Chapitre 3 Couche Transport

1

Chapitre 3: Couche Transport

But:

- Comprendre les principes derrière les services de la couche transport :
  - multiplexage, démultiplexage
  - transfert de données fiable
  - contrôle de flux
  - contrôle de congestion
- En savoir plus sur les protocoles de couche de transport Internet :
  - UDP : transport sans connexion
  - TCP : transport fiable orienté connexion Contrôle d'encombrement TCP

2

Chapitre 3: Plan

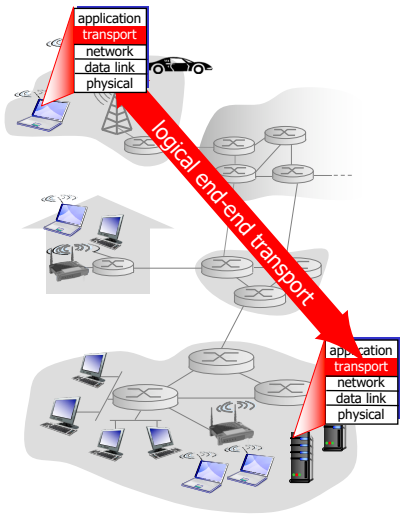
3.1 Services de la couche transport
3.2 Multiplexage et démultiplexage
3.3 Transport sans connexion: UDP
3.4 Principes de transfert de données fiable
3.5 Transport orienté connexion: TCP
- structure de segment
- transfert de données fiable
- contrôle de flux
- gestion des connexions

3.6 Principes du contrôle de la congestion
3.7 Contrôle de congestion TCP

3

Services de transport et protocoles

- fournir une communication logique entre les processus d'application exécutés sur des hôtes différents
- les protocoles de transport s'exécutent dans les systèmes d'extrémité
  - côté expéditeur: sépare les messages d'application en segments, passe à la couche réseau
  - côté récepteur : réassemble les segments en messages, passe à la couche application
- plusieurs protocoles de transport disponibles pour les applications
  - Internet: TCP et UDP



4

Couche Transport vs. Réseau

- *Couche reseau :* communication logique entre les hôtes
- *Couche transport :* communication logique entre les processus
  - s'appuie sur, et améliore, les services de couche réseau

Analogie du ménage :

12 enfants de la maison d'Alice envoient des lettres aux 12 enfants dans la maison de Bob:

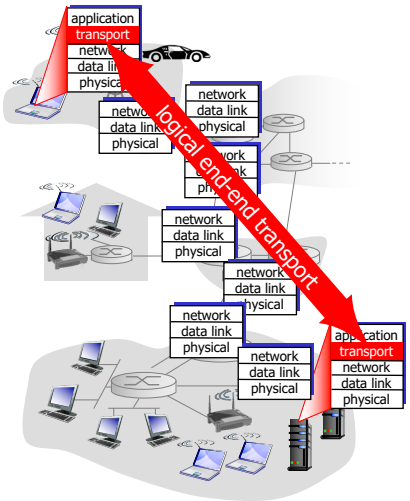
- hôtes = *maisons*
- processus = *enfants*
- messages app = *lettres dans enveloppes*
- protocole couche transport = *Alice et Bob*
- protocole couche réseau = *le service postal*

Transport Layer 3-5

5

Protocoles transport sur Internet

- livraison fiable (TCP)
  - contrôle de la congestion
  - contrôle de flux
  - configuration de la connexion
- livraison non fiable et non ordonnée : UDP
  - extension rapide de service IP «best-effort»
- services non disponibles :
  - garanties de délai
  - garanties de bande passante



Transport Layer 3-6

6

Chapitre 3: Plan

3.1 Services de la couche transport

3.2 Multiplexage et démultiplexage

3.3 Transport sans connexion : UDP

3.4 Principes de transfert de données fiable

3.5 Transport orienté connexion : TCP
- structure de segment
- transfert de données fiable
- contrôle de flux
- gestion des connexions

3.6 Principes du contrôle de la congestion

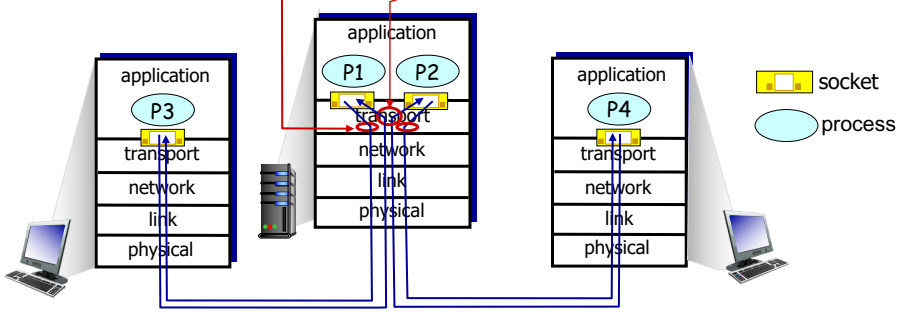
3.7 Contrôle de congestion TCP

7

Multiplexage/demultiplexage

Multiplexage à l'expéditeur:
gérer les données de plusieurs sockets, ajouter un en-tête de transport (utilisé par la suite pour le démultiplexage)

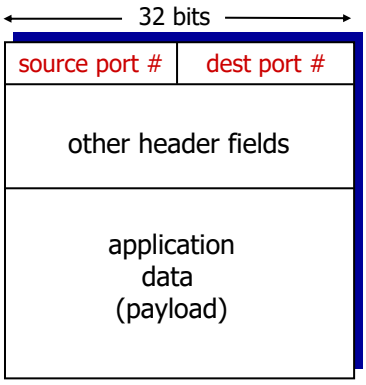
Demultiplexage au récepteur:
utiliser les informations d'en-tête pour diffuser des segments reçus à la bonne destination(socket)



8

Comment fonctionne le démultiplexage

- Hôte reçoit datagrammes IP
  - chaque datagramme a @IP source, @IP de destination
  - chaque datagramme porte un segment de couche transport
  - chaque segment a son port source, son port destination
- Hôte utilise *addresses IP & no. port* pour diriger le segment au socket approprié



Format du segment TCP/UDP

Transport Layer 3-9

9

Démultiplexage sans connexion

- *Prog:* création socket avec no. port sur le hôte local :

```
DatagramSocket mySocket1
= new DatagramSocket(12534);
```

- *Info:* lors de la création d'un datagramme pour envoyer dans le socket UDP, on doit spécifier
  - adresse IP de destination
  - no. port de destination

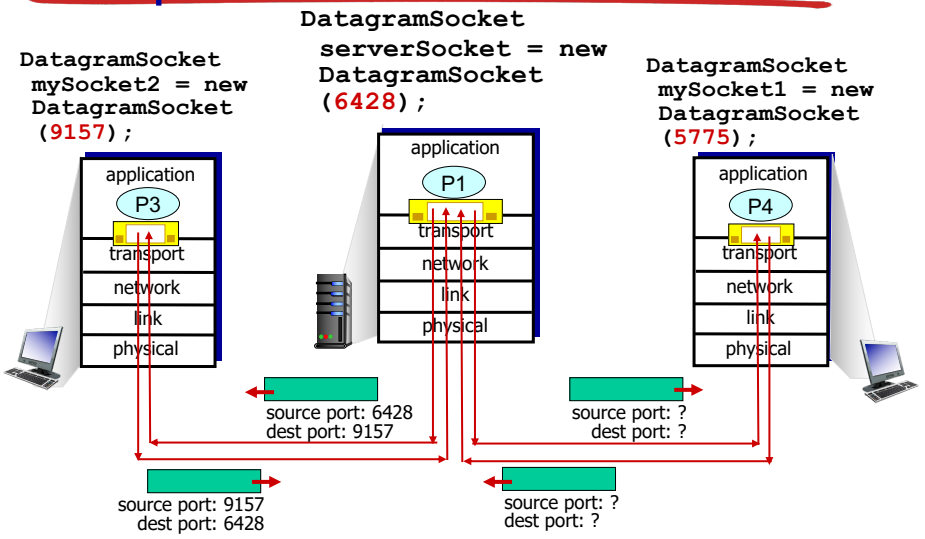
- quand l'hôte reçoit le segment UDP :
  - vérifie le numéro de port de destination dans le segment
  - dirige le segment UDP vers socket avec ce port

➡ datagrams IP avec *même port dest,* mais different source IP et/ou port source vont être dirigés vers le *même socket* à la destination

Transport Layer 3-10

10

Démultiplexage sans connexion : example



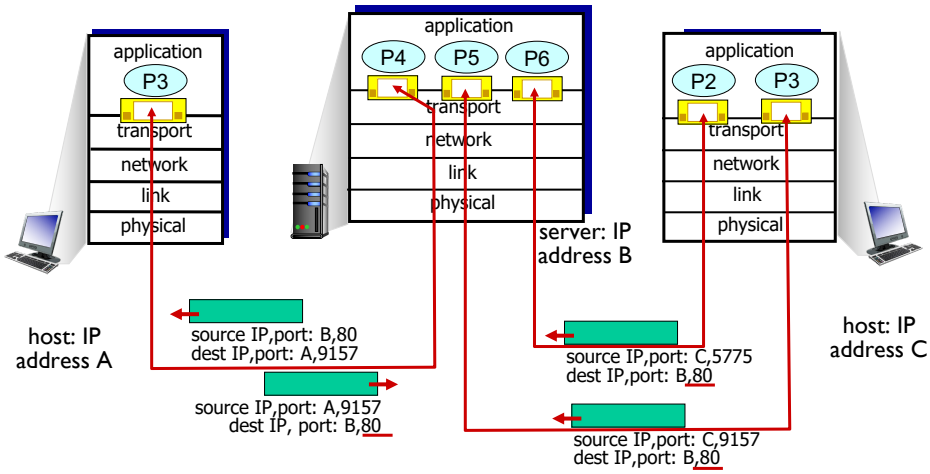
11

Démultiplexage avec connexion

- TCP socket identifié par 4-tuples :
  - IP source
  - *port source*
  - IP dest
  - *port dest*
- Demux : le récepteur utilise les quatre valeurs pour diriger le segment vers le socket appropriée
- serveur hôte peut prendre en charge plusieurs sockets TCP simultanés :
  - chaque socket identifié par son propre 4-tuples
- les serveurs Web ont des sockets différents pour chaque client connecté
  - HTTP non persistant aura un socket différent pour chaque requête

12

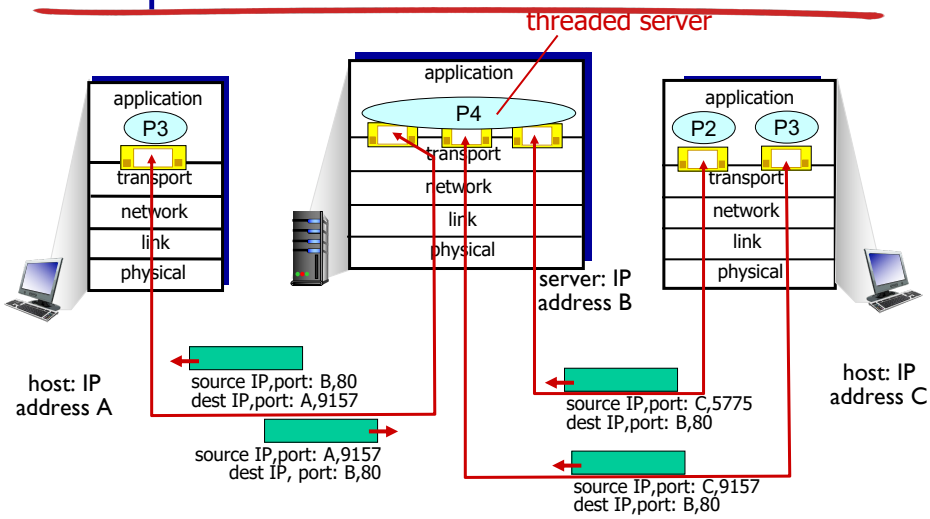
Démultiplexage avec connexion : example



three segments, all destined to IP address: B, dest port: 80 are demultiplexed to *different* sockets

13

Démultiplexage avec connexion : example



14

Chapitre 3: Plan

3.1 Services de la couche transport
3.2 Multiplexage et démultiplexage
3.3 Transport sans connexion: UDP
3.4 Principes de transfert de données fiable
3.5 Transport orienté connexion : TCP
- structure de segment
- transfert de données fiable
- contrôle de flux
- gestion des connexions

3.6 Principes du contrôle de la congestion
3.7 Contrôle de congestion TCP

15

UDP: User Datagram Protocol [RFC 768]

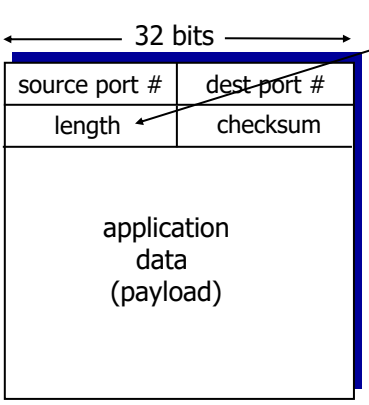
- Protocole transport pour Internet :"pas compliqué" et "minimaliste"
- Service "best effort", segment UDP peut :
  - Être perdu
  - Arriver en désordre au niveau application
- *Sans connexion :*
  - Pas de handshaking entre expéditeur et récepteur, pas de délai en plus
  - Chaque segment UDP traité indépendamment des autres
- UDP utilisé dans :
  - Application temps-réels streaming multimédia (tolérant au perte, sensible à variation débit)
  - DNS
  - SNMP
- Transfert fiable sur UDP :
  - Ajouter la fiabilité dans la couche application
  - récupération d'erreur spécifique à l'application !

16

Internet apps: application, protocole transport

Application	Application-Layer Protocol	Underlying Transport Protocol
Electronic mail	SMTP	TCP
Remote terminal access	Telnet	TCP
Web	HTTP	TCP
File transfer	FTP	TCP
Remote file server	NFS	Typically UDP
Streaming multimedia	typically proprietary	UDP or TCP
Internet telephony	typically proprietary	UDP or TCP
Network management	SNMP	Typically UDP
Routing protocol	RIP	Typically UDP
Name translation	DNS	Typically UDP

UDP : en tête



Format segment UDP

longueur, en **octets** du segment UDP, en-tête compris

Pourquoi y a-t-il UDP ?

- pas d'établissement de connexion (ce qui peut ajouter du délai)
- simple : pas d'état de connexion à l'expéditeur, au destinataire
- petite taille d'en-tête
- pas de contrôle de la congestion : UDP peut aller aussi vite que l'on veut

Checksum (somme de contrôle) UDP

But: détecter les «erreurs» (par exemple, les bits inversés) dans le segment transmis

Expéditeur :
- traiter le contenu du segment, y compris les champs d'en-tête, comme une séquence d'entiers de 16 bits
- somme de contrôle : ajout (complément de somme) au contenu du segment
- l'expéditeur place la valeur de somme de contrôle dans le champ *checksum* dans UDP

Récepteur :
- calculer le somme de contrôle du segment reçu
- vérifier si la somme de contrôle calculée est égale à la valeur du champ :
  - NON – erreur détectée
  - OUI – pas d'erreur détectée. *Mais peut-être il y en a quand même ?* Plus de détail plus tard ….

19

Internet checksum : exemple

Exemple : addition de deux mots de 16-bit

```
                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
              ---------------------------------
wraparound    1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
              ---------------------------------
sum             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
```

Remarque : lors de l'addition de numéros, un résultat du bit le plus significatif doit être ajouté au résultat.

20

TD3 : exo 1

21

Chapitre 3 : Plan

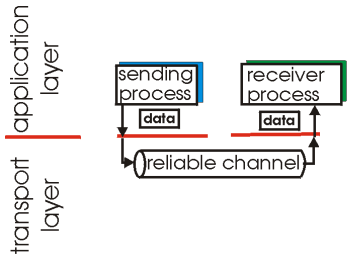
3.1 Services de la couche transport
3.2 Multiplexage et démultiplexage
3.3 Transport sans connexion : UDP
3.4 Principes de transfert de données fiable*
3.5 Transport orienté connexion : TCP
- structure de segment
- transfert de données fiable
- contrôle de flux
- gestion des connexions

3.6 Principes du contrôle de la congestion
3.7 Contrôle de congestion TCP

22

Principes de transfert fiable

- Important dans les couches application, transport, liaison
  - top-10 des thèmes importants en réseau!

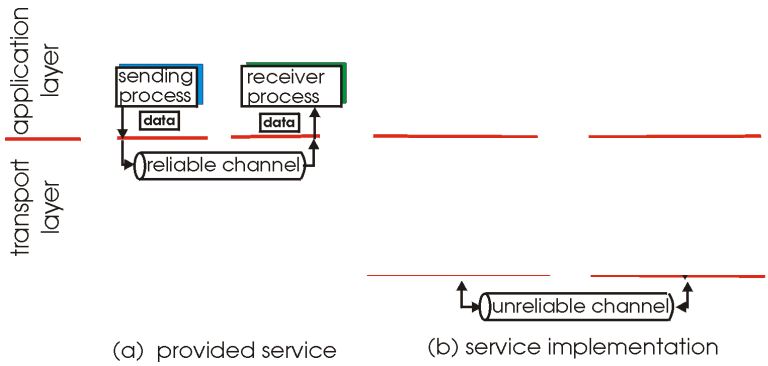


(a) provided service

- Les caractéristiques d'un canal non fiable détermineront la complexité d'un protocole de transfert fiable (reliable data transfer - rdt)

Principes de transfert fiable

- Important dans les couches application, transport, liaison
  - top-10 des thèmes importants en réseau!

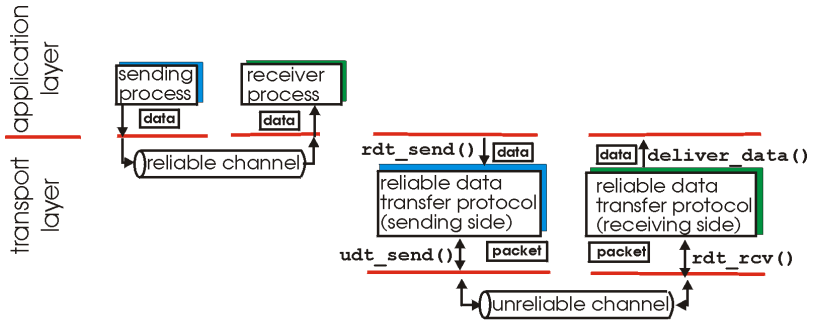


(a) provided service (b) service implementation

- Les caractéristiques d'un canal non fiable détermineront la complexité d'un protocole de transfert fiable (rdt)

Principes de transfert fiable

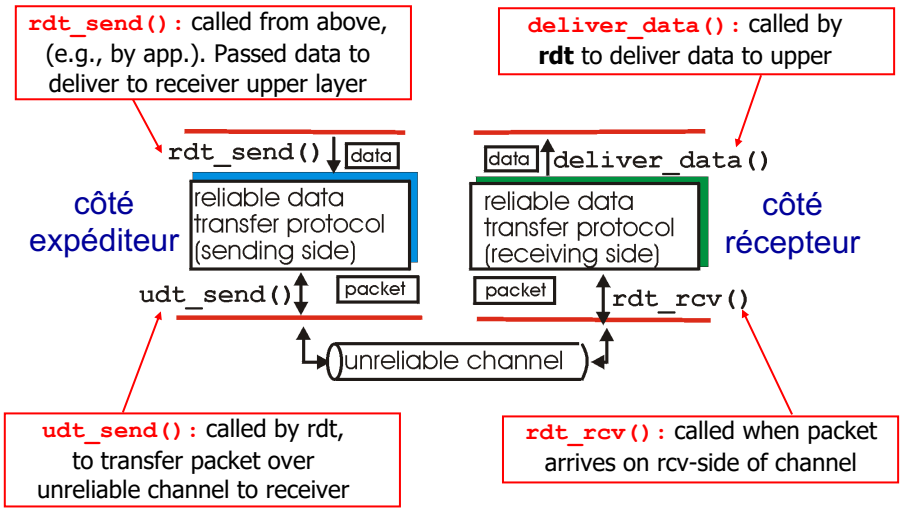
- Important dans les couches application, transport, liaison
  - top-10 des thèmes importants en réseau!



(a) provided service (b) service implementation

- Les caractéristiques d'un canal non fiable détermineront la complexité d'un protocole de transfert fiable (rdt)

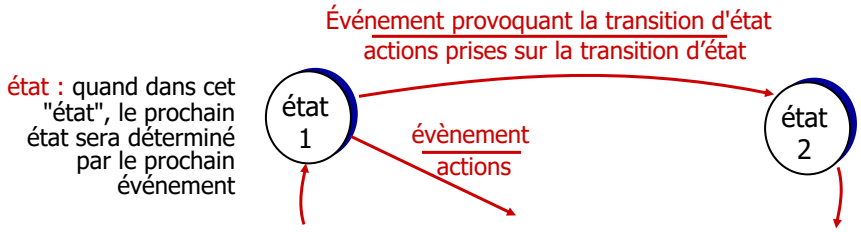
Transfert fiable : c'est parti!



Transfert fiable : c'est parti!

On va commencer à :

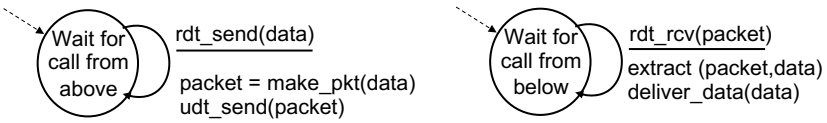
- développer de manière incrémentale un protocole de transfert fiable de données (rdt) côté expéditeur et récepteur en considérant uniquement le transfert de données unidirectionnel
- utiliser des machines à états finis (Finite State Machine-FSM) pour spécifier l'expéditeur et le récepteur



27

rdt 1.0 : transfert fiable sur un canal fiable

- canal sous-jacent parfaitement fiable
  - pas d'erreur de bits
  - pas de perte de paquets
- FSM séparés pour l'expéditeur et le récepteur :
  - l'expéditeur envoie des données dans le canal sous-jacent
  - le récepteur lit les données depuis le canal sous-jacent



expéditeur récepteur

28

rdt2.0: canal fiable avec des erreurs

- canal sous-jacent peut inverser des bits dans le paquet
  - somme de contrôle pour détecter les erreurs de bits
- *la question : comment réparer des erreurs?*

Comment les humains réparent-ils des «erreurs» pendant la conversation?

29

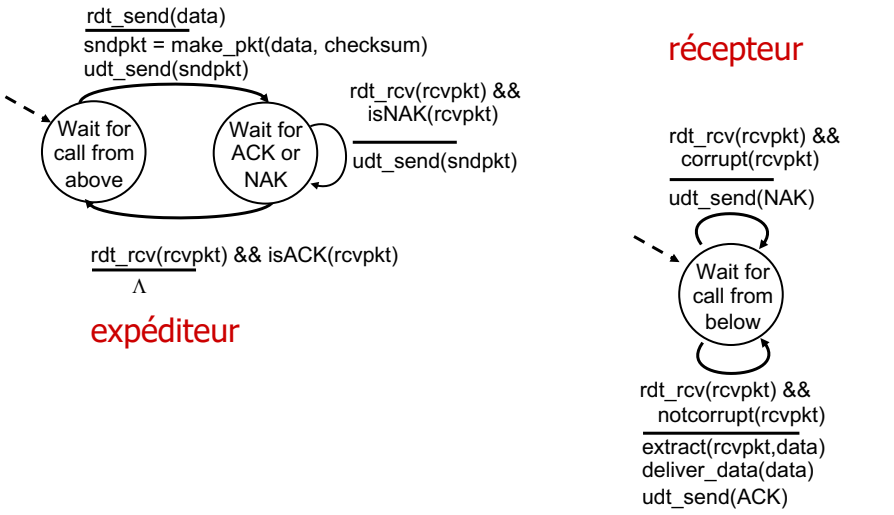
rdt2.0: canal avec des erreurs

- canal sous-jacent peut inverser des bits dans le paquet
  - somme de contrôle pour détecter les erreurs de bits
- question: comment réparer des erreurs?
  - *accusés de réception (ACK)*: le récepteur indique explicitement à l'expéditeur que pkt est reçu OK
  - *accusés de réception négatifs (NAK)*: le récepteur indique explicitement à l'expéditeur que pkt a des erreurs
  - expéditeur retransmet pkt à la réception de NAK
- nouveaux mécanismes dans **`rdt2.0`** (au-delà **`rdt1.0`**):
  - détection d'erreur
  - Retour : msgs contrôle (ACK,NAK) depuis le récepteur à l'expéditeur

Transport Layer 3-30

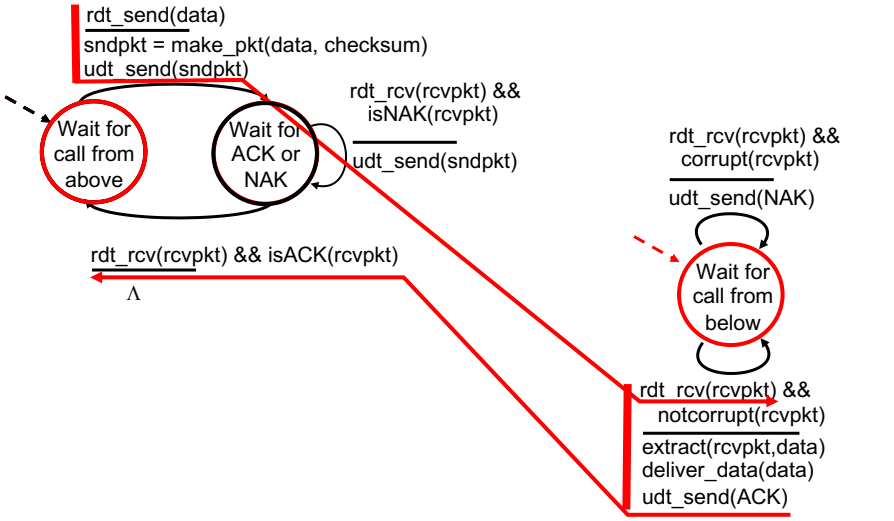
30

rdt2.0: specification FSM



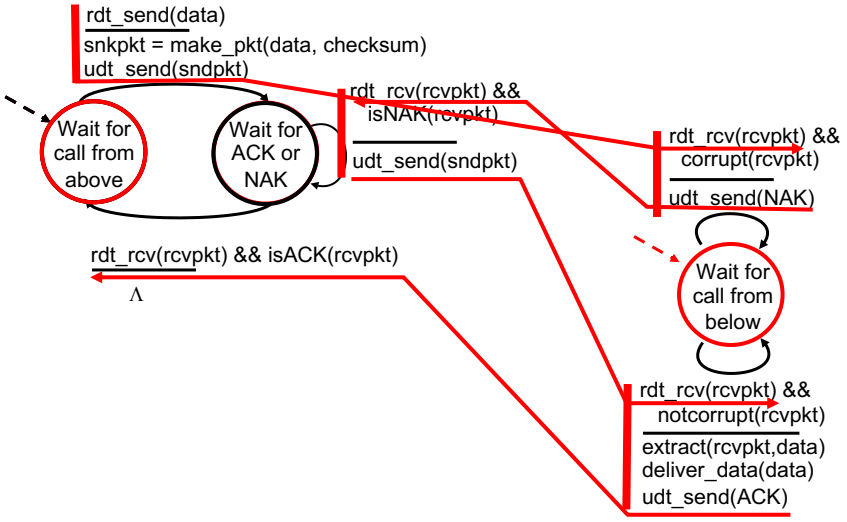
31

rdt2.0: operation sans erreurs



32

rdt2.0: scenario avec erreur



rdt2.0 a un défaut fatal !

Que se passe-t-il si ACK/NAK est corrompu?

- expéditeur ne sait pas ce qui s'est passé chez le récepteur !
- ne peut pas simplement retransmettre : duplicata possible

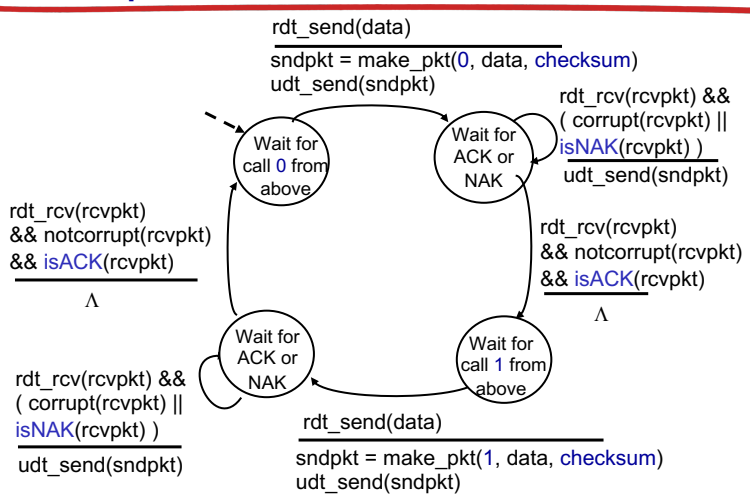
Traitement des doublons :

- expéditeur retransmet le pkt courant si ACK/NAK corrompu
- expéditeur ajoute *le numéro de séquence* à chaque paquet
- récepteur ne délivre pas (jette) le pkt dupliqué

stop and wait

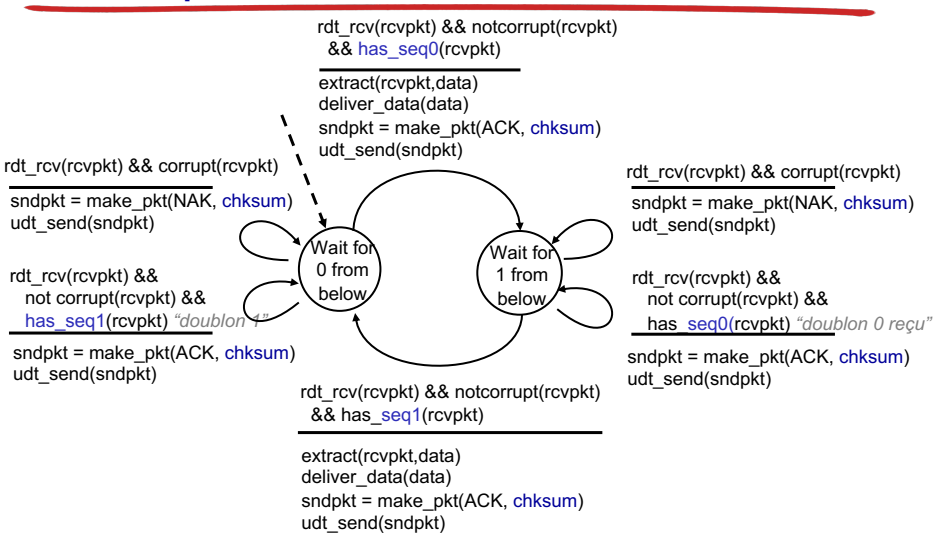
expéditeur envoie un paquet, puis attend la réponse du récepteur

rdt2.1: expéditeur qui gère les ACK/NAKs corrompus



35

rdt2.1: récepteur qui gère les ACK/NAKs corrompus



36

rdt2.1: discussion

Expéditeur :

- no. seq ajouté au pkt
- 2 no. seq. (0,1) vont suffir. Pourquoi?
- doit vérifier si ACK/ NAK reçu est corrompu
- deux fois plus d'états
  - état doit "mémoriser" si pkt "attendu" a no. seq de 0 ou 1

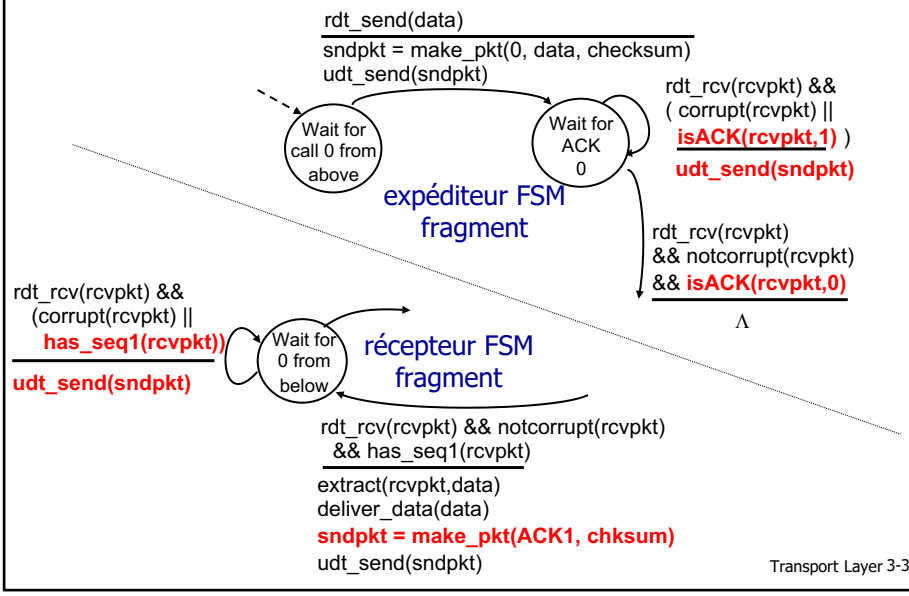
Récepteur :

- doit vérifier si le paquet reçu est reçu en double
  - état indique si 0 ou 1 est attendu pour le no. de seq

rdt2.2 : protocole sans NAK

- Même fonctionnalité que rdt2.1, en utilisant uniquement les ACK
- au lieu de NAK, le récepteur envoie un ACK pour le dernier pkt reçu OK
  - le récepteur doit explicitement inclure le nombre de pkt acquittés (ACKed)
- ACK en double à l'expéditeur entraîne la même action que NAK : *retransmission du pkt actuel*

rdt2.2 : fragments expéditeur/récepteur



39

TD3 : exo 2

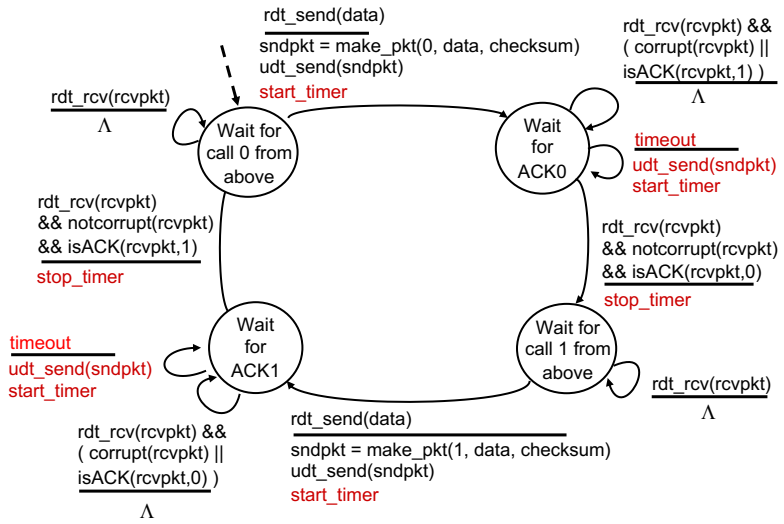
40

rdt3.0: canaux avec erreurs *et* pertes

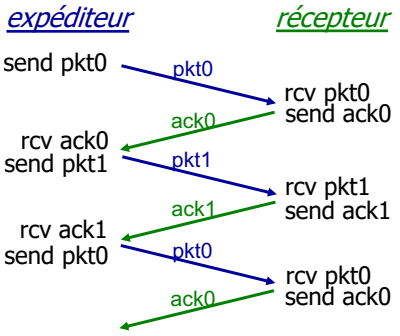
Hypothèse : canal sous-jacent peut perdre des paquets (data, ACKs)
- checksum, no. seq., ACKs, retransmissions vont aider … mais ce n'est pas suffisant

Approche : expéditeur attend un ACK avec un délai "raisonable"
- Retransmets si pas de ACK reçu durant ce temps
- Si pkt (ou ACK) juste en retard (pas de perte) :
  - retransmission va dupliquer, mais no. seq. va traiter ca
  - récepteur doit spécifier no. seq. du pkt ACKed
  - Besoin d'un compteur

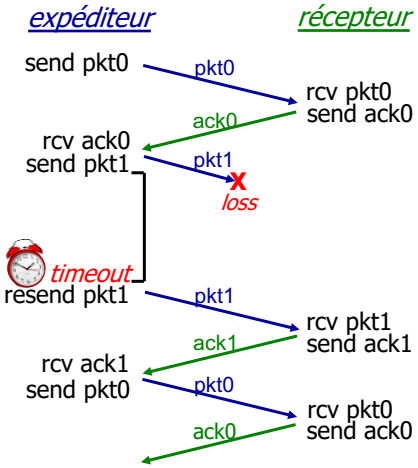
rdt3.0 expéditeur



rdt3.0 en action

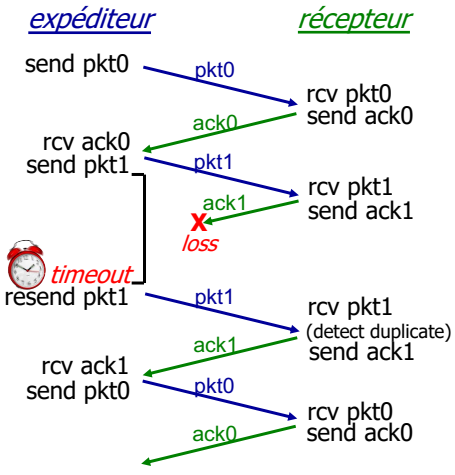


(a) Sans perte

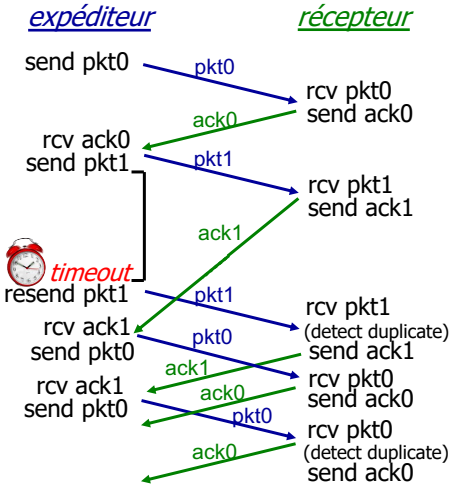


(b) Avec perte du pkt

rdt3.0 in action



(c) ACK perdu



(d) Timeout prématuré/ACK retardé

Performance de rdt3.0

- rdt3.0 est correct, mais donne mauvais performance
- exp: lien 1 Gbps, délai prop. 15 ms, pkt 8000 bit :

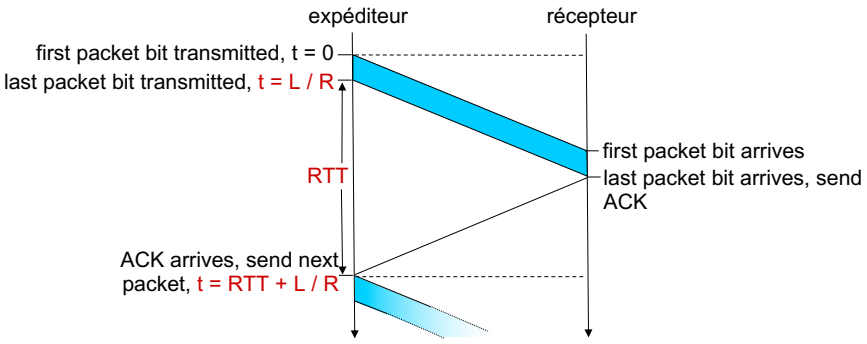
$$D_{trans} = \frac{L}{R} = \frac{8000\ bits}{10^9\ bits/sec} = 8\ microsecs$$

  - U $_{expéditeur}$: *utilisation* = fraction de temps expéditeur occpué à envoyer

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

  - Si RTT=30 msec, pkt 1Ko tous les 30 msec: 267kbp (débit sur le lien 1 Gbps)
- Protocole réseau limite l'utilisation de ressource de la couche physique !

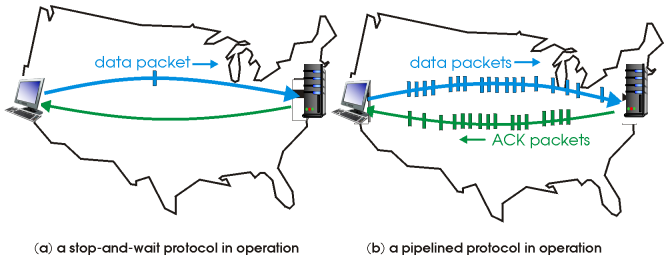
rdt3.0 : operation stop-and-wait



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

Protocoles *Pipeline*

pipelining: expéditeur permet multiple pkt, "*in-flight (en vol)*", à acquitter
- Intervalle de numéro de séquence doit être augmenté
- Mise en tampon à l'expéditeur et/ou récepteur



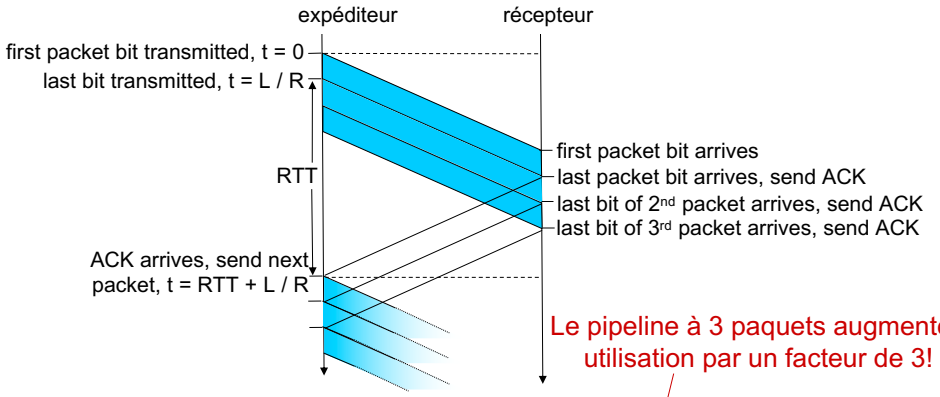
(a) a stop-and-wait protocol in operation (b) a pipelined protocol in operation

- Deux formes génériques de protocoles *pipeline* : *go-Back-N (GBN)* et *selective repeat (SR)*

Transport Layer 3-47

47

Pipelining : utilization augmentée



Le pipeline à 3 paquets augmente utilisation par un facteur de 3!

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

Transport Layer 3-48

48

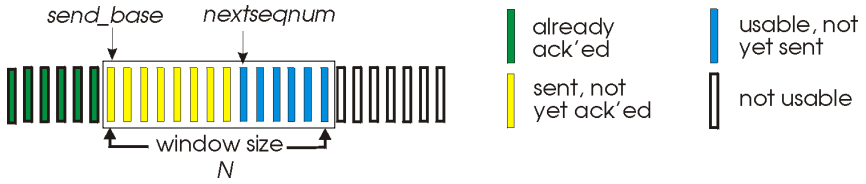
Protocoles Pipeline: vue globale

Go-back-N (GBN) :
- expéditeur peut avoir jusqu'à N pkt sans ack dans le pipeline
- récepteur envoi seulement un *ack cumulatif*
  - ne pas acquitter le pkt si il y a un "écart"
- expéditeur a un compteur pour le plus ancien pkt sans ack
  - Quand compteur expire, retransmet *tous les pkt sans ack*

Selective Repeat (SR) :
- expéditeur peut avoir jusqu'à N pkt sans ack dans le pipeline
- récepteur envoi *ack individuel* pour chaque pkt
- expéditeur maintient un compteur pour chaque pkt sans ack
  - lorsque le délai expire, ne retransmettez que ce paquet sans ack

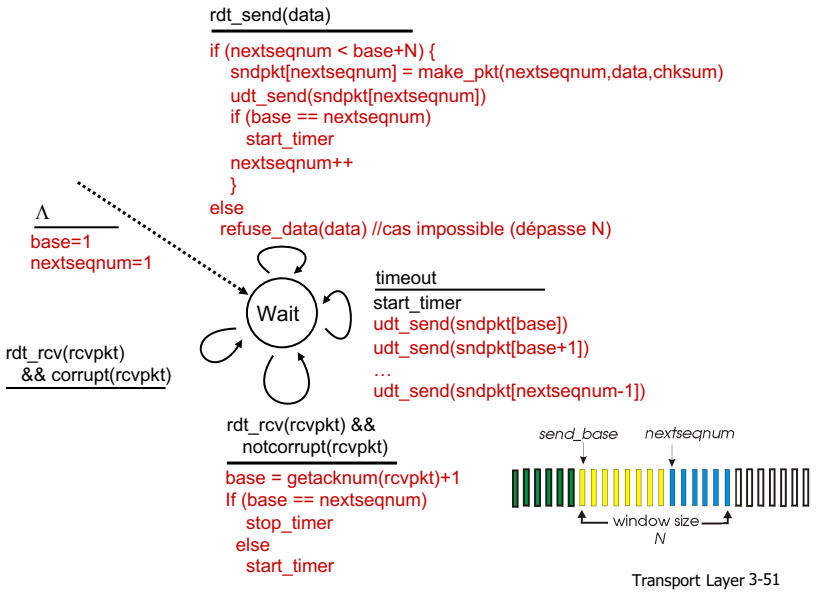
Go-Back-N: expéditeur

- No. seq de k-bit dans l'entête du pkt
- "fenêtre" allant jusqu'à N, pkts consécutifs sans ack autorisés

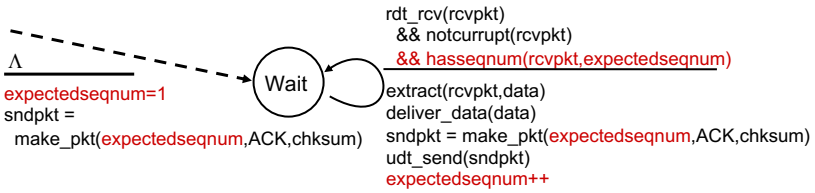


- ACK(n): acquitter tous les pkts jusqu'à no. seq. n (y compris n), *"ACK cumulatif"*
  - Peut recevoir ACKs dupliqué (voir récepteur)
- Compteur pour le pkt le plus ancien dans la fenêtre
- *timeout(n):* retransmet le pkt n ainsi que tous autres qui a le no. seq supérieurs à n dans la fenêtre

GBN: extension expéditeur de FSM



GBN: extension récepteur FSM



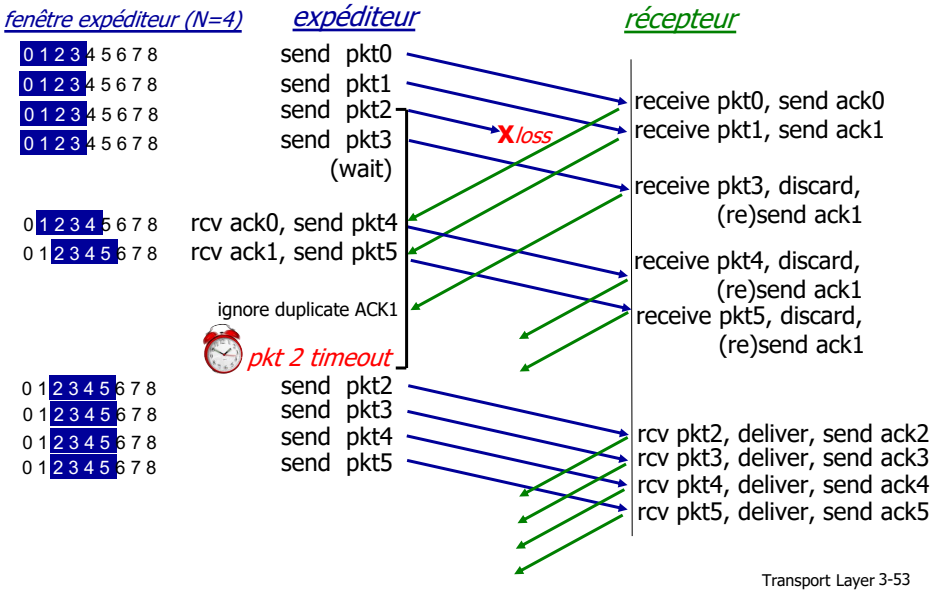
Toujours envoyer un ACK pour le pkt correctement reçu avec le no. seq le plus élevé dans l'ordre

- peut générer des ACK en double
- il suffit de se rappeler du **`expectedseqnum`**

Paquets désordonnés :

- Jeter (ne pas mettre en buffer) : *pas de buffer côté récepteur !*
- re-ACK pkt avec le no. seq le plus élevé dans l'ordre

GBN en action



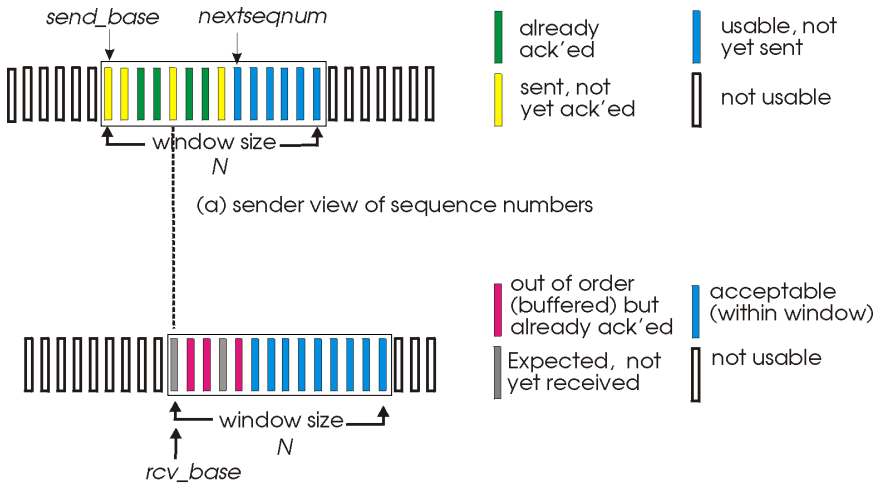
53

Selective repeat (SR)

- Récepteur acquitte *individuellement* tous les pkts correctement reçus
  - Mise en buffer les pkts, au besoin, pour une livraison éventuelle dans la couche supérieure
- Expéditeur ne renvoie que les pkts pour lesquels ACK n'a pas été reçu
  - expéditeur mets un timeout pour chaque pkt sans ACK
- Fenêtre expéditeur
  - N no. seq consécutifs
  - limite les nombres de pkts envoyés et sans ACK.

54

Selective repeat : fenêtre expéditeur et récepteur



55

Selective repeat

expéditeur

Données reçu d'en haut :
- si prochain no.seq disponible dans la fenêtre, envoyer pkt

Timeout(n):
- renvoyer pkt n, redémarrer le minuteur

ACK(n) reçu dans la fenêtre :
- marquer pkt n comme reçu
- si n est le plus petit pkt sans ack, avancez le début de la fenêtre vers le no. seq suivant sans ack

récepteur

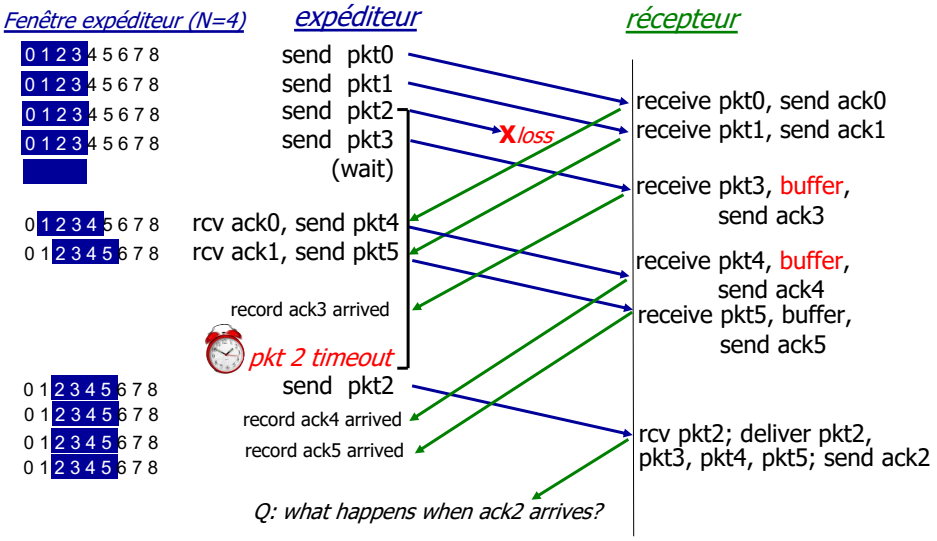
pkt n dans[rcvbase, rcvbase+N-1]
- envoi ACK(n)
- désordonné : met en buffer
- dans l'ordre : livrer (également des pkts en buffer, en ordre), avance la fenêtre au prochain pkt non encore reçu

pkt n dans [rcvbase-N,rcvbase-1]
- ACK(n)

sinon:
- ignorer

56

Selective repeat en action



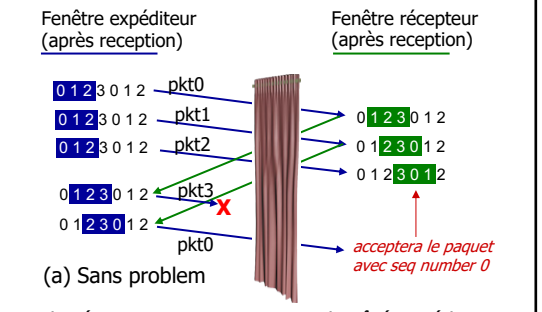
Selective repeat: dilemma

example:

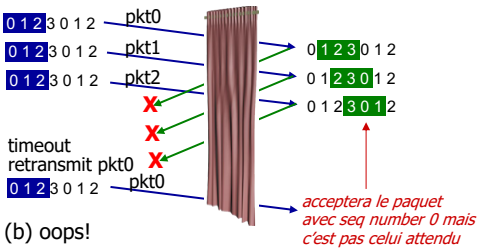
- Seq no.: 0, 1, 2, 3, …
- Taille fenêtre=3

- le récepteur ne voit aucune différence dans deux scénarios!
- données en double acceptées comme nouvelles dans (b)

Q: Quelle relation entre la taille de la séquence et la taille de la fenêtre pour éviter le problème dans (b)?



le récepteur ne peut pas voir le côté expéditeur.
comportement du récepteur identique dans les deux cas
quelque chose ne va pas!



TD3 : exo 3

Chapitre 3: Plan

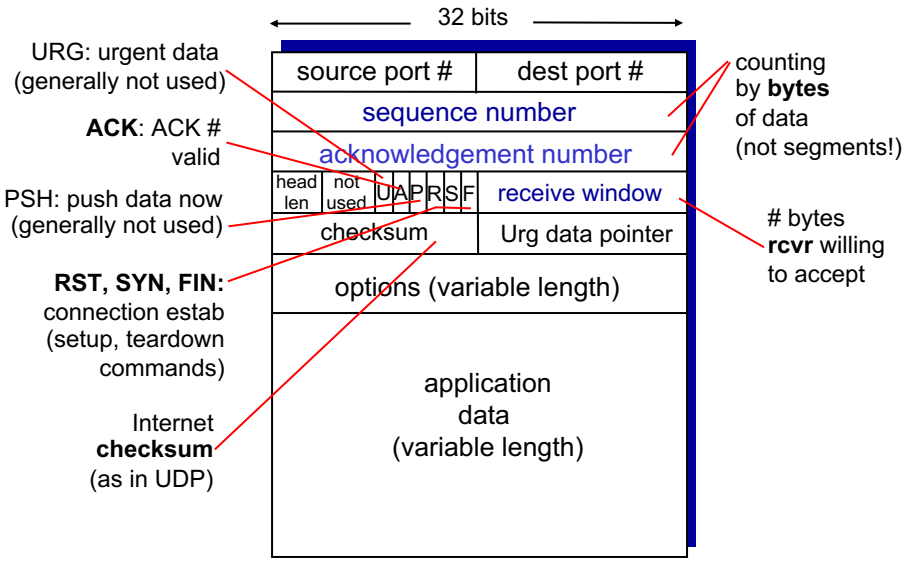
3.1 Services de la couche transport
3.2 Multiplexage et démultiplexage
3.3 Transport sans connexion: UDP
3.4 Principes de transfert de données fiable
3.5 Transport orienté connexion: TCP
- structure de segment
- transfert de données fiable
- contrôle de flux
- gestion des connexions

3.6 Principes du contrôle de la congestion
3.7 Contrôle de congestion TCP

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

- Point-à-point:
  - Un expéditeur, un récepteur
- Flux de données fiable et dans l'ordre :
  - Pas de « frontière de message »
- *Pipeline*:
  - contrôle de congestion et de flux règlent la taille de la fenêtre
- Données full duplex :
  - Flux de données bidirectionnel dans la même connexion
  - MSS: maximum segment size (taille max de données applicatifs)
- Orienté connexion :
  - handshaking (échange de msgs contrôle) initialise les états de expéditeur/récepteur avant l'échange de donnée
- Flux contrôlé :
  - L'expéditeur ne va pas submerger le récepteur

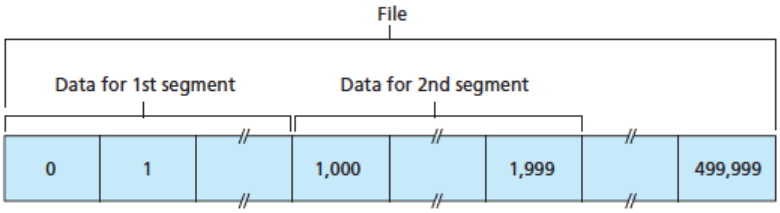
Structure du segment TCP



Numéro de séquence

No. séquence :

- No. du **premier** octet dans le segment de données



Division des données de fichiers en segments TCP

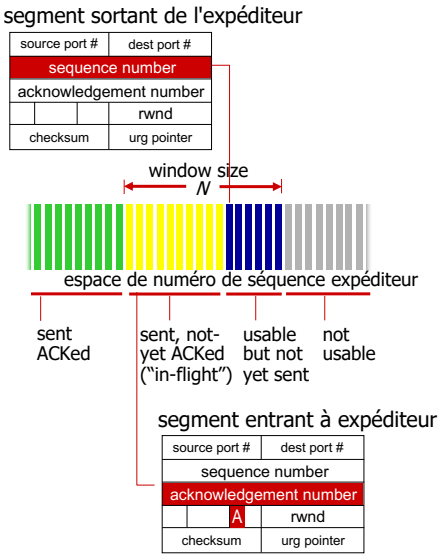
Numéro de ACKs

No. Acknowledgements :

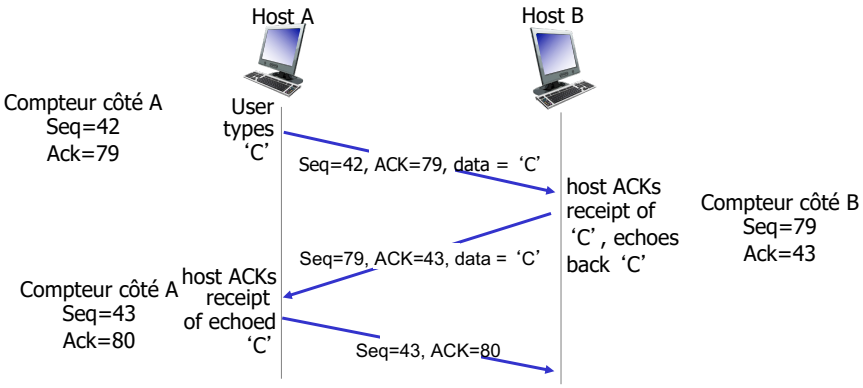
- No. seq d'octet suivant attendu de l'autre côté
- ACK cumulatif

Q : Comment le récepteur gère les segments désordonnés?

R : La spécification TCP ne dit rien – selon l'implémentation



TCP: no. séquence / ACKs



Scenario telnet simple

65

TCP round trip time, timeout

Q : Comment définir la valeur du délai d'attente TCP ?

- Plus long que RTT
  - mais RTT varie
- *Trop court*: timeout prématuré , retx inutiles
- *Trop long*: réaction lente aux pertes de segment

Q : Comment estimer le RTT ?

- **SampleRTT**: temps mesuré depuis la transmission du segment jusqu'à la réception ACK
  - ignorer retransmissions
- SampleRTT variera, vous voulez que le RTT estimé soit plus "lisse"
  - moyenne de plusieurs mesures récentes, et pas seulement de SampleRTT en cours

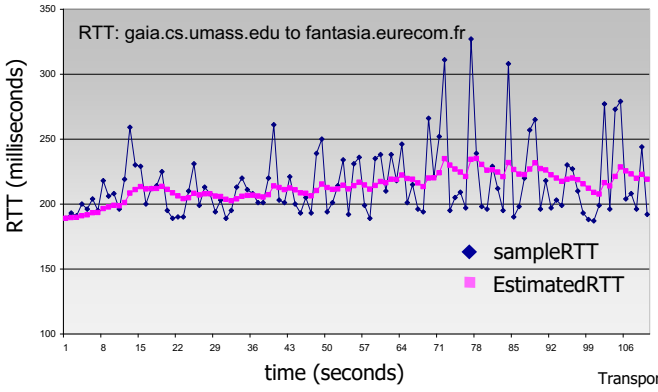
retx=retransmission

66

TCP round trip time, timeout

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

- moyenne glissante pondérée exponentiellement
- l'influence de l'échantillon passé diminue exponentiellement
- valeur type : α = 0.125



TCP: round trip time, timeout

- Intervalle timeout : **EstimatedRTT** plus "marge de sécurité"
  - plus grande variation dans **EstimatedRTT** -> plus grande marge
- Estimer la deviation de SampleRTT depuis EstimatedRTT :

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

(typically, β = 0.25)

TimeoutInterval = EstimatedRTT + 4*DevRTT

RTT estimaté

"marge de sécurité"

TD3 : exo 4

69

Chapitre 3: Plan

3.1 Services de la couche transport
3.2 Multiplexage et démultiplexage
3.3 Transport sans connexion: UDP
3.4 Principes de transfert de données fiable
3.5 Transport orienté connexion: TCP
- structure de segment
- transfert de données fiable
- contrôle de flux
- gestion des connexions

3.6 Principes du contrôle de la congestion
3.7 Contrôle de congestion TCP

70

TCP reliable data transfer (rdt)

- TCP crée un service rdt en plus du service non fiable d'IP
  - segments en pipeline
  - Ack cumulatif
  - minuteur de retransmission unique
- retransmissions déclenchées par :
  - événements « timeout »
  - les acks dupliqués

Considérons au départ l'expéditeur TCP simplifié :

- ignore les acks dupliqués
- ignore le contrôle de flux, le contrôle de congestion

TCP evènements expéditeur

données reçues de l'appli :

- créer un segment avec no. seq
- no. seq est le numéro du premier octet de données dans le segment
- démarrer le minuteur s'il n'est pas déjà en cours d'exécution
  - penser au minuteur comme pour le segment le plus ancien
  - intervalle d'expiration : `TimeOutInterval`

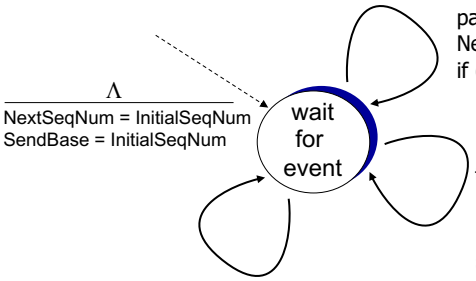
Timeout :

- retransmet le segment qui cause le timeout
- redémarre le timeout

ack reçu :

- si ack acquitte des segments précédemment reçu sans ack
  - mettre à jour ce qui est connu pour être acquittés
  - démarrage le minuteur s'il y a encore des segments sans ack

TCP expéditeur (simplifié)



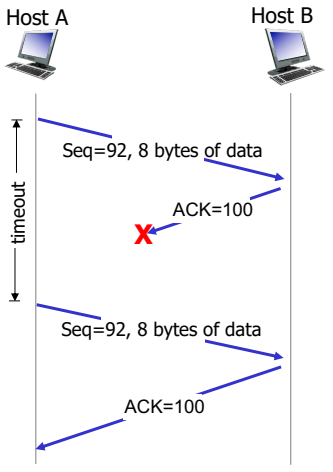
data received from application above
create segment, seq. #: NextSeqNum
pass segment to IP (i.e., "send")
NextSeqNum = NextSeqNum + length(data)
if (timer currently not running)
start timer

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

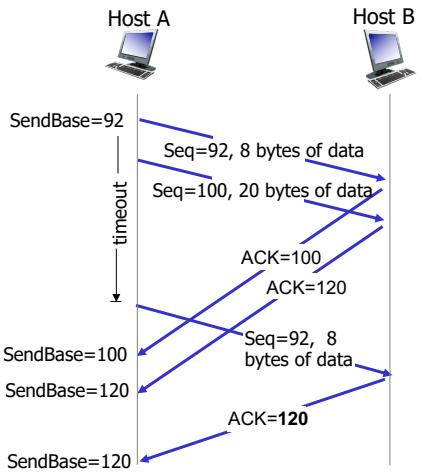
ACK received, with ACK field value y

```
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: qqs scenarios de retransmission

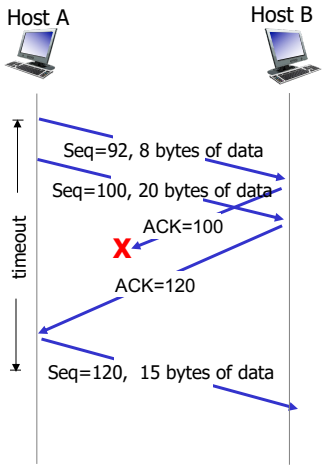


Scénario avec ACK perdu



Scénario avec timeout prématuré

TCP: scenarios de retransmission



Scénario avec ACK cumulatif

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

évènement récepteur	*action récepteur*
arrivée du segment en ordre avec no.seq attendu. Toutes les données jusqu'à seq attendu déjà acquittés	retarde ACK. Attendre jusqu'à 500ms pour le segment suivant. Si pas de segment suivant, envoyer ACK
arrivée du segment en ordre avec no.seq attendu. Un autre segment (en ordre) a un ACK en attente	envoyer immédiatement un ACK cumulatif pour acquitter les deux segments en ordre
arrivée du segment désordonné supérieur à no. prévu. Écart détecté	envoyer immédiatement ACK dupliqué, indiquant no. seq du prochain octet attendu
arrivée du segment qui comble partiellement ou complètement l'intervalle	envoi immédiat ACK, à condition que le segment était le début de l'intervalle

TCP retransmission rapide

- délai d'attente souvent relativement long :
  - long délai avant de renvoyer le paquet perdu
- détecter les segments perdus via des ACK dupliqué.
  - expéditeur envoie souvent plusieurs segments à la suite
  - Si le segment est perdu, il y aura probablement de nombreux ACK dupliqués

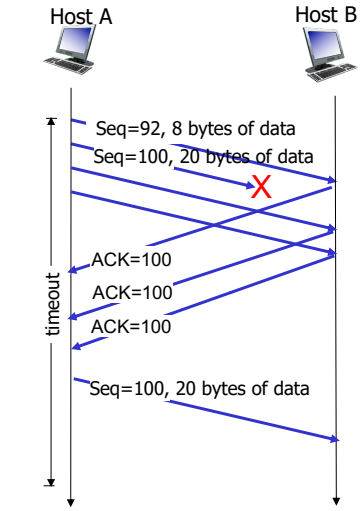
TCP retransmission rapide

si expéditeur recoit 3 ACKs pour même seq ("triple ACKs doublons"), renvoi le segment sans ack qui a le + petit no.

- probable que le segment sans ack soit perdu donc n'attend pas le timeout

77

TCP retransmit rapide



retransmission rapide après expéditeur réception d'un triple ACK

78

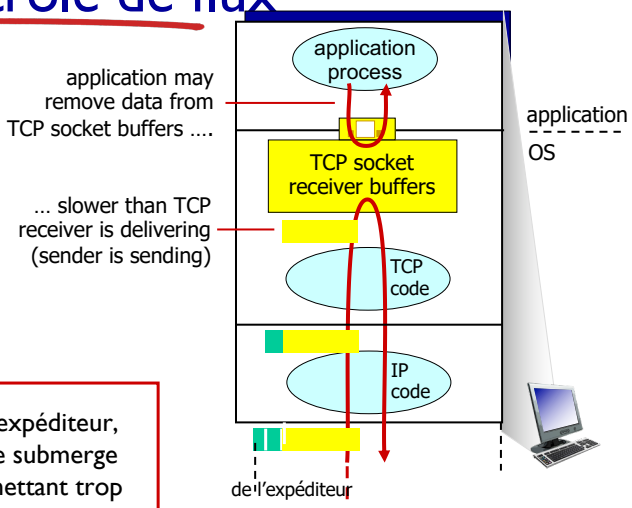
Chapitre 3: Plan

3.1 Services de la couche transport
3.2 Multiplexage et démultiplexage
3.3 Transport sans connexion: UDP
3.4 Principes de transfert de données fiable

3.5 Transport orienté connexion: TCP
- structure de segment
- transfert de données fiable
- contrôle de flux
- gestion des connexions

3.6 Principes du contrôle de la congestion
3.7 Contrôle de congestion TCP

TCP - Contrôle de flux

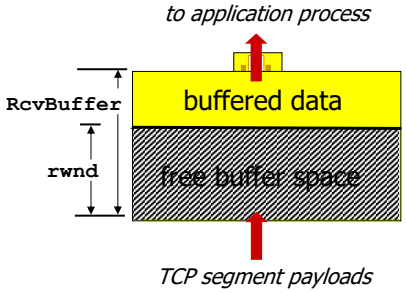


Pile protocolaire du récepteur

Contrôle de flux
Le récepteur contrôle l'expéditeur, pour que l'expéditeur ne submerge pas son buffer en transmettant trop de pkt trop rapidement

TCP - Contrôle de flux

- Le récepteur «annonce» l'espace libre du buffer en incluant la valeur **`rwnd`** dans l'en-tête TCP des segments envoyé à l'expéditeur
  - **`RcvBuffer`** taille définie via les options de socket (par défaut, 4096 octets)
  - nombreux systèmes d'exploitation auto-ajuste **`RcvBuffer`**
- L'expéditeur limite quantité de donnée sans ack ("in-flight") à la valeur de **`rwnd`**
- garantit que le buffer de réception ne débordera pas



Receiver-side buffering

Chapitre 3: Plan

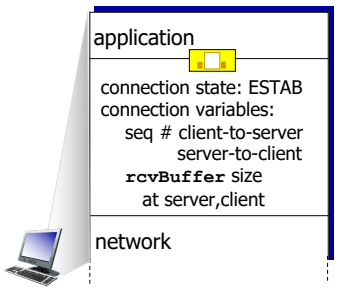
3.1 Services de la couche transport
3.2 Multiplexage et démultiplexage
3.3 Transport sans connexion: UDP
3.4 Principes de transfert de données fiable
3.5 Transport orienté connexion: TCP
- structure de segment
- transfert de données fiable
- contrôle de flux
- gestion des connexions

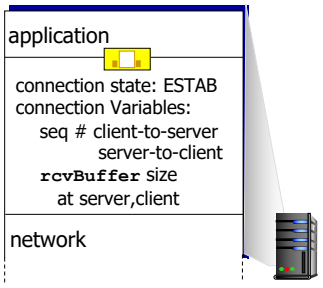
3.6 Principes du contrôle de la congestion
3.7 Contrôle de congestion TCP

Gestion de la connection

Avant l'échange de données, expéditeur/récepteur doivent faire le "handshake" :

- accepter d'établir la connexion (chacun connaissant l'autre disposé à établir la connexion)
- convenir des paramètres de connexion






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

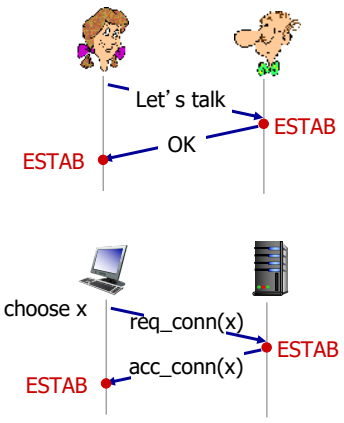
```
Socket connectionSocket =
  welcomeSocket.accept();
```

Transport Layer 3-83

83

Acceptation d'établir une connexion

2-way handshake:



Q: 2-way handshake fonctionnera-t-elle toujours en réseau ?

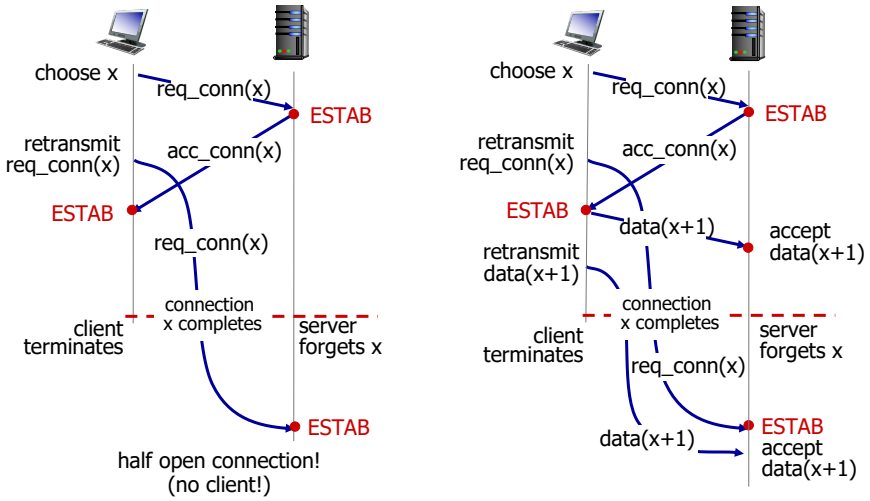
- délai variable
- messages retransmits (e.g., req_conn(x)) à cause des pertes
- ne voit pas de l'autre côté

Transport Layer 3-84

84

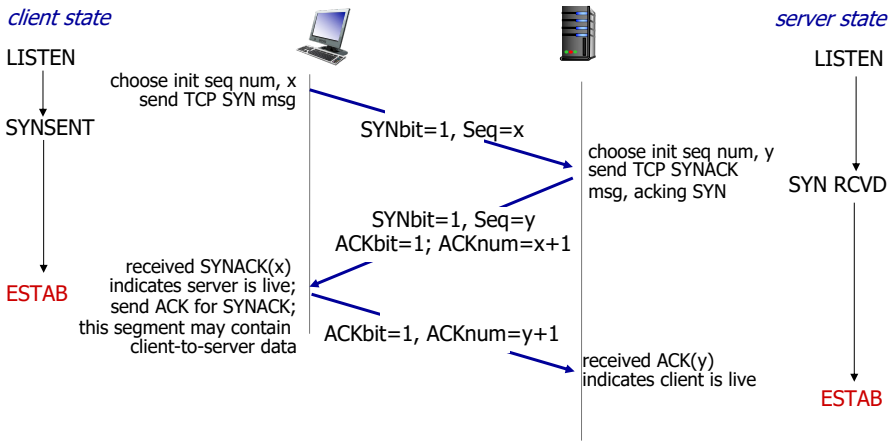
Acceptation d'établir une connexion

2-way handshake, scenario d'échec :



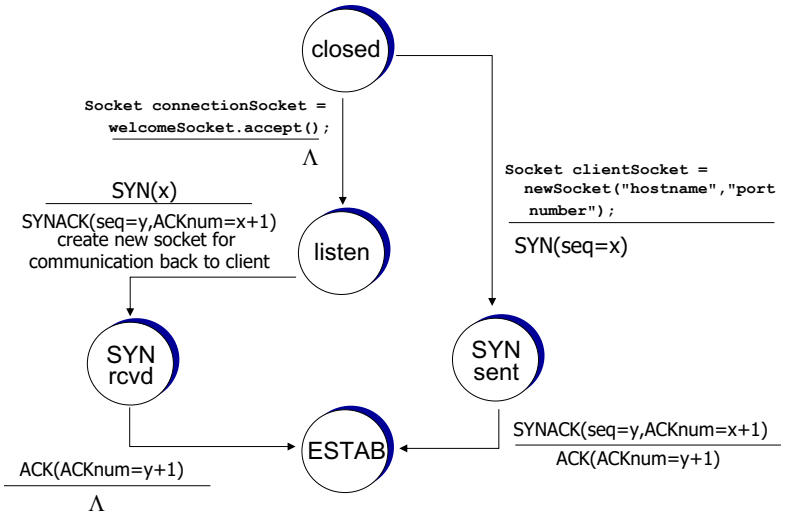
85

TCP 3-way handshake



86

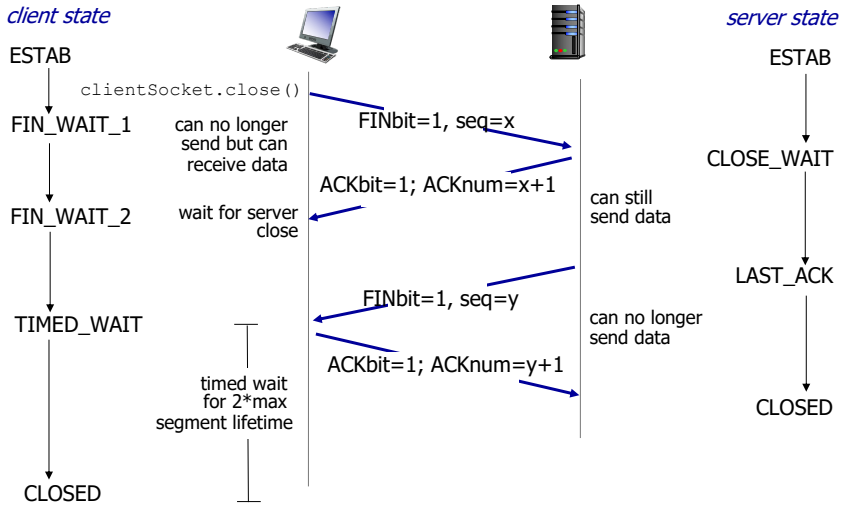
TCP 3-way handshake: FSM



TCP: fermeture de la connexion

- client et serveur ferment chacun son côté de la connexion
  - envoyer un segment TCP avec le bit FIN = 1
- répondre à FIN reçu avec ACK
  - en recevant FIN, ACK peut être combiné avec son propre FIN
- les échanges simultanés FIN peuvent être traités

TCP: fermeture de la connexion



89

Chapitre 3: Plan

3.1 Services de la couche transport
3.2 Multiplexage et démultiplexage
3.3 Transport sans connexion: UDP
3.4 Principes de transfert de données fiable
3.5 Transport orienté connexion: TCP
- structure de segment
- transfert de données fiable
- contrôle de flux
- gestion des connexions

3.6 Principes du contrôle de la congestion
3.7 Contrôle de congestion TCP

90

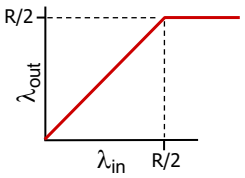
Principes du contrôle de la congestion

Congestion :

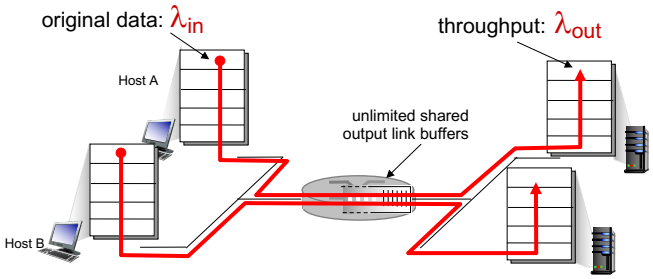
- Informel : "trop de sources envoient trop de données trop rapidement pour que le *réseau* puisse les gérer"
- différent du contrôle de flux !
- Résultats :
  - paquets perdus (débordement de buffer aux routeurs)
  - longs délais (file d'attente dans les buffer de routeurs)
- un des top 10 de problèmes en réseau !

Causes/coûts de la congestion: scénario 1

- 2 expéditeurs, 2 récepteurs
- un routeur, buffer infini
- capacité de lien sortie : R
- sans retransmission

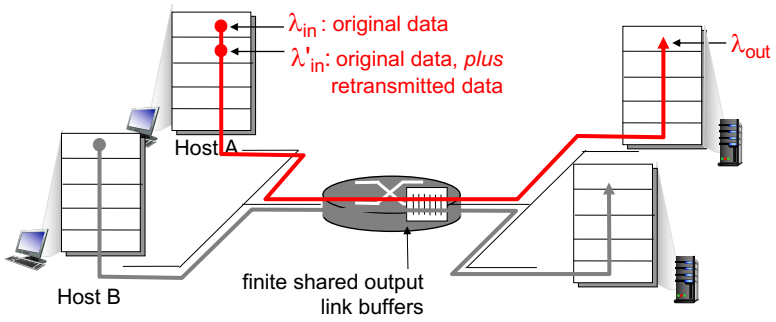


- débit maximum par connexion : R/2



Causes/coûts de la congestion : scénario 2

- un router, buffer de capacité *limitée*
- retransmission de l'expéditeur du paquet expiré
  - entrée de couche d'application= sortie de couche d'application: $\lambda_{in} = \lambda_{out}$
  - entrée de couche transport comprenant rtx : $\lambda'_{in} \geq \lambda_{in}$

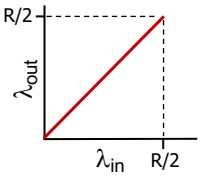


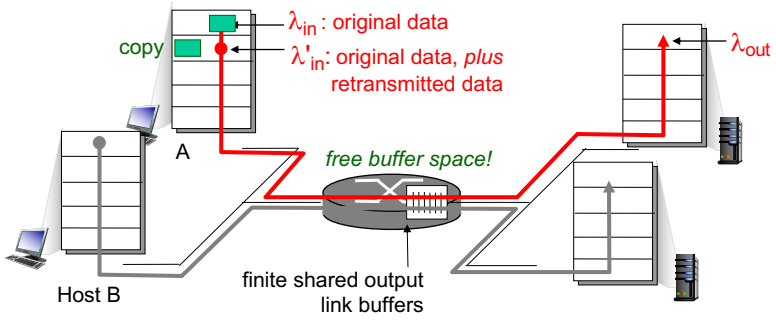
93

Causes/coûts de la congestion : scénario 2

Idéaliste : connaissance parfaite

- expéditeur envoie uniquement lorsque le tampon de routeur est disponible

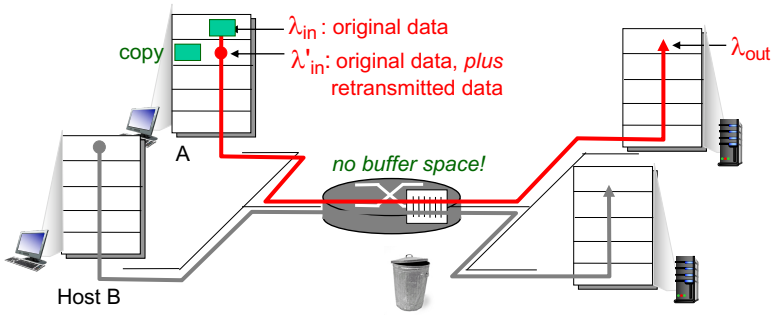




94

Causes/coûts de la congestion: scénario 2

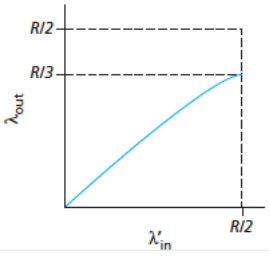
Idéaliste : des pertes connues
les paquets peuvent être perdus, supprimé au routeur en raison de tampon plein
- expéditeur ne renvoie que si le paquet est *connu* comme perdu



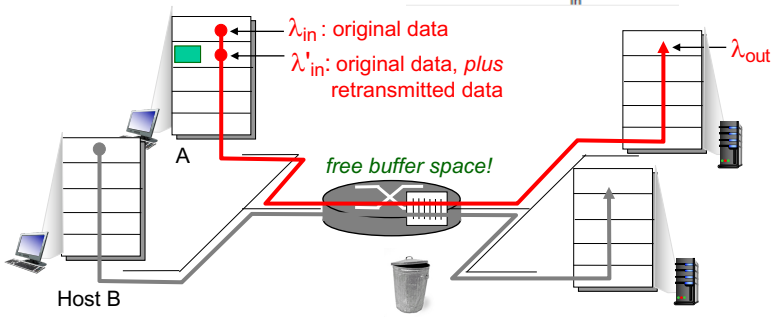
95

Causes/coûts de la congestion: scénario 2

Idéaliste : des pertes connues
les paquets peuvent être perdus, supprimé au routeur en raison de tampon plein
- expéditeur ne renvoie que si le paquet est *connu* comme perdu



Lors de l'envoi à R/2, certains pkt sont des retransmissions, R/3 sont des originaux en moyenne.

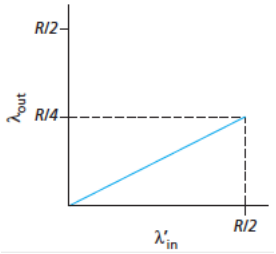


96

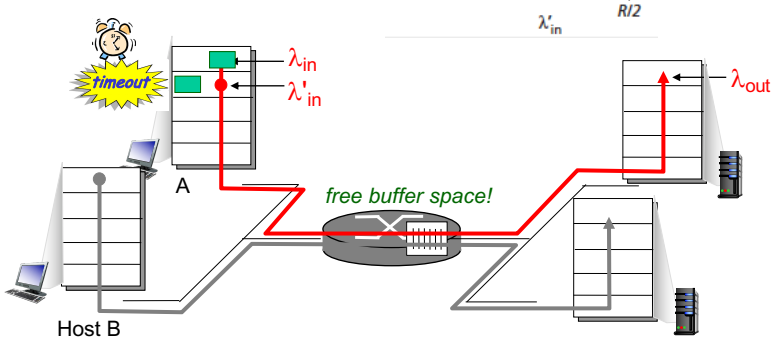
Causes/coûts de la congestion : scénario 2

Réaliste : doublons

- les paquets peuvent être perdus, supprimé au routeur en raison de tampon plein
- expiration prématurée de l'expéditeur, envoi de *deux* copies, toutes deux livrées



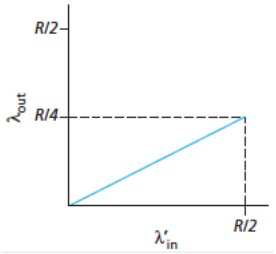
lors de l'envoi à R/2, certains paquets sont des retransmissions, y compris les doublons livrés !



Causes/coûts de la congestion : scénario 2

Réaliste : doublons

- les paquets peuvent être perdus, supprimé au routeur en raison de tampon plein
- expiration prématurée de l'expéditeur, envoi de *deux* copies, toutes deux livrées



lors de l'envoi à R/2, certains paquets sont des retransmissions, y compris les doublons livrés !

"coût" de la congestion :

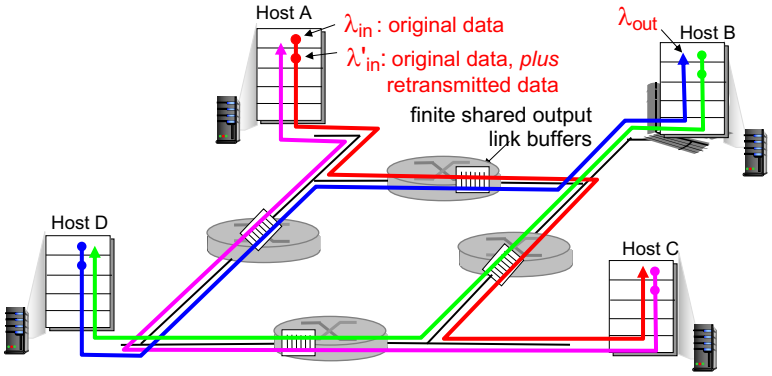
- plus de travail (retransmissions) pour un même rendement
- retransmissions inutiles : le lien porte plusieurs copies de pkt
  - rendement décroissant

Causes/coûts de la congestion : scénario 3

- 4 expéditeurs
- chemin multi-saut
- timeout/retransmission

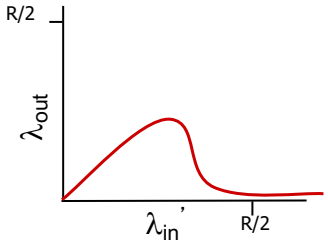
Q : que se passe-t-il si λ_{in} et λ_{in}' augmentent ?

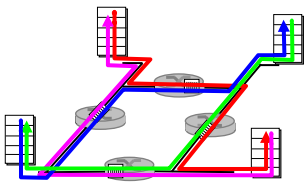
R : comme λ_{in}' augmente, tous les pkts bleus arrivant dans la file d'attente supérieure sont abandonnés, débit de bleu → 0



99

Causes/coûts de la congestion : scénario 3





un autre «coût» de la congestion :

- en cas de paquet abandonné, toute «capacité de transmission en amont » utilisée pour ce paquet est gaspillée !

100

Chapitre 3: Plan

3.1 Services de la couche transport
3.2 Multiplexage et démultiplexage
3.3 Transport sans connexion: UDP
3.4 Principes de transfert de données fiable
3.5 Transport orienté connexion: TCP
- structure de segment
- transfert de données fiable
- contrôle de flux
- gestion des connexions

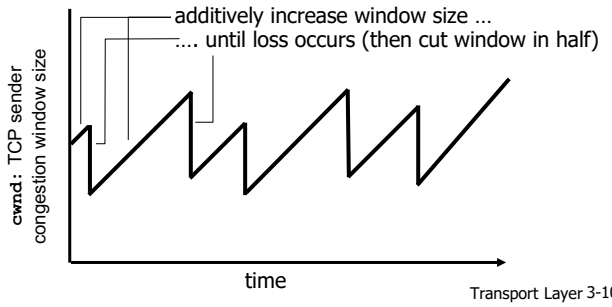
3.6 Principes du contrôle de la congestion
3.7 Contrôle de congestion TCP

101

TCP Contrôle de congestion : additive increase multiplicative decrease (AIMD)

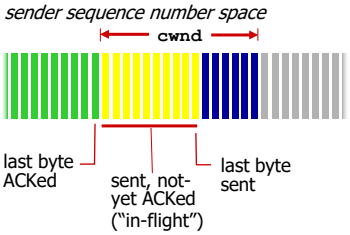
- *Approche :* expéditeur augmente le taux de tx (taille de fenêtre), teste la bande passante utilisable, jusqu'à ce que la perte se produise
  - *additive increase :* augment **`cwnd`** part 1 MSS à chaque RTT jusqu'à la perte détectée
  - *multiplicative decrease :* diminue **`cwnd`** de moitié après la perte

AIMD saw tooth Behavior : probing for bandwidth



102

TCP Contrôle de congestion : details



- expéditeur limite la tx :

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

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

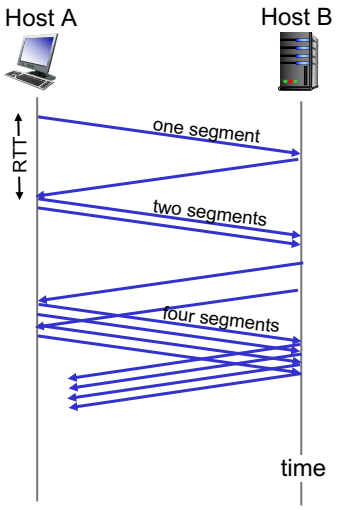
Taux de tx TCP :

- *en gros :* envoie cwnd octets, attend RTT pour ACKS, puis envoie plus d'octets

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

TCP Slow Start (démarrage lent)

- Lorsque la connexion commence, augmentez le taux de manière exponentielle jusqu'au premier événement de perte :
  - **cwnd** initiale = 1 MSS
  - doubler **cwnd** à chaque RTT
  - fait par une incrémentation de **cwnd** pour chaque ACK reçu
- *Résumé :* taux initial est lent mais grimpe de manière exponentielle



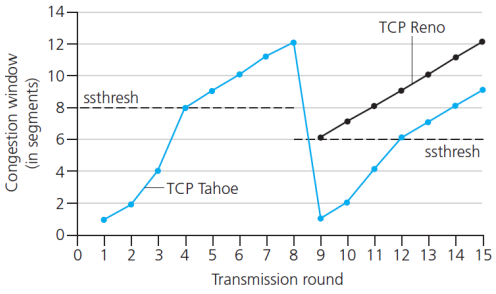
TCP : détecter et réagir à la perte

- perte indiquée par le timeout :
  - **cwnd** est diminuée à 1 MSS ;
  - la fenêtre augmente ensuite de façon exponentielle (comme au démarrage lent) pour atteindre le seuil, puis croît linéairement
- perte indiquée par triple ACK : TCP RENO
  - ACKs tripliqués indiquent un réseau capable de délivrer certains segments
  - **cwnd** est diminuée de moitié puis incrémentée linéairement
- TCP Tahoe règle **cwnd** toujours à 1 (timeout ou triple acks)

105

TCP : passer de slow start à CA

Q : Quand est-ce que l'augmentation exponentielle passe en linéaire ?
R : quand **cwnd** atteint la moitié de sa valeur avant expiration.

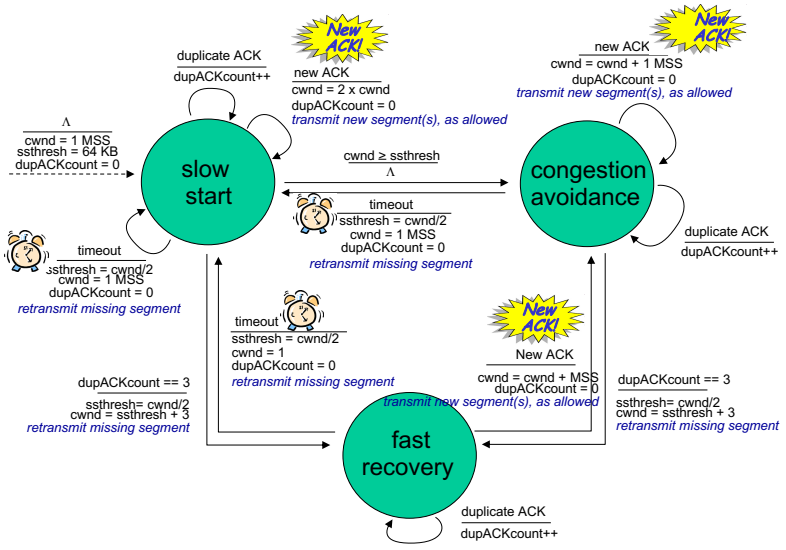


Implementation :

- variable **ssthresh**
- en cas de perte, **ssthresh** est réglé sur ½ de **cwnd** (juste avant l'événement de perte)

106

Résumé: TCP Contrôle de congestion



Chapitre 3: résumé

- principes derrière les services de couche de transport:
  - multiplexage, démultiplexage
  - transfert de données fiable
  - contrôle de flux
  - contrôle de la congestion
- instanciation, implémentation sur Internet
  - UDP
  - TCP

La suite:

- quitter le "edge" du réseau (couches application et transport)
- pour le «cœur» du réseau

TD3 : exo 5&6