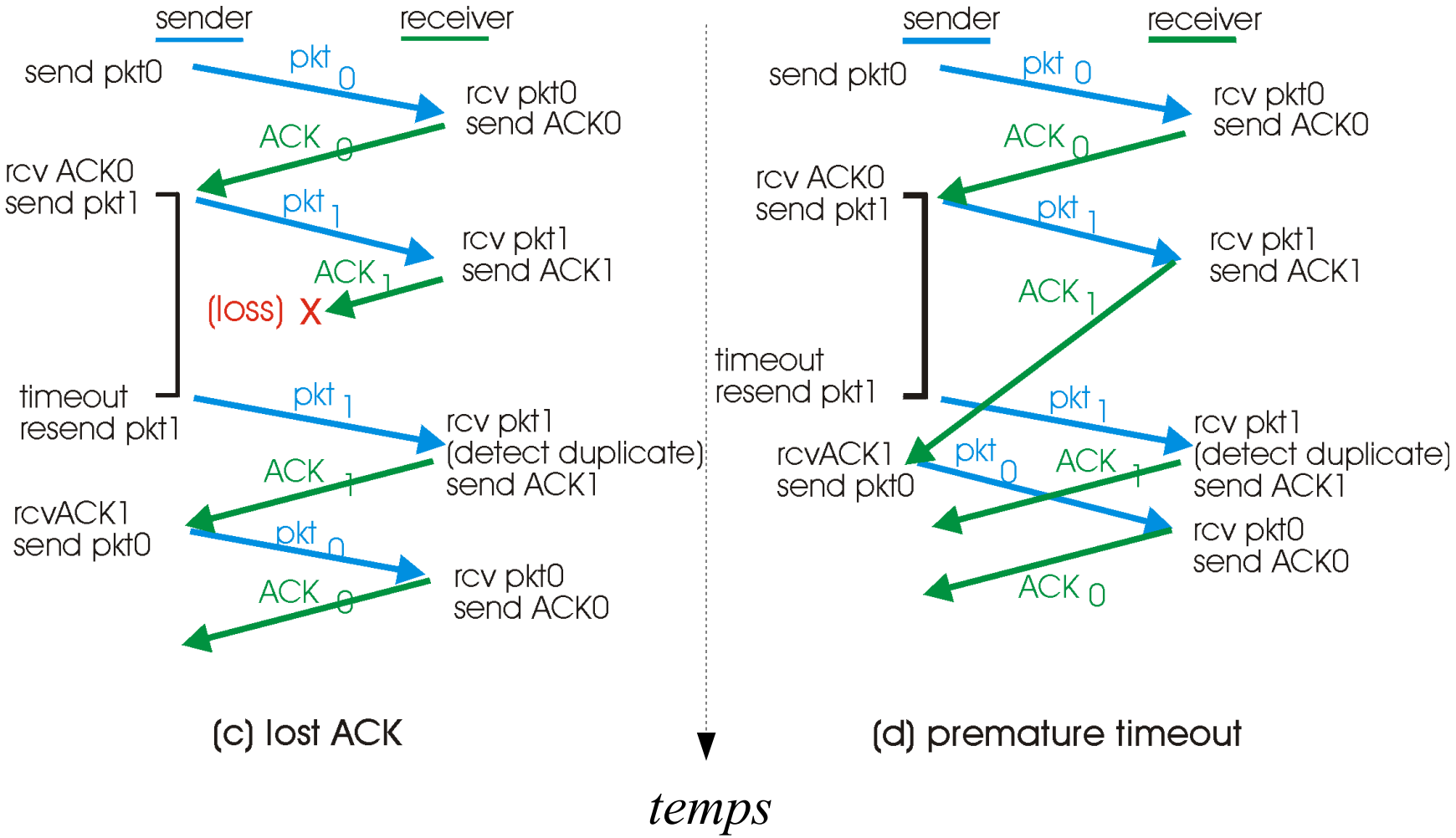
- Calcule le checksum du segment reçu
- Vérifie si checksum calculé égale au champ checksum :
  - NON - erreur détectée
  - OUI - pas d'erreur détectée... *mais y'en a-t-il malgré tout ?* à suivre….

Exemples d'echange

Protocole d'échange avec possibilité de perte ou d'erreur dans les données

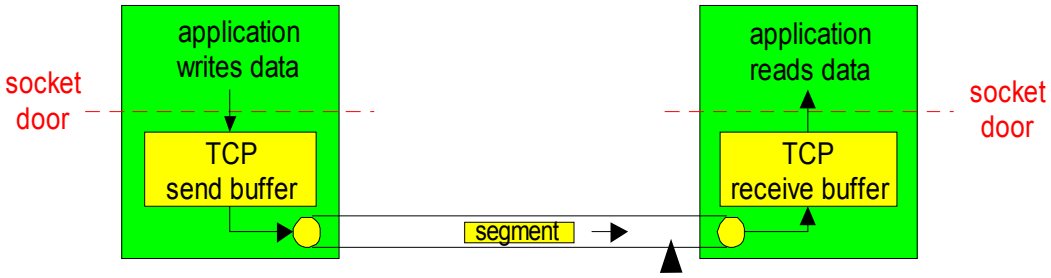


Exemples d'echange (suite)



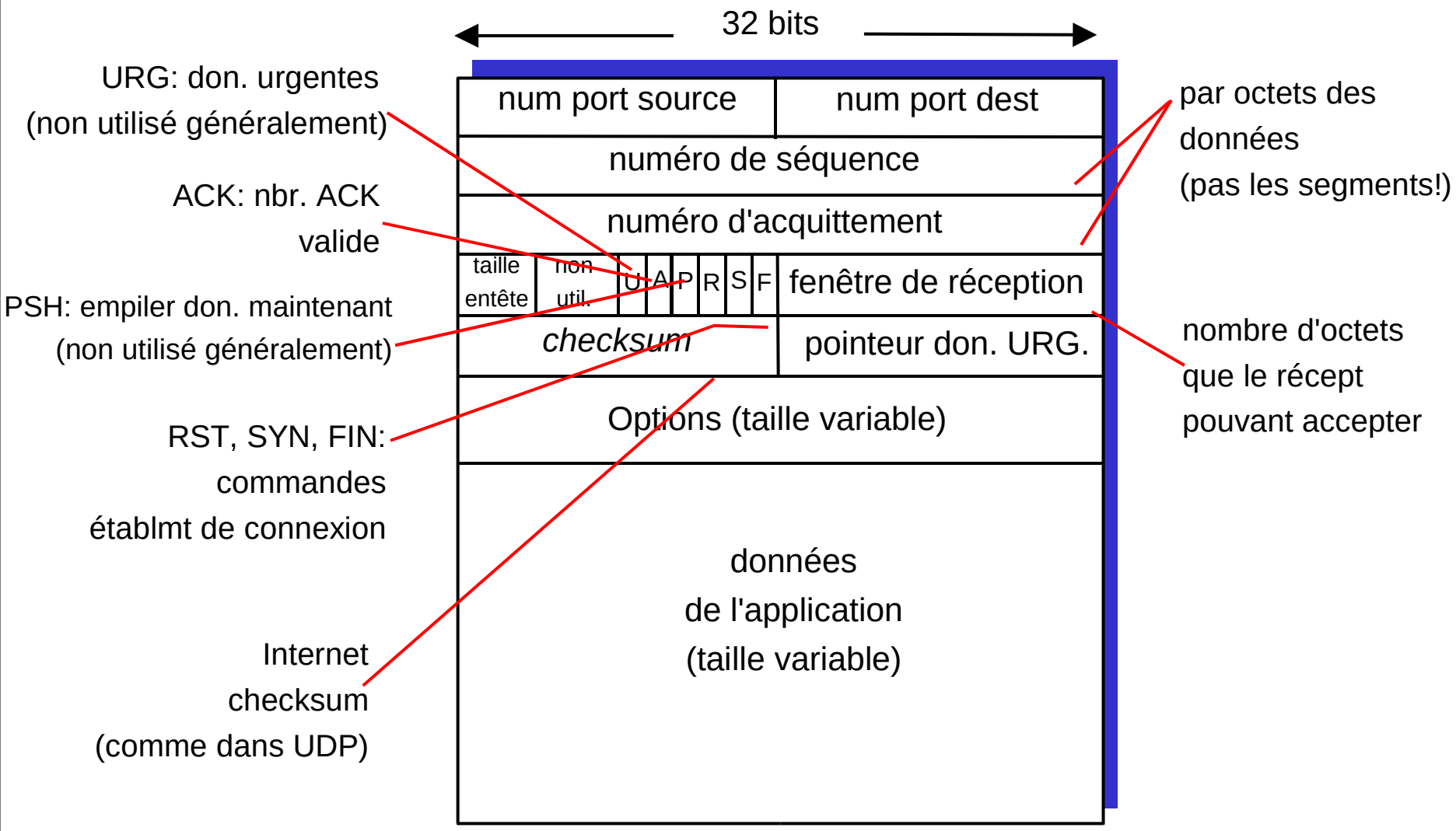
TCP: Vue d'ensemble RFCs: 793, 1122, 1323, 2018, 2581

- point-à-point:
  - un émetteur, un récepteur
- transfert fiable, ordonné:
  - pas de “messages sur la touche”
- pipelining:
  - contrôle de flux et de congestion, positionnement de la taille de la fenêtre d'émission
- *buffers d'émission et de réception*



- données en full duplex:
  - flot de données bi-directionel dans la même connecxion
  - MSS: *maximum segment size*
- mode connecté:
  - initialisation de la connexion (échange de msgs de contrôle) avant l'échange de données
- flux controlé:
  - émetteur ne peut submerger le récepteur

TCP segment structure



TCP: num. de séq. et de ACKs

num. séq.:

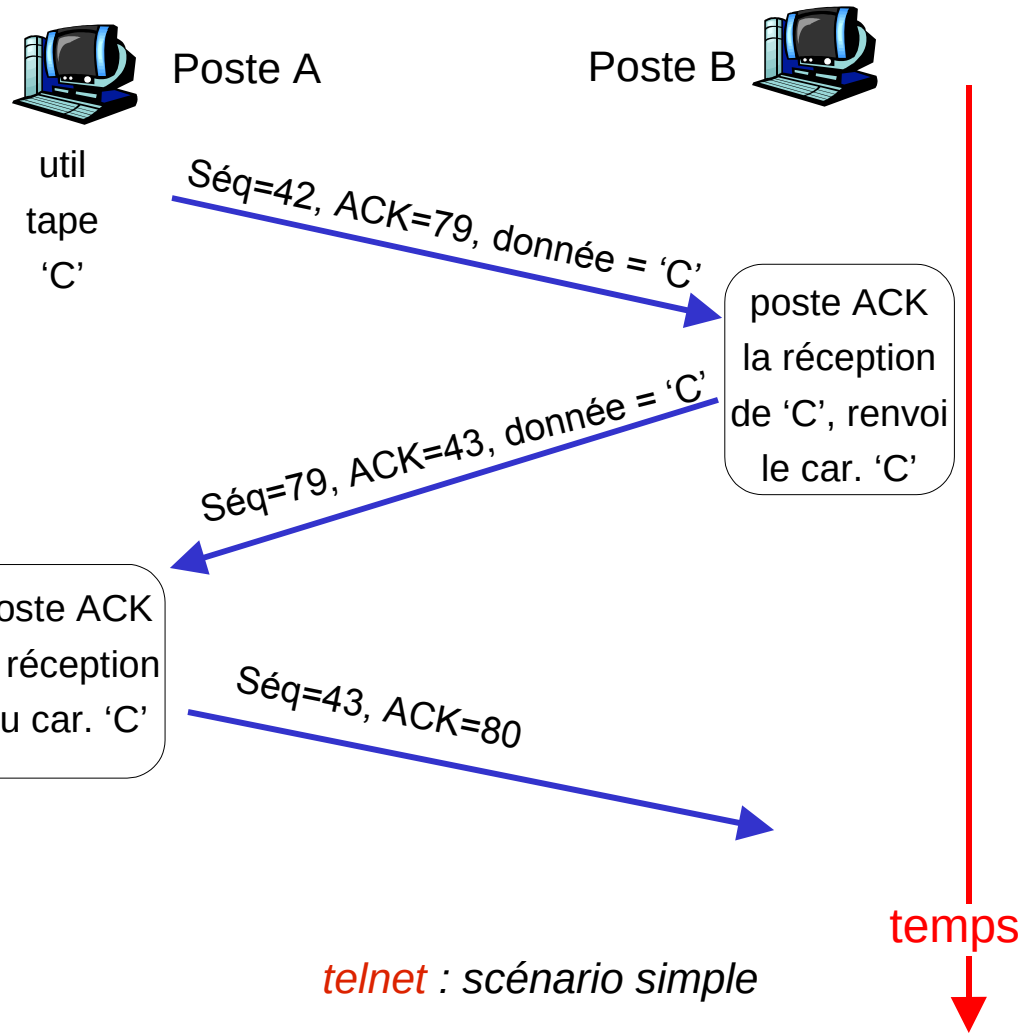
- num. du premier octet du seg. de données dans le flux des octets

ACKs:

- num. de séq du prochain octet attendu
- ACK cumulatif

Q: comment le récepteur traite les segments hors séquence

- R: la spéc de TCP ne dit rien : *implementation dependent*



telnet : scénario simple

TCP: *Round Trip Time* et *Timeout*

Q: comment TCP initialise la valeur de timeout?

- plus grand que RTT
  - mais RTT varie
- trop petit : timeout prématuré
  - retransmissions inutiles
- trop grand : retarde la réaction à la perte d'un segment

Q: comment estimer RTT?

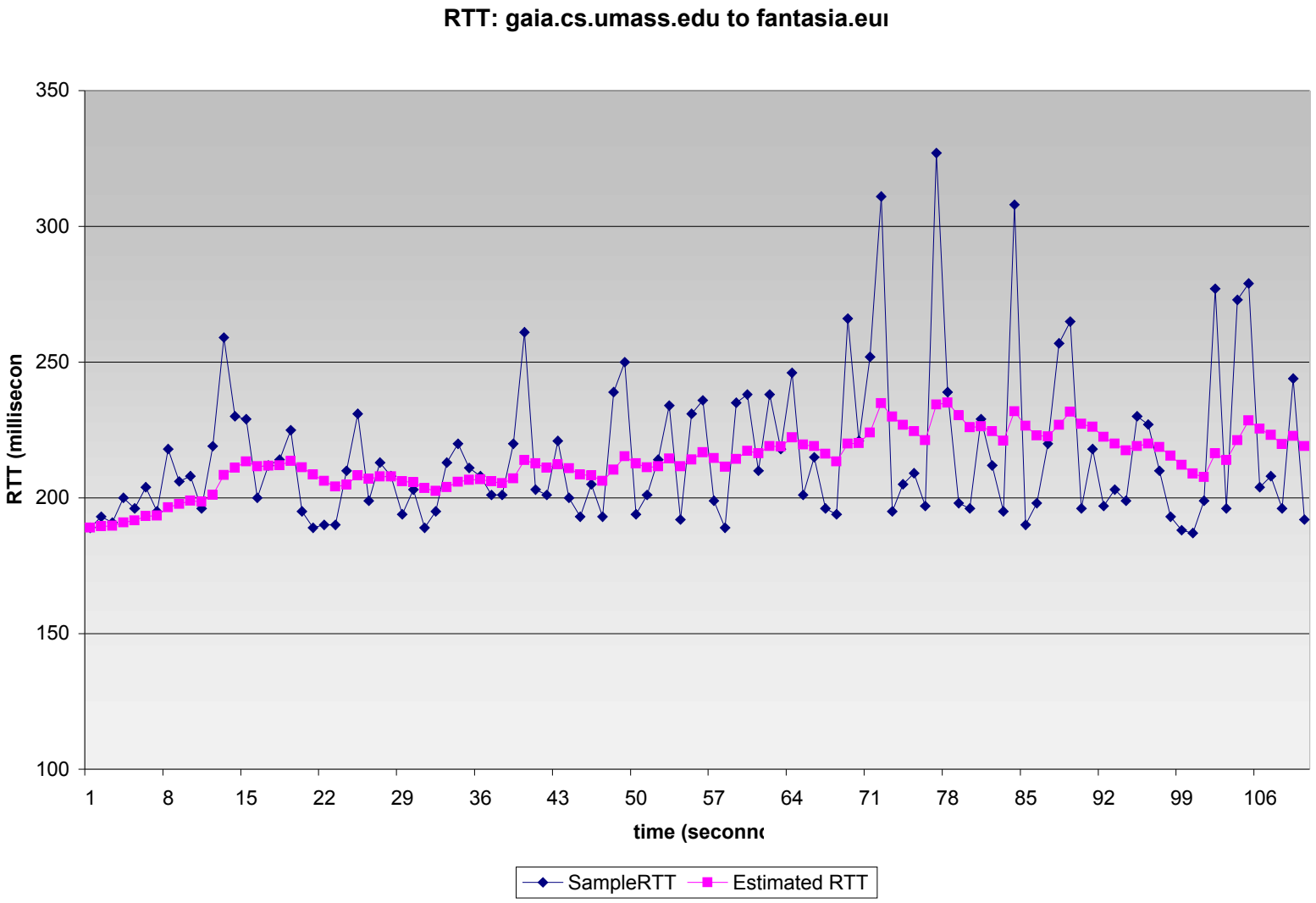
- **`EchantillonRTT`**: temps mesuré depuis la transmission d'un segment jusqu'à la réception du ACK
  - en ignorant les retransmissions
- **`EchantillonRTT`** varie => lisser l'estimation de RTT
  - moyenne de plusieurs mesures récentes
  - pas seulement le **`EchantillonRTT`** courant

TCP: Round Trip Time et Timeout

RTTestimé = (1- α)*RTTestimé + α*EchantillonRTT

- l'influence des échantillons passés décroît exponentiellement
- valeur typique pour α : 0.125

RTT : exemple d'estimation



TCP: *Round Trip Time* et *Timeout*

Calcul du *timeout*

- **`RTTestimé`** plus une “marge de sécurité”
  - grande variation de **`RTTestimé`** **`->`** grande marge de sécurité
- estimer d'abord de combien dévie **`EchantillonRTT`** par rapport au **`RTTestimé`**:

`DevRTT = (1-β)*DevRTT + β*|EchantillonRTT - RTTestimé|`

`(typiquement, β = 0.25)`

L'intevalle de *timeout* :

`IntervalleTimeout = RTTestimé + 4*DevRTT`

TCP : transfert fiable de données

- TCP implémente un service de TFD au dessus du service non fiable de IP
- Pipelining des segments
- ACKs cumulatifs
- TCP utilise un seul minuteur pour les retransmissions
- Retransmissions declenchées par :
  - événement de minuterie
  - ACKs dupliqués
- Pour simplifier, l'émetteur TCP :
  - ignorer ACKs dupliqués
  - ignorer le contrôle de flux, le contrôle de congestion

TCP : événements de l'émetteur

données reçues depuis la couche appli. :

- Création segment avec un num. de séq.
- num. de séq. = numéro du premier octet du flux de données dans le segment
- démarrer le minuteur si ce n'est déjà fait (minuteur sur le dernier seg. non acquitté)
- intervalle de timeout : Variable **`IntervalleTimeOut`**

timeout :

- retransmettre le segment qui a causé le timeout
- redémarrer le minuteur

Ack réçu :

- Si acquittement reçu sur des segments non acquittés
  - prendre acte des acquittements
  - démarrer le minuteur si des segments restent en suspend

NumSeqSuiv = NumSeqInitial
DebutFenetre = NumSeqInitial

loop (toujours) {
switch(evenmt)

evenmt: *données reçus depuis application*
- créer segment TCP avec num. séq. NumSeqSuiv
- si (minuteur ne tourne pas)
  - démarrer minuteur
- passer le segment à IP
- NumSeqSuiv = NumSeqSuiv + longueur(données)

evenmt: *minuteur atteint le timeout*
- retransmettre le segment non encore acquitté et ayant le plus petit num. de séq.
- démarrer minuteur

evenmt: ACK reçu, avec le champ num. ACK = y
- si (y > DebutFenetre) {
  - DebutFenetre = y
  - si (il y a des segments non encore acquittés)
    - démarrer minuteur

}

} /* fin loop */

Émetteur TCP (simplifié)

Commentaire :

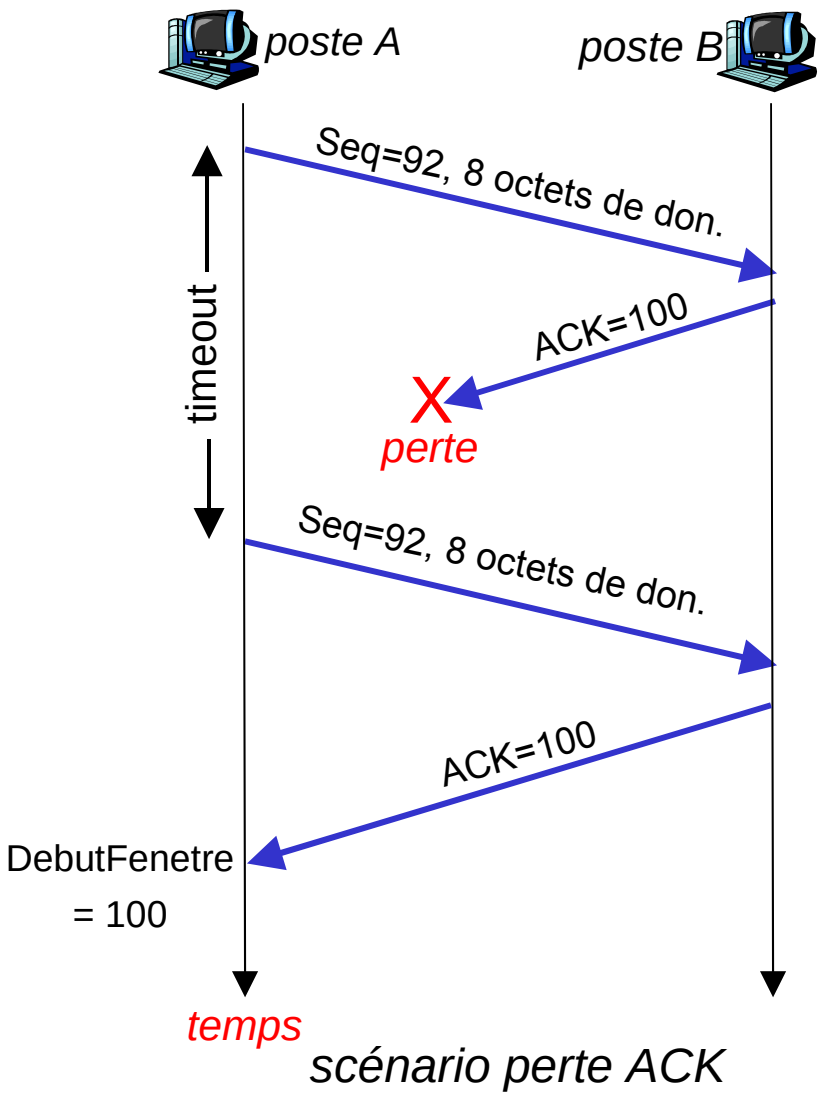
- DebutFenetre-1: dernier octet acquitté (cumulativement)

Exemple :

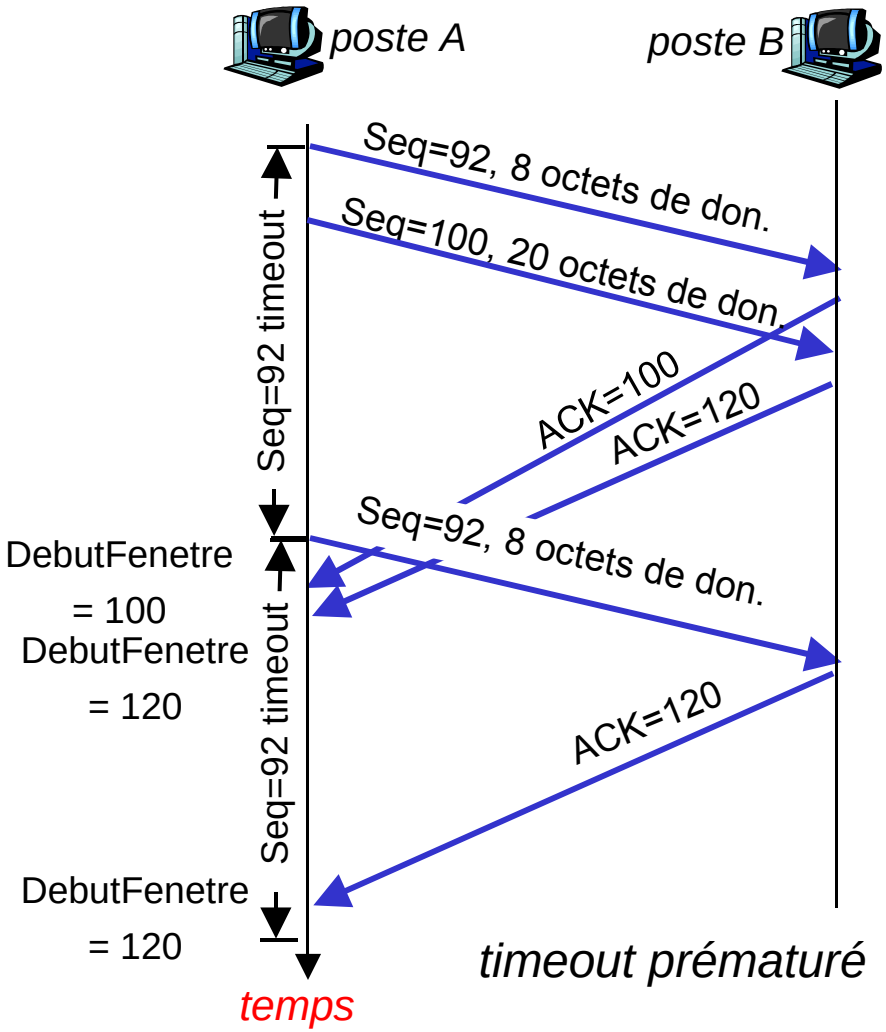
- DebutFenetre-1 = 71; y= 73, donc le récpt veut 73+ ;

y > DebutFenetre, donc une nouvelle donnée est acquittée

TCP: scénarios de retransmission

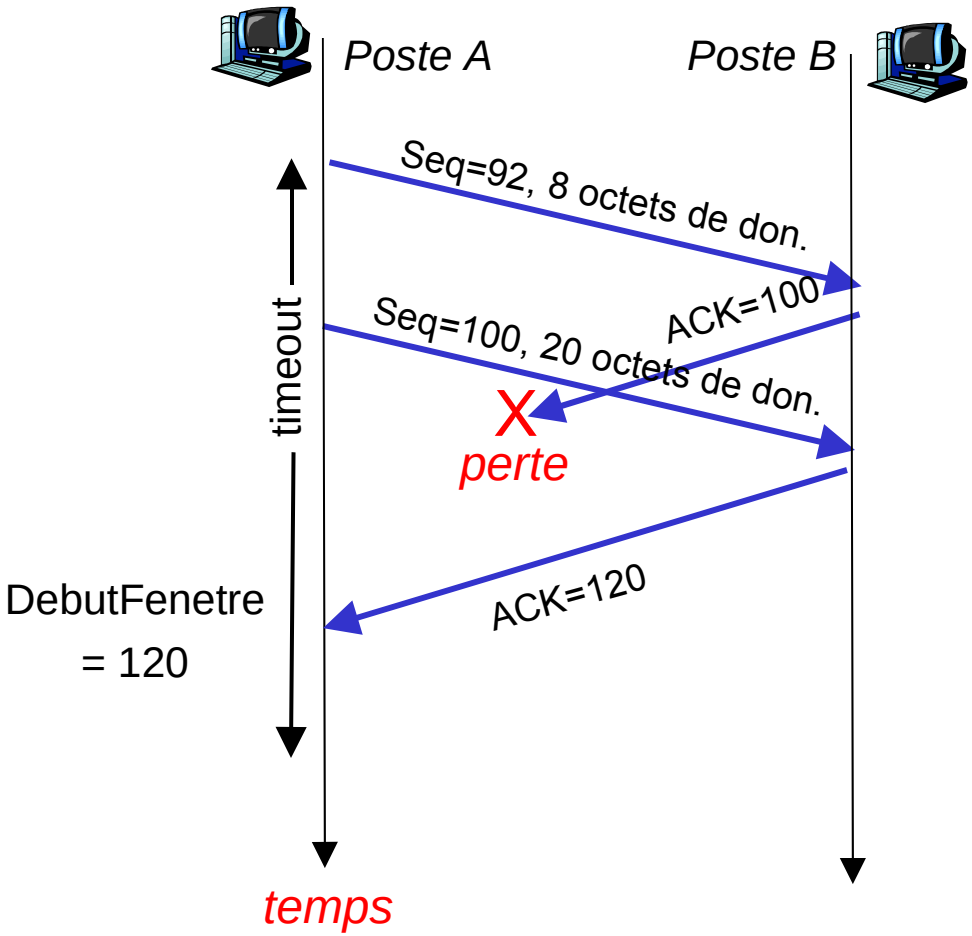


scénario perte ACK



timeout prématuré

TCP: scénarios de retransmission (suite)



scénario de ACK cumulatif

TCP : génération de ACK [RFC 1122, RFC 2581]

Evénmt récepteur	Action récepteur TCP
Arrivée d'1 seg. dans l'ordre avec num. séq. attendu. Ttes données jusqu'au num. séq. *ACKées*	Retarder le ACK: Attendre 500ms l'arrivée du prochain seg. Si pas de prochain seg., envoyer ACK
Arrivée d'1 seg. dans l'ordre avec num. séq. attendu. Un autre seg. a le ACK en suspens	Envoyer immédiatement un seul ACK cumulatif: *ACKer* plusieurs segments arrivés dans l'ordre
Arrivée d'un seg. hors séq. (plus grand que num. séq. attendu) Trou détecté	Envoyer immédiatement un ACK dupliqué indiquant le num. séq. du prochain octet attendu
Arrivée d'un segment remplissant partiellement ou totalement le trou	Envoyer immédiatement un ACK disant que le segment doit commencer à la borne inférieure du nouveau trou

Retransmission Rapide

- L'intervalle *timeout* souvent relativement long:
  - long retard avant retransmission du paquet perdu
- Détecter segments perdus via les ACKs dupliqués.
  - L'expéditeur envoie souvent beaucoup de segments en rafale
  - Si segment perdu, il y aura probablement beaucoup de ACKs dupliqués.

- Si l'ématteur reçoit 3 ACKs pour la même donnée, il suppose que le segment après les données *ACKées* est perdu:
  - retransmission rapide: renvoyer le segment avant expiration du minuteur

Algorithme de retransmission rapide:

```
evenmt: ACK reçu, avec le champ ACK qui vaut y
        if (y > DebutFenetre) {
            DebutFenetre = y
            if (il y a des segments non encore acquittés)
                démarrer le minuteur
        }
        else {
            incrémenter le compteur des ACKs dupliqués reçus pour y
            if (compteur des ACKs dupliqués reçus pour y = 3) {
                retransmette le segment avec y comme num. de séq.
            }
```

un ACK dupliqué pour un segment déjà acquitté

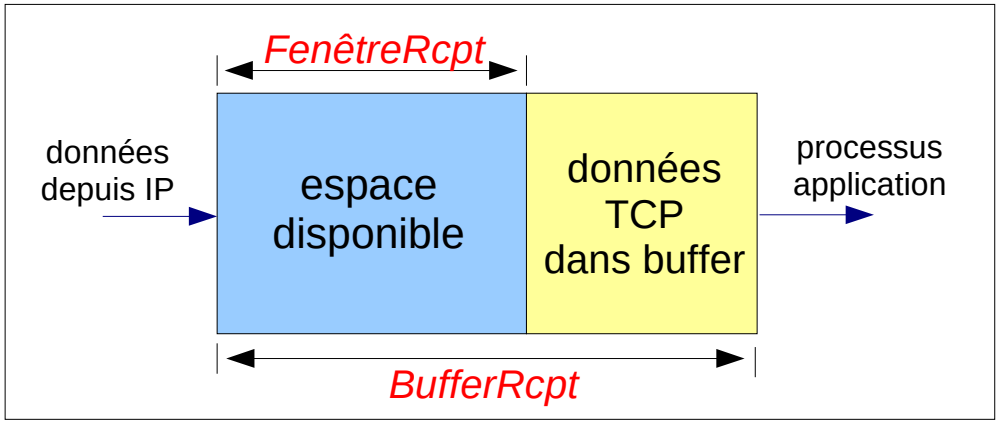
retransmission rapide

TCP : Contrôle de flux

contrôle de flux

l'émetteur ne doit pas faire déborder le buffer du récepteur trop rapidement

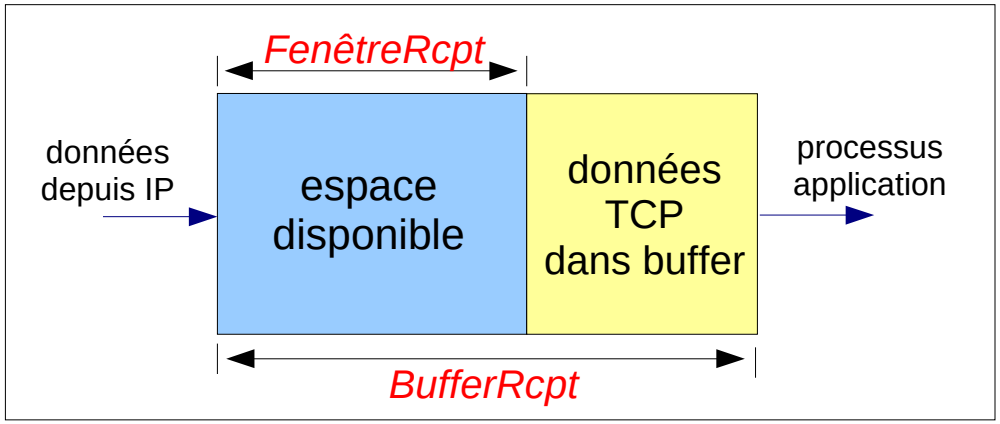
- Le côté récepteur d'une connexion TCP gère un buffer:



- les processus (côté applis) peuvent être lents à lire dans le buffer

- service <u>correspondance de vitesse</u> (*speed-matching*) : faire correspondre le taux d'envoi des segments avec le taux d'absorption des données par les applis

Contrôle de flux : comment ça marche?



(Supposons que le récepteur abandonne les segments arrivés hors séquence)

- espcae disponible dans le buffer

= FenêtreRcpt

= BufferRcpt – [DernierOctetReçu – DernierOctetLu]

- Le récepteur montre l'espace disponible en mettant la valeur de **FenêtreRcpt** dans les segments
- L'émetteur limite les données non acquittés à **FenêtreRcpt**
  - garantie du non débordement du buffer du récepteur

TCP : Gestion de la connexion

Rappel : L'émetteur et le récepteur TCP établissent une “connexion” avant d'échanger les segments de données

- initialisation des variables TCP:
  - numéros de séquence
  - buffers, infos de contrôle de flux (ex. **`FenêtreRcpt`**)
- *client :* l'initiateur de la connexion
  - JAVA : `Socket Soc_Clt = new Socket("poste","port");`
  - C : `Soc_Clt = socket(AF_INET,SOCK_STREAM,0))`
- *serveur :* contacté par le client
  - JAVA : `Socket SCon = SocketEcoute.accept();`
  - C : `SCon = accept(SocketEcoute,(struct sockaddr*) &clt_skaddr, &addrlen))`

Connexion en trois étapes

Etape1: le poste client envoi au serveur le segment SYN TCP

- specifie le numéro de séquence initial
- pas de données

Etape2: le poste serveur reçoit SYN, répond avec le segment SYNACK

- le serveur alloue les buffers
- le serveur specifie son numéro de séquence initial

Etape3: le client reçoit SYNACK, répond avec le segment ACK, qui peut contenir des données

TCP : Gestion de connexion (suite)

Fin de connexion:

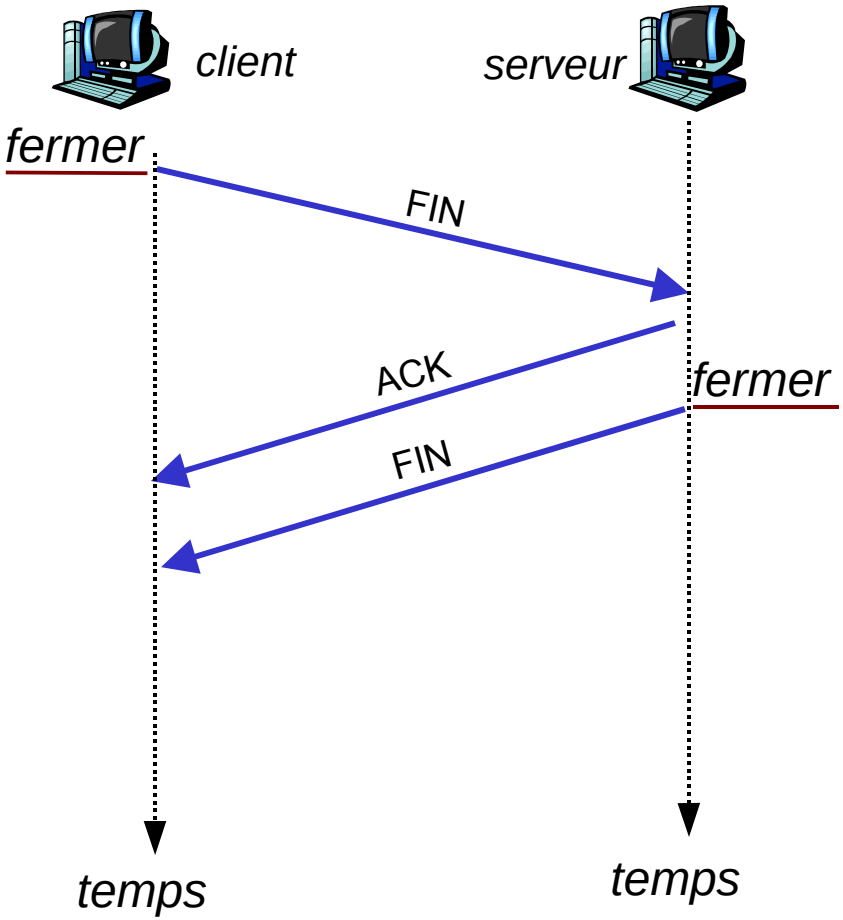
le client ferme la socket:

Java : **`SokClt.close();`**

C : **`close(SokClt);`**

Etape 1: client envoi au serveur le segment de contrôle TCP FIN

Etape 2: serveur reçoit FIN, répond avec ACK, ferme la connexion et envoi FIN.

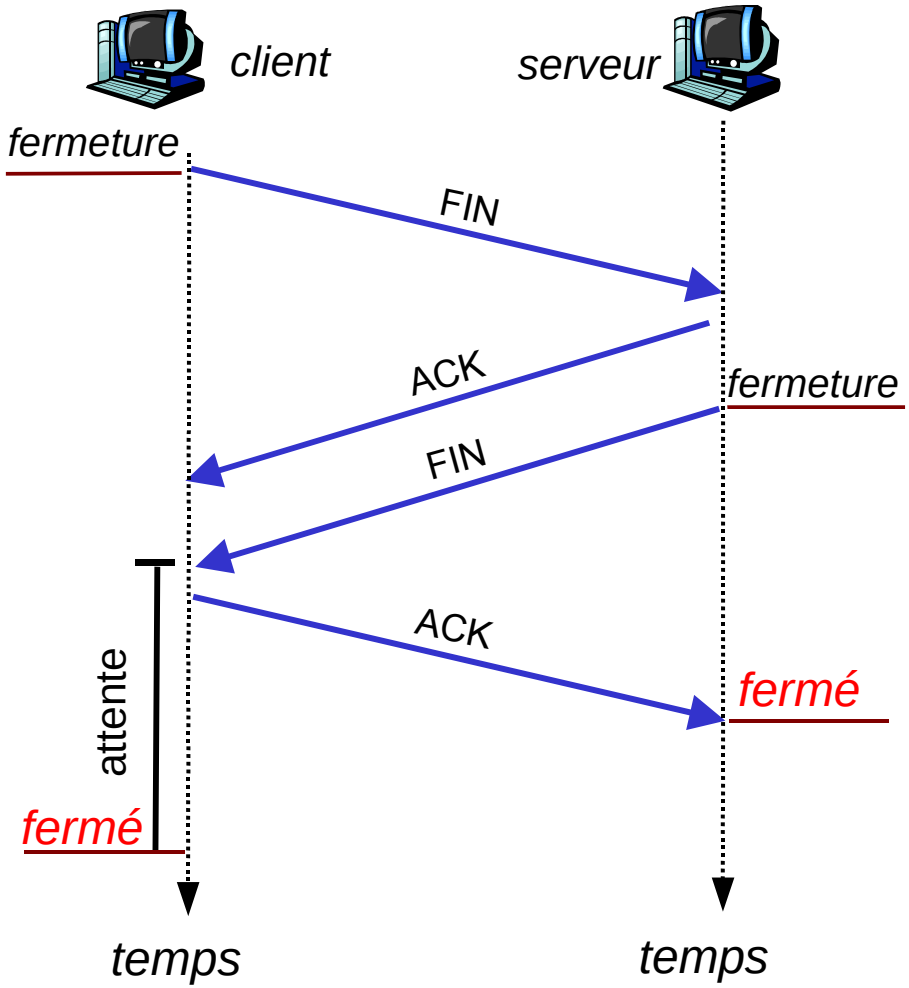


TCP : Gestion de connexion (suite)

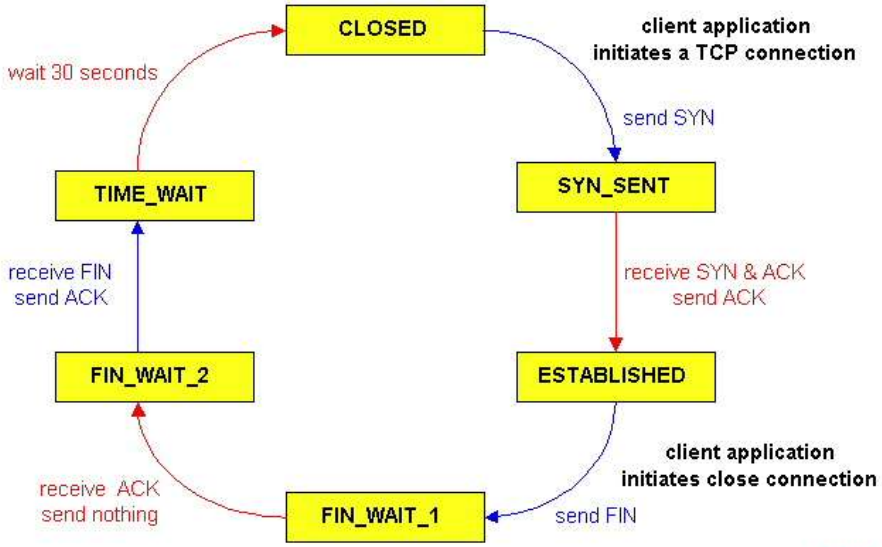
Etape 3: client reçoit FIN, répond avec ACK.

- Entre dans une attente temporelle – au cas où le ACK se perde : *Connexion fermée*

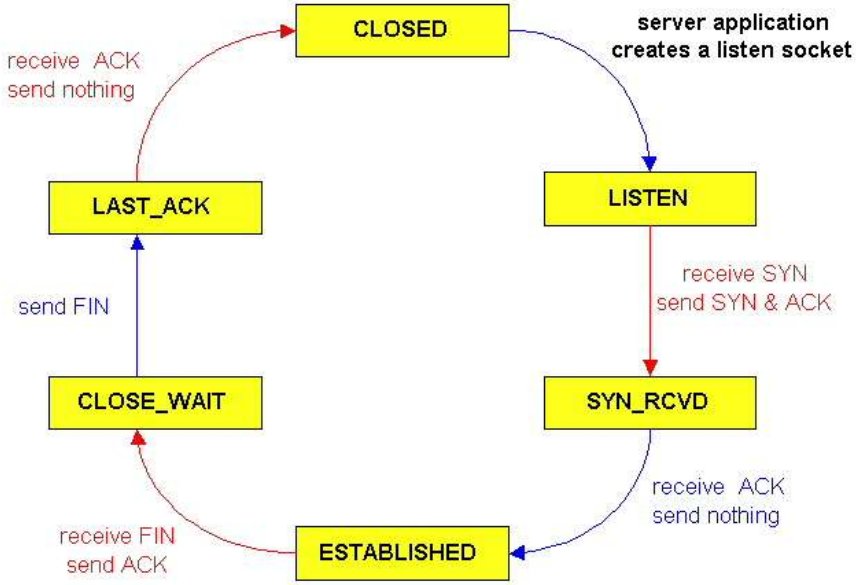
Etape 4: serveur, reçoit ACK : *Connexion fermée.*



TCP : Gestion de connexion (suite)



cycle de vie du client TCP



cycle de vie du serveur TCP

Contrôle de congestion dans TCP

- contrôle end-end (pas d'assistance réseau)
- l'émetteur limite la transmission:

  LastByteSent-LastByteAcked

  $\leq$ **CongWin**

- Autrement dit :

$$\textbf{taux} = \frac{\textbf{CongWin}}{\textbf{RTT}} \textit{ octets/sec}$$

- **CongWin** est dynamique, fonction de la percéption de la congestion de réseau

Comment l'émetteur pérçoit la congestion?

- évnmt de perte = *timeout ou* 3 acks dupliqués
- émetteur TCP réduit le taux (donc **CongWin**) après une perte

Trois mécanismes:

- AIMD
- démarrage lent (*slow start*)
- conservateur après les *timeouts*

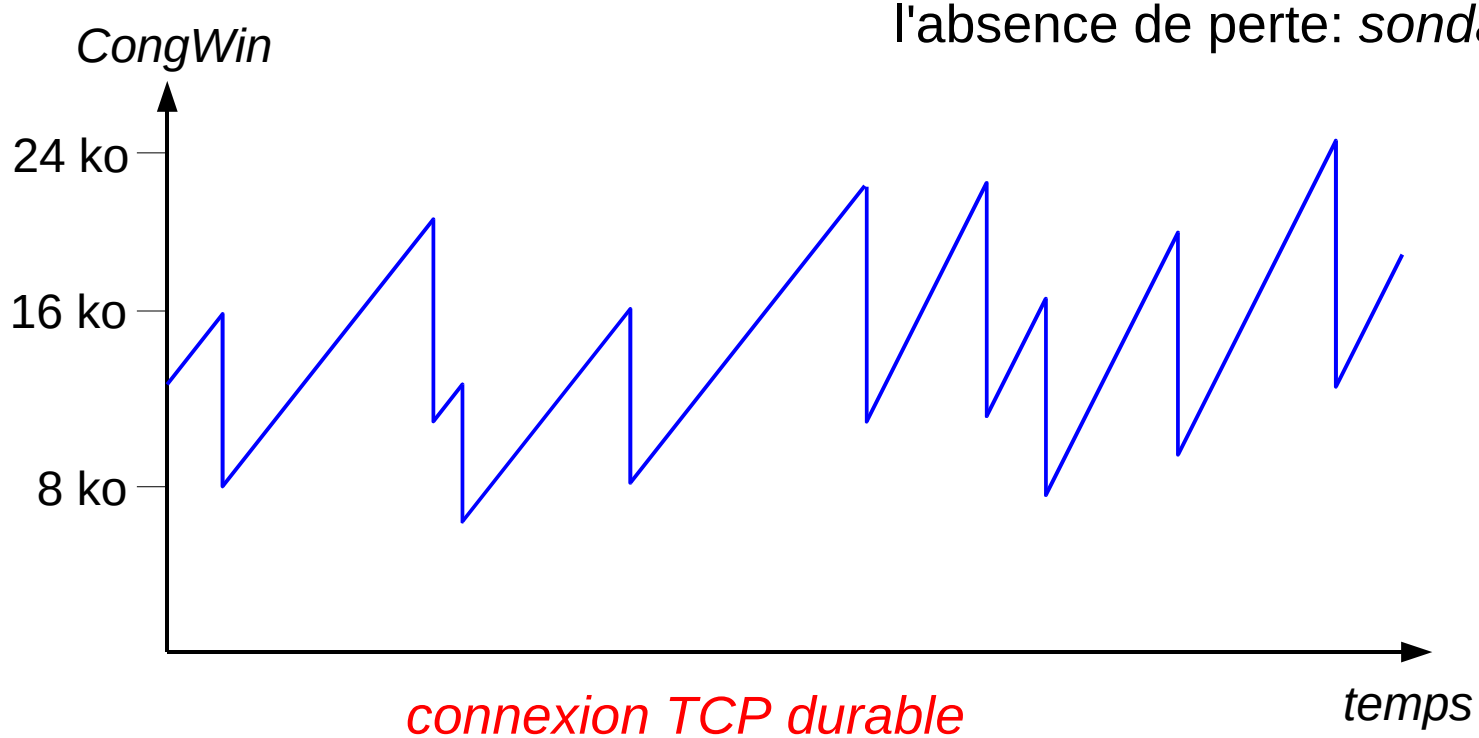
TCP : AIMD

diminution multiplicative :

diviser **`CongWin`** par 2 à chaque perte

augmentation additive :

augmenter **`CongWin`** d'1 MSS (*Maximum Segment Size*) chaque RTT en l'absence de perte: *sondage*



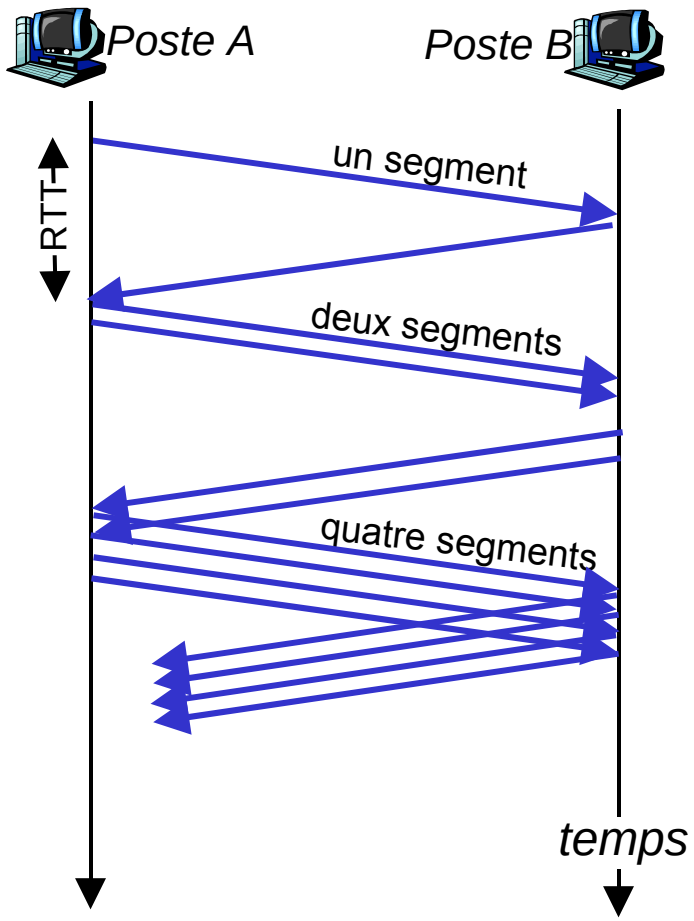
connexion TCP durable

TCP : Démarrage lent

- Au début de la connexion : **`CongWin`** = 1 MSS
  - Exemple: MSS = 500 bytes & RTT = 200 msec
  - taux initial = 20 kbps
- la bande passante disponible peut être >> MSS/RTT
  - en souhaitant atteindre rapidement un taux respectable
- Au début de la connexion :
  - augmenter exponentiellement le taux (de transfert) jusqu'au premier événement de perte

TCP : Démarrage lent (suite)

- Au début de la connexion: augmenter exponentiellement le taux (de transfert) jusqu'au premier événement de perte:
  - doubler **`CongWin`** à chaque RTT : à chaque réception d'un ACK
- En résumé : le taux initial est faible mais croit exponentiellement



TCP : Démarrage lent (raffinement)

- Après 3 ACKs dup. :
  - **`CongWin`** est coupée en 2
  - puis la fenêtre grandi linéairement

mais

- Après un *timeout* :
  - **`CongWin`** remise à 1 MSS;
  - puis la fenêtre grandi exponentiellement
  - jusqu'au à un seuil, puis grandi linéairement

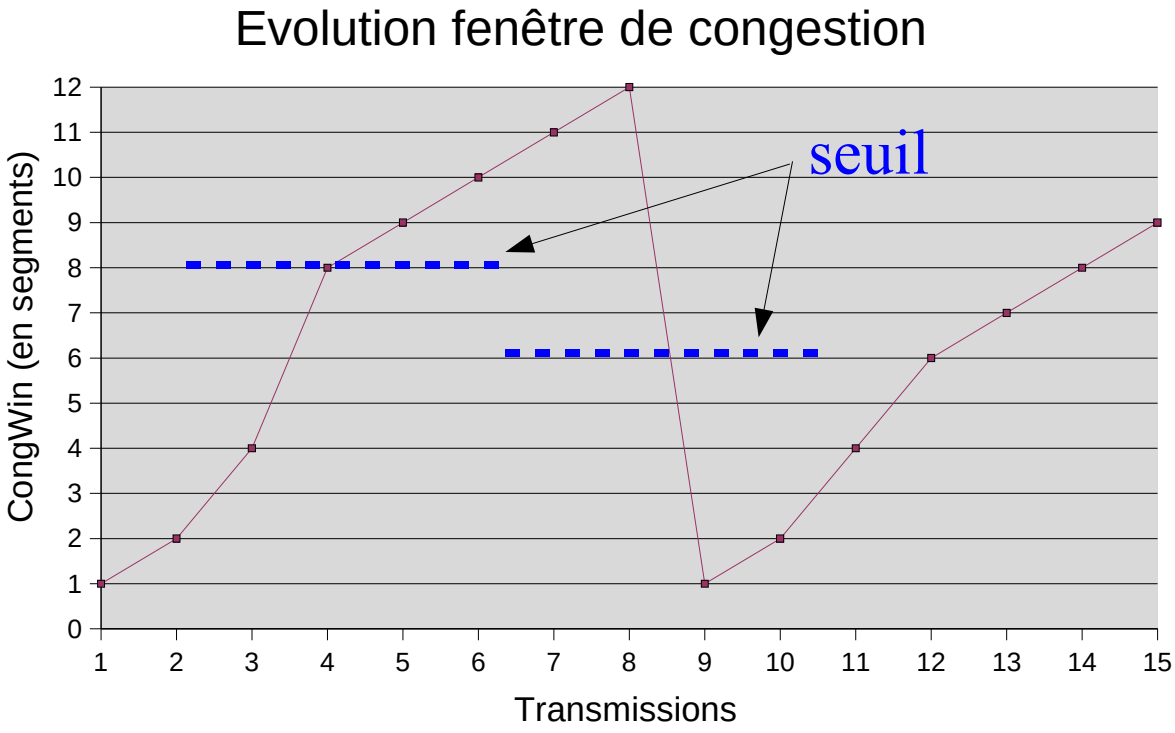
Philosophie:

- 3 ACKs dup. indiquent que le réseau est capable d'acheminer quelques segments
- *timeout* avant les 3 ACKs dup. est “plus alarmant”

TCP : Démarrage lent (raffinement)

Q: Quand est ce que la croissance exponentielle devient linéaire?

R: Quand **`CongWin`** devient la moitié de sa valeur avant le *timeout*.



Implémentation:

- Variable **`Seuil`**
- **si** événement perte **alors**

  `Seuil` := (**`CongWin`** juste avant l'événement perte) / 2

TCP : Contrôle de congestion – résumé

- Quand **`CongWin`** < **`Seuil`**, émetteur dans sa phase démarrage lent, fenêtre grandi exponentiellement.
- Quand **`CongWin`** > **`Seuil`**, émetteur dans la phase congestion-évitable, fenêtre grandi linéairement.
- Quand un triple ACK dupliqué se produit, **`Seuil`** mis à **`CongWin/2`** et **`CongWin`** mise à **`Seuil`**.
- Quand timeout se produit, **`Seuil`** mis à **`CongWin/2`** et **`CongWin`** mise à 1 MSS.

La suite ?

Quitter la vision « bord » du réseau (couches application et transport) pour aller explorer son « coeur » :

Couche réseau