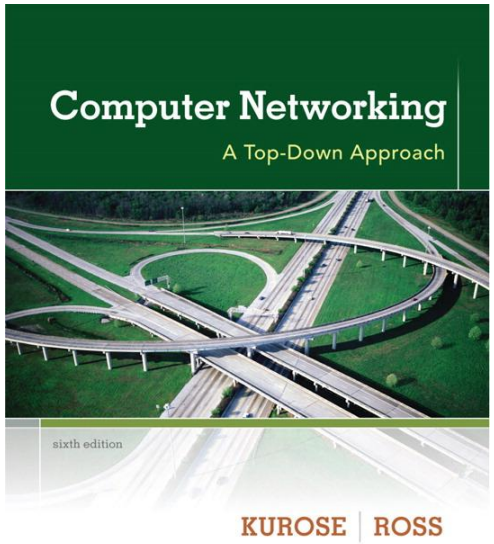
Chapitre III
La couche transport



A note on the use of these Powerpoint slides:

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

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

Thanks and enjoy! JFK/KWR

Computer Networking: A Top Down Approach
6th edition
Jim Kurose, Keith Ross
Addison-Wesley
March 2012

Chapitre III: La couche transport

objectifs:

- comprendre les principes des services de la couche transport:
  - multiplexage/ démultiplexage
  - transfert fiable de données
  - contrôle de flux
  - contrôle de congestion
- découvrir les protocoles de la couche transport:
  - UDP: transport sans connexion
  - TCP: transport fiable orienté connexion
  - contrôle de congestion de TCP

Chapitre III: plan

3.1 services de la couche transport

3.2 multiplexage et démultiplexage

3.3 transport sans connexion: UDP

3.4 principes du transfert fiable des données

3.5 transport orienté connexion: TCP

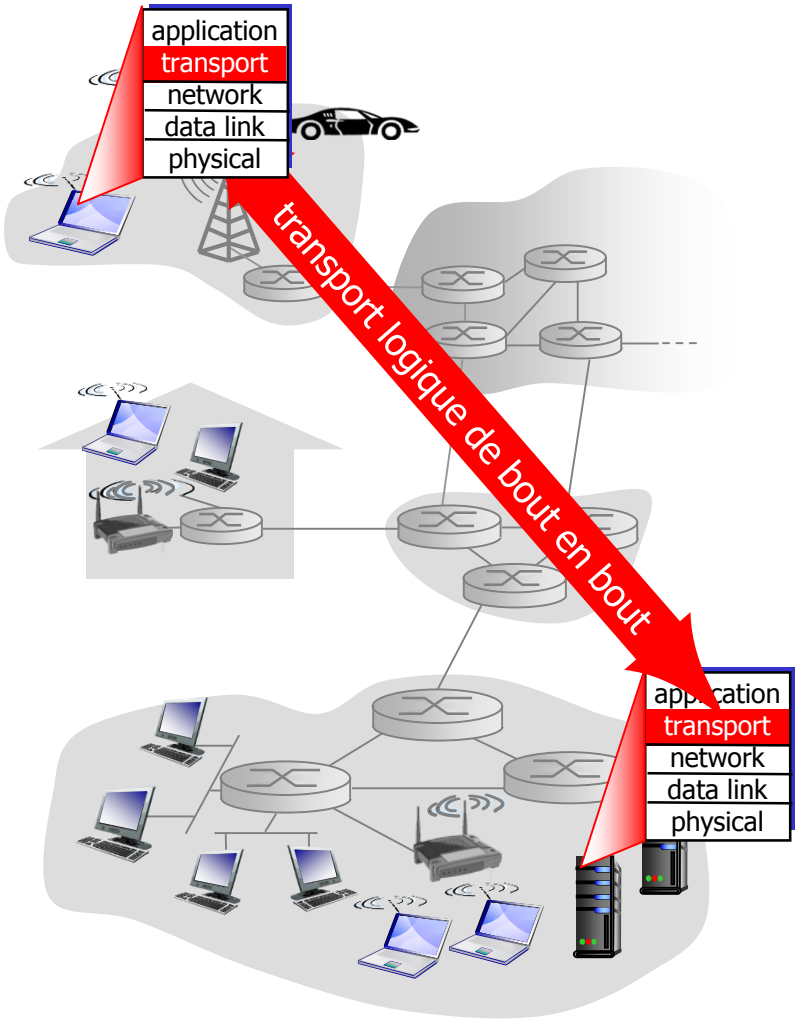
- structure d'un segment
- transfert fiable
- contrôle de flux
- gestion de connexion

3.6 principes du contrôle de congestion

3.7 contrôle de congestion dans TCP

Services et protocoles

- une **communication logique** entre les processus exécutés sur différents hôtes
- les protocoles de transport s'exécutent sur les terminaux
  - Émetteur: **découper** le message en segments, les passer à la couche réseau
  - Récepteur: **réassembler** les segments en messages, les passer à la couche application
- plusieurs protocoles disponibles aux apps
  - Internet: TCP et UDP



Couche transport vs. réseau

- *couche réseau:* communication logique entre hôtes
- *couche transport:* communication logique entre processus
  - dépend de et améliore les services de la couche réseau

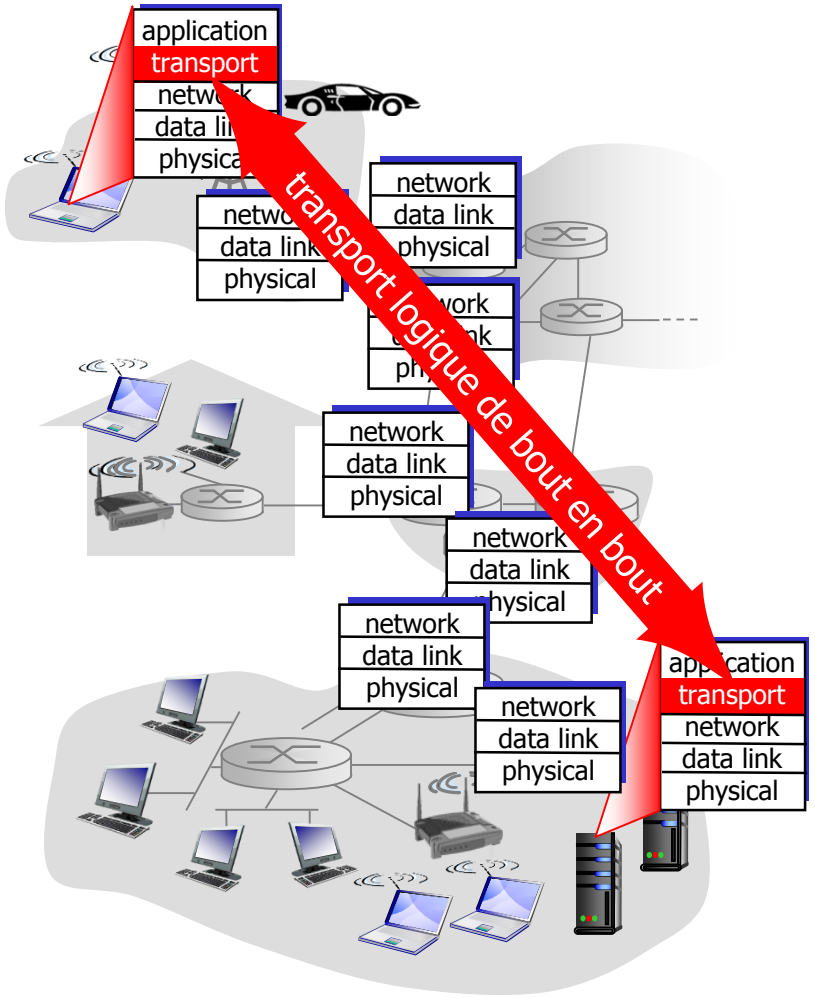
analogie:

12 enfants de la maison de Anne envoie des lettres à 12 enfants de la maison de Bill:

- hôtes = maisons
- processus = enfants
- messages app = lettres dans enveloppes
- protocole transport = Anne et Bill
- protocole réseau = service de poste

Protocoles de la couche transport

- fiable, délivre en ordre : TCP
  - contrôle de congestion
  - contrôle de flux
  - établissement de connexion
- non fiable, délivre sans ordre: UDP
  - extension de IP “best-effort”
- services non dispo.:
  - garanties de délai ou de bande passante



Chapitre III: plan

3.1 services de la couche transport

3.2 multiplexage et démultiplexage

3.3 transport sans connexion: UDP

3.4 principes du transfert fiable des données

3.5 transport orienté connexion: TCP

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

3.6 principes du contrôle de congestion

3.7 contrôle de congestion dans TCP

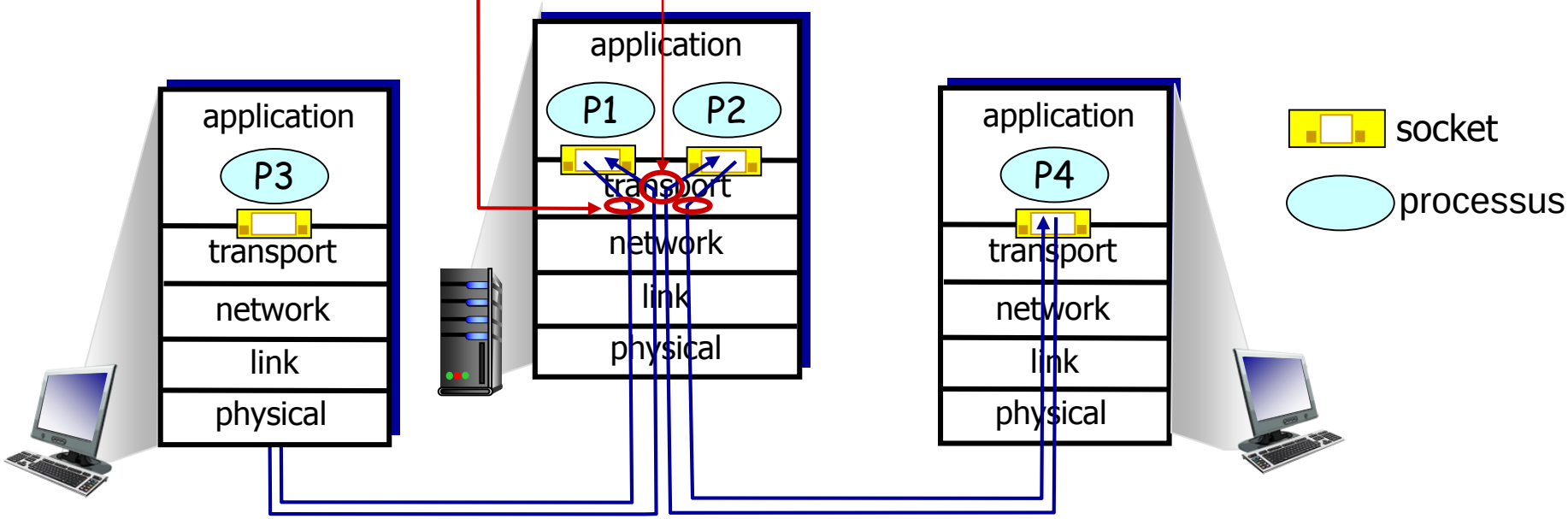
Multiplexage/démultiplexage

multiplexage à l'émetteur:

collecte les données de plusieurs sockets, puis ajoute les en-têtes (utilisées plus tard)

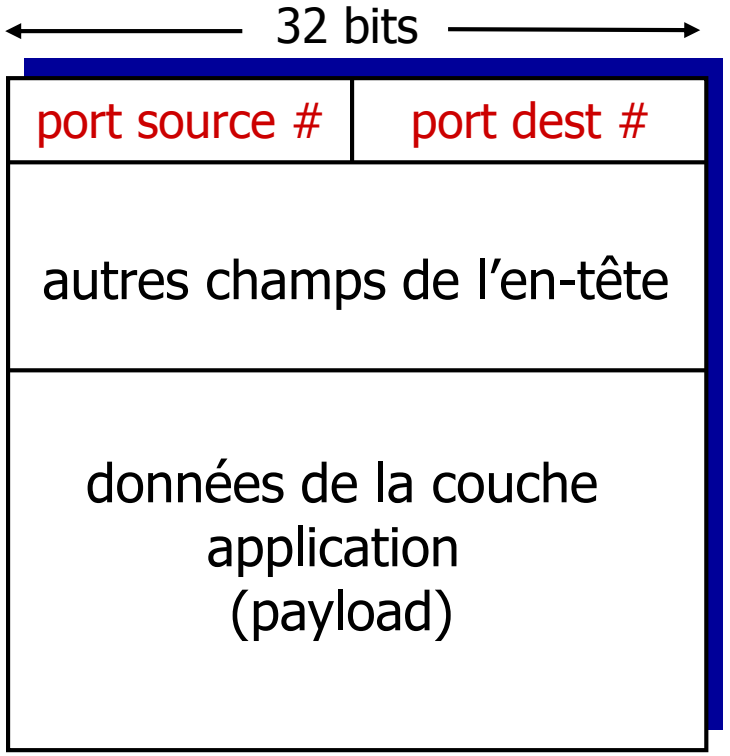
démultiplexage au recepteur:

utilise les infos dans les en-têtes des segments reçus pour les délivrer à la bonne socket



Démultiplexage (Démux.)

- un hôte reçoit des datagrammes IP
  - chaque datagramme a une adresse IP source et destination
  - chaque datagramme transporte un segment
  - un segment possède un port source et un port destination
- un hôte utilise les *adresses IP & numéros de port* pour diriger les segments vers la bonne socket



format d'un segment TCP/UDP

Démux. sans connexion

- lors de la création d'un paquet pour envoie sur un socket UDP on doit spécifier
  - adresse IP destination
  - # port destination

- quand l'hôte reçoit un segment UDP :
  - vérifie le # port destination
  - dirige le segment UDP au socket qui possède le # port

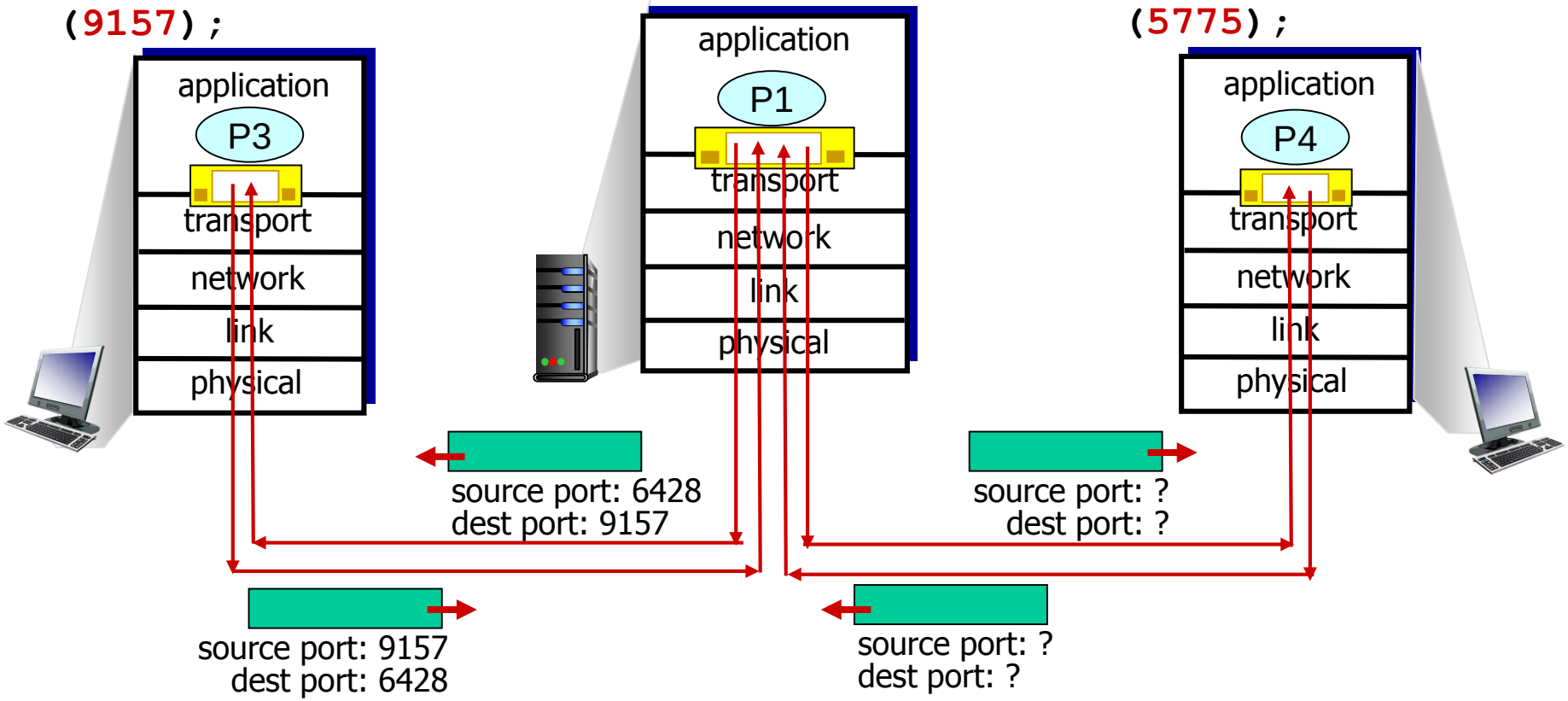
les datagrammes IP avec le même # port destination mais une adresse IP source différente ou un # port source différent sont dirigés au même socket

Démux. sans connexion: exemple

```
DatagramSocket
 mySocket2 = new
 DatagramSocket
 (9157);
```

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

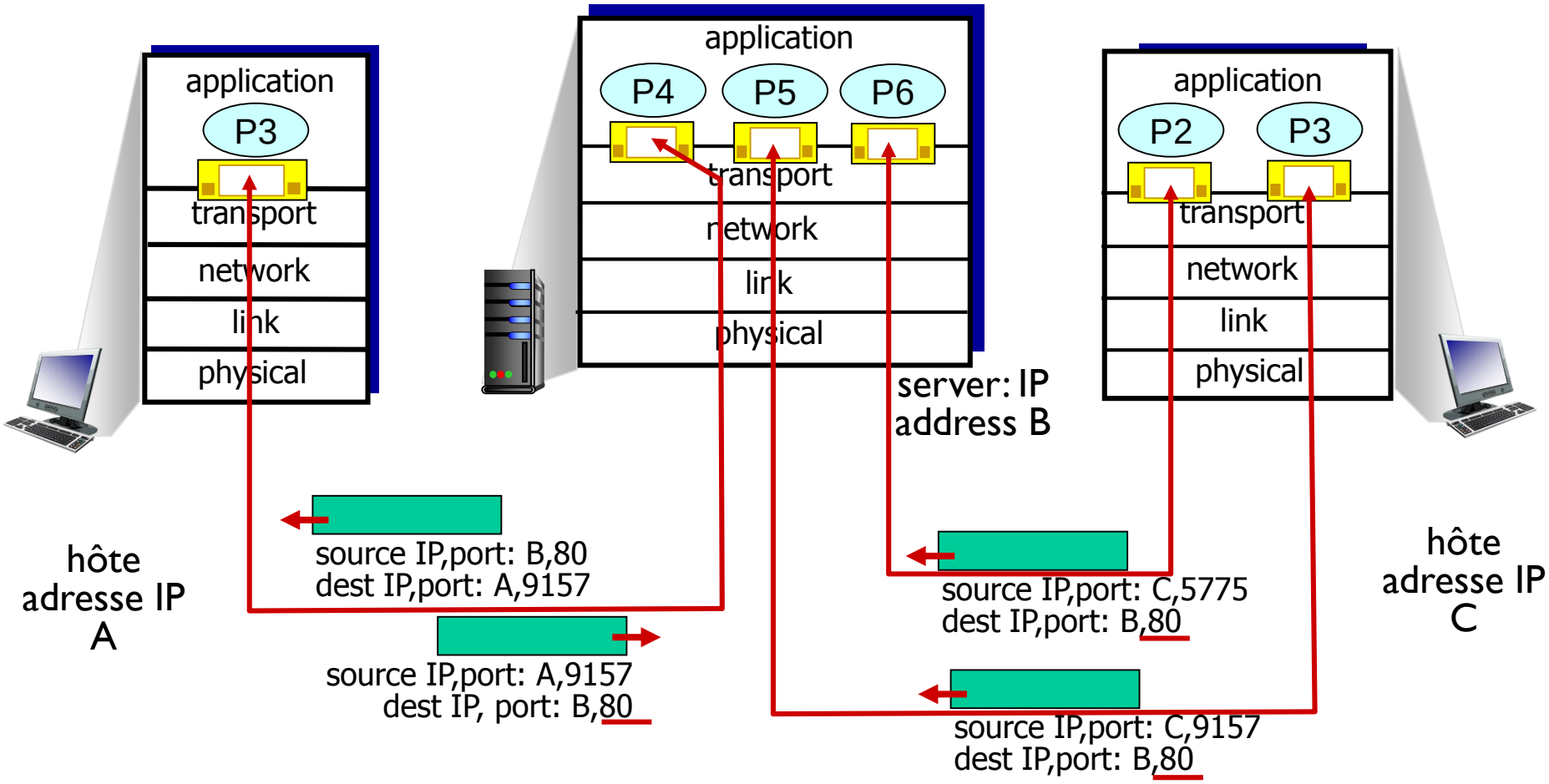
```
DatagramSocket
 mySocket1 = new
 DatagramSocket
 (5775);
```



Démux. orienté connexion

- socket TCP identifié par le quadruplet:
  - adresse IP source
  - numéro du port source
  - adresse IP destination
  - numéro du port dest.
- Le récepteur utilise les 4 valeurs pour diriger le segment vers le socket approprié
- Le serveur peut supporter plusieurs sockets TCP simultanément:
  - chaque socket est identifié par son propre quadruplet
- Les serveurs Web ont un socket pour chaque client connecté
  - HTTP non-persistent va avoir un socket pour chaque requête

Démux. orienté connexion: exemple



trois segments, destinés à l'adresse IP : B,
port dest : 80 sont démultiplexés vers des sockets *differents*

Chapitre III: plan

3.1 services de la couche transport

3.2 multiplexage et démultiplexage

3.3 transport sans connexion: UDP

3.4 principes du transfert fiable des données

3.5 transport orienté connexion: TCP

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

3.6 principes du contrôle de congestion

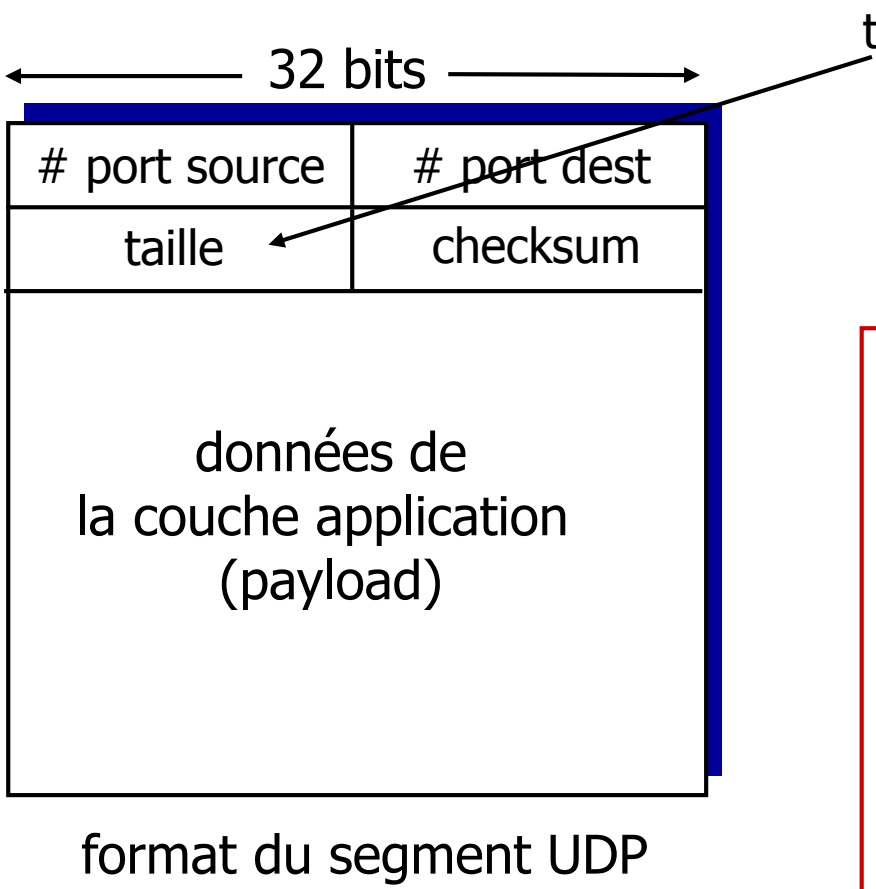
3.7 contrôle de congestion dans TCP

UDP: User Datagram Protocol [RFC 768]

- protocole de transport « bas coût »
- service « au mieux », UDP segments peuvent:
  - être perdus
  - être délivrés sans ordre
- *sans connexion:*
  - pas de poignée de mains entre émetteur et récepteur
  - chaque segment est traité indépendamment des autres

- utilisation de UDP:
  - streaming multimédia (tolérant aux pertes, sensible au débit)
  - DNS
  - SNMP
- transfert fiable sur UDP:
  - ajout de la fiabilité à la couche application

UDP: en-tête du segment



format du segment UDP

pourquoi UDP existe?

- pas d'établissement de connexion (moins de délai)
- simple: pas d'état de connexion
- une petite en-tête
- pas de contrôle de congestion: UDP peut transmettre aussi rapidement qu'il veut

UDP: somme de contrôle (checksum)

but: détecter les "erreurs" (ex. bits modifiés) dans le segment transmis

émetteur:

- traite le contenu du segment comme une séquence d'entiers de 16-bits
- calcule le checksum: le complément à un de la somme de tous les mots
- met la valeur de la somme de contrôle dans le champ checksum d'UDP

récepteur:

- calcule la somme de contrôle du segment reçu
- vérifie si la somme de contrôle calculée égale à la valeur du champ checksum:
  - NON - erreur détectée
  - OUI – pas d'erreurs détectées mais peut-être il y en a? À suivre

checksum: exemple

exemple: somme de deux entiers 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
ajout du reste	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: quand on additionne des nombres, le reste dans le bit le plus signifiant doit être ajouté au résultat

Chapitre III: plan

3.1 services de la couche transport

3.2 multiplexage et démultiplexage

3.3 transport sans connexion: UDP

3.4 principes du transfert fiable des données

3.5 transport orienté connexion: TCP

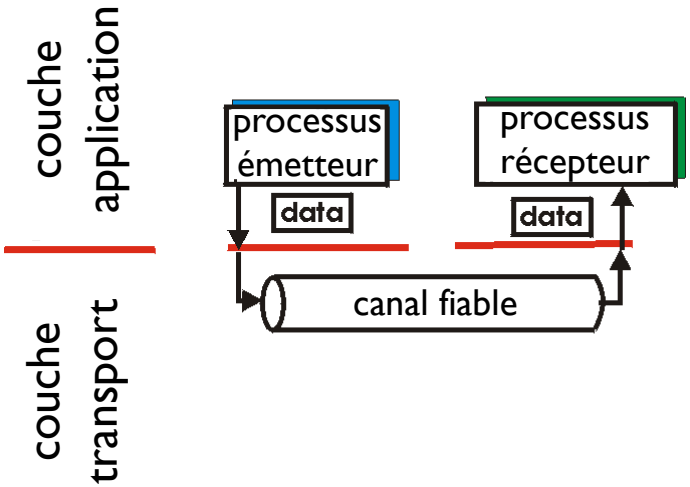
- structure d'un segment
- transfert fiable
- contrôle de flux
- gestion de connexion

3.6 principes du contrôle de congestion

3.7 contrôle de congestion dans TCP

Principes du transfert fiable

- important dans les couches application, transport et liaison

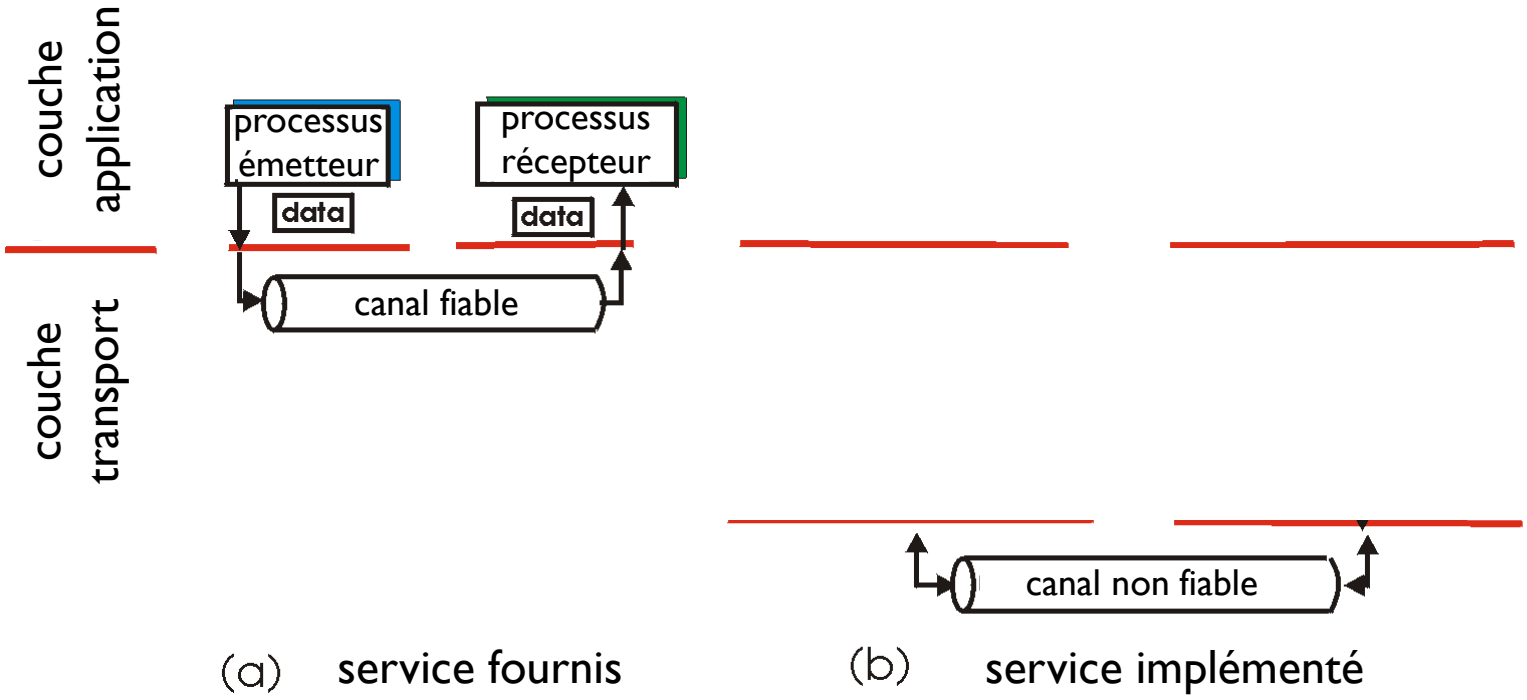


(a) service fournis

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

Principes du transfert fiable

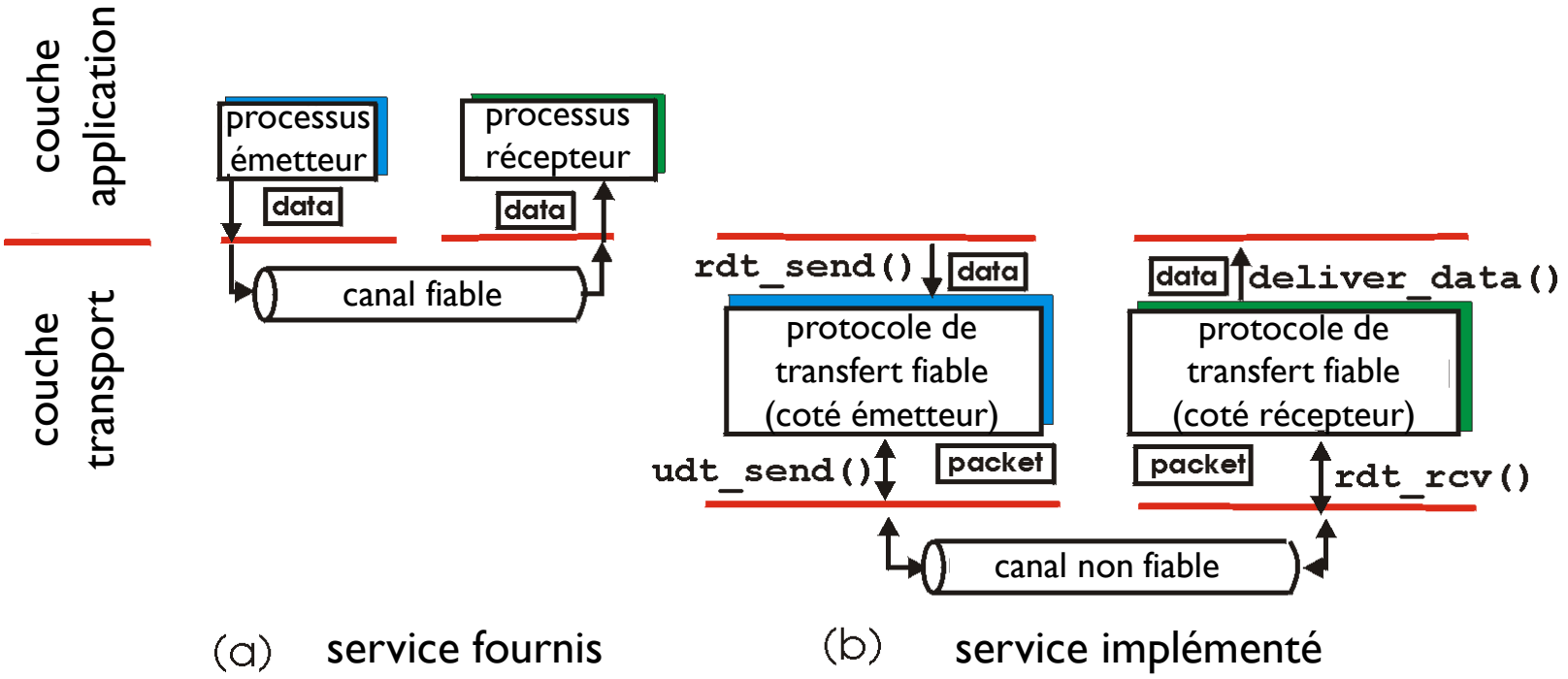
- important dans les couches application, transport et liaison



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

Principes du transfert fiable

- important dans les couches application, transport et liaison



- Les caractéristiques d'un canal non fiable détermineront la complexité du protocole de transfert fiable de données

Canal fiable

- Si le canal est parfaitement fiable
  - pas de bits en erreur
  - pas de paquets perdus

- Alors
  - L'émetteur envoie des données sur le canal
  - Le récepteur lit les données du canal

Canal avec erreurs

- le canal peut modifier des bits
  - le checksum peut détecter ceci
- *la* question: comment corriger ces erreurs?

comment les humains procédent?

Canal avec erreurs

- le canal peut modifier des bits
  - le checksum peut détecter ceci
- *la* question: comment corriger ces erreurs?
  - *acquiescements (ACKs): le Rx informe explicitement le Tx que le paquet est bien reçu*
  - *acquiescements négatifs (NAKs): le Rx informe explicitement le Tx que le paquet contient des erreurs*
  - *Le Tx retransmet le paquet à la réception d'un NAK*
- *le protocole implémente de nouveaux mécanismes:*
  - *détection d'erreurs*
  - *retour du récepteur: msgs de contrôle (ACK,NAK) Rx->Tx*

Un défaut fatal!

Et si un ACK/NAK est corrompu?

- l'émetteur n'a pas d'information sur ce qui s'est passé au récepteur!
- ne peut retransmettre: duplication possible

Traiter la duplication:

- L'émetteur retransmets le paquet actuel si ACK/NAK est corrompu
- L'émetteur ajoute un *numéro de séquence* à chaque paquet
- Le récepteur élimine (ne délivre pas vers le haut) le paquet dupliqué

arrêt et attente

l'émetteur envoie un paquet puis attend la réponse du récepteur

Discussion

émetteur:

- Un # séq. est ajouté au paquet
- deux # séq. (0,1) suffiront. pourquoi?
- il doit vérifier si le ACK/NAK reçu est corrompu
- deux états
  - L'état doit "se rappeler" si le paquet attendu a un # 0 ou 1

récepteur:

- doit vérifier si le paquet reçu est dupliqué
  - L'état indique si le # séq. 0 ou 1 est attendu
- note: le récepteur ne peut pas savoir si le dernier ACK/NAK est reçu correctement du côté émetteur

Protocole sans NAK

- Les mêmes fonctionnalités en utilisant des ACKs seulement
- À la place de NAK, le récepteur envoie un ACK pour le dernier paquet reçu correctement
  - Le récepteur doit explicitement inclure le # séq. du paquet déjà acquiescé
- ACK dupliqué à l'émetteur donne le même résultat qu'un NAK: retransmettre le présent paquet

Canal avec erreurs et pertes

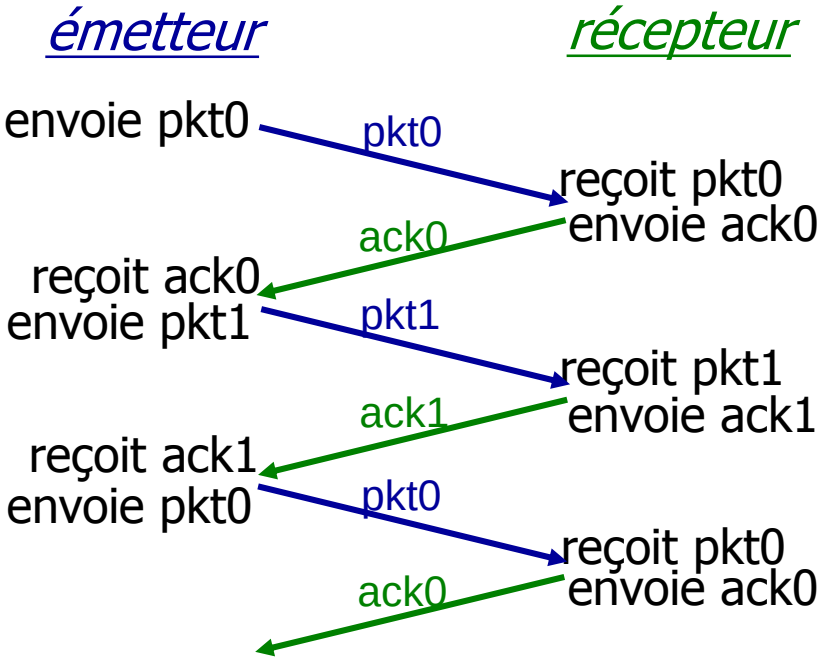
hypothèse: le canal peut aussi perdre des paquets (données ou ACKs)

- checksum, # seq., ACKs, retransmissions peuvent aider mais ne sont pas suffisants

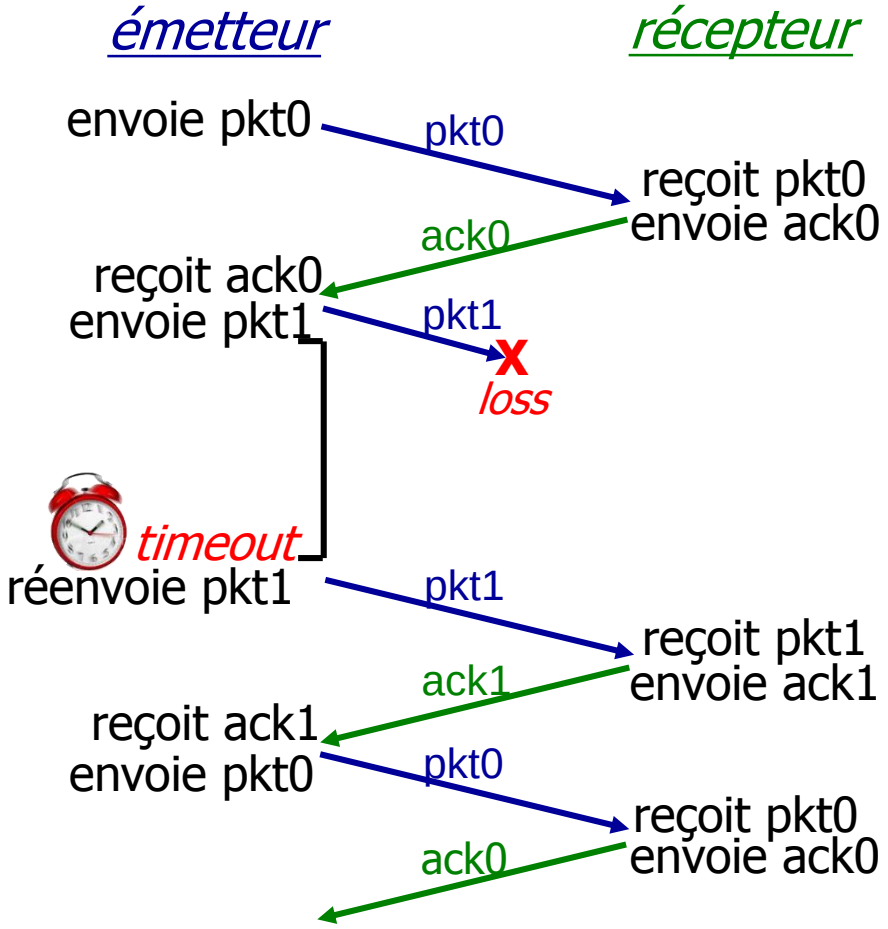
approche: l'émetteur attend un temps "raisonnable" pour ACK

- Il retransmet si le ACK n'est pas reçu à temps
- Si le paquet (ou ACK) est en retard (pas perdu):
  - La retransmission sera dupliqué, mais l'utilisation d'un # seq. traite ça déjà
  - Le récepteur doit spécifier le # séq du paquet déjà acquitté
- Cela nécessite un temporisateur (timer)

Exemple

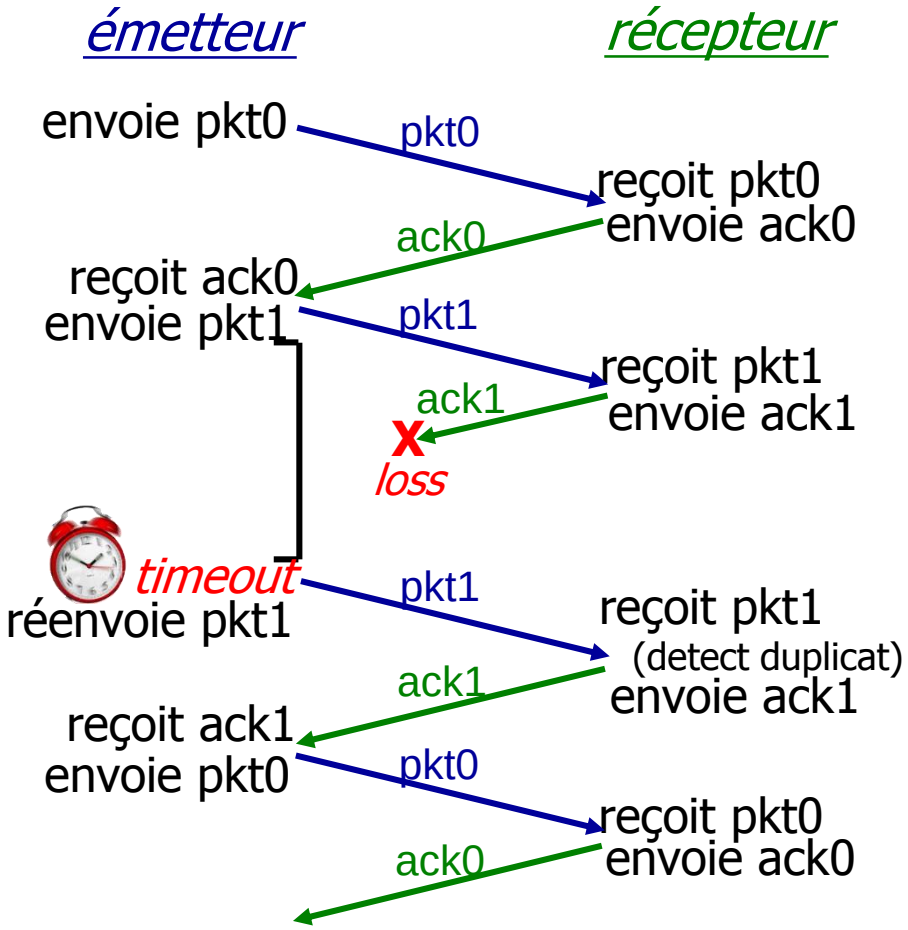


(a) pas de perte

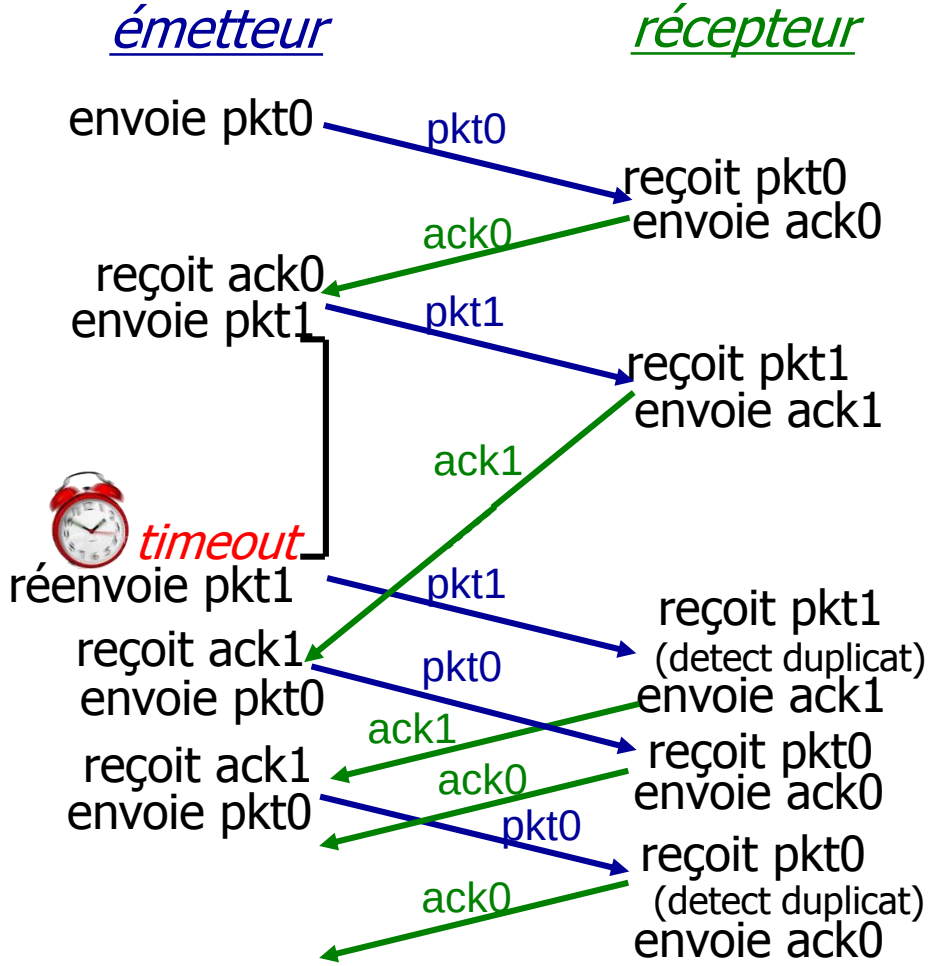


(b) perte d'un paquet

Exemple



(c) perte d'un ACK



(d) timeout prématuré / ACK en retard

Performance du protocole

- Le protocole marche, mais mauvaises performances
- ex.: lien de 1 Gbps, délai prop.15 ms, paquet 8000 bit:

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

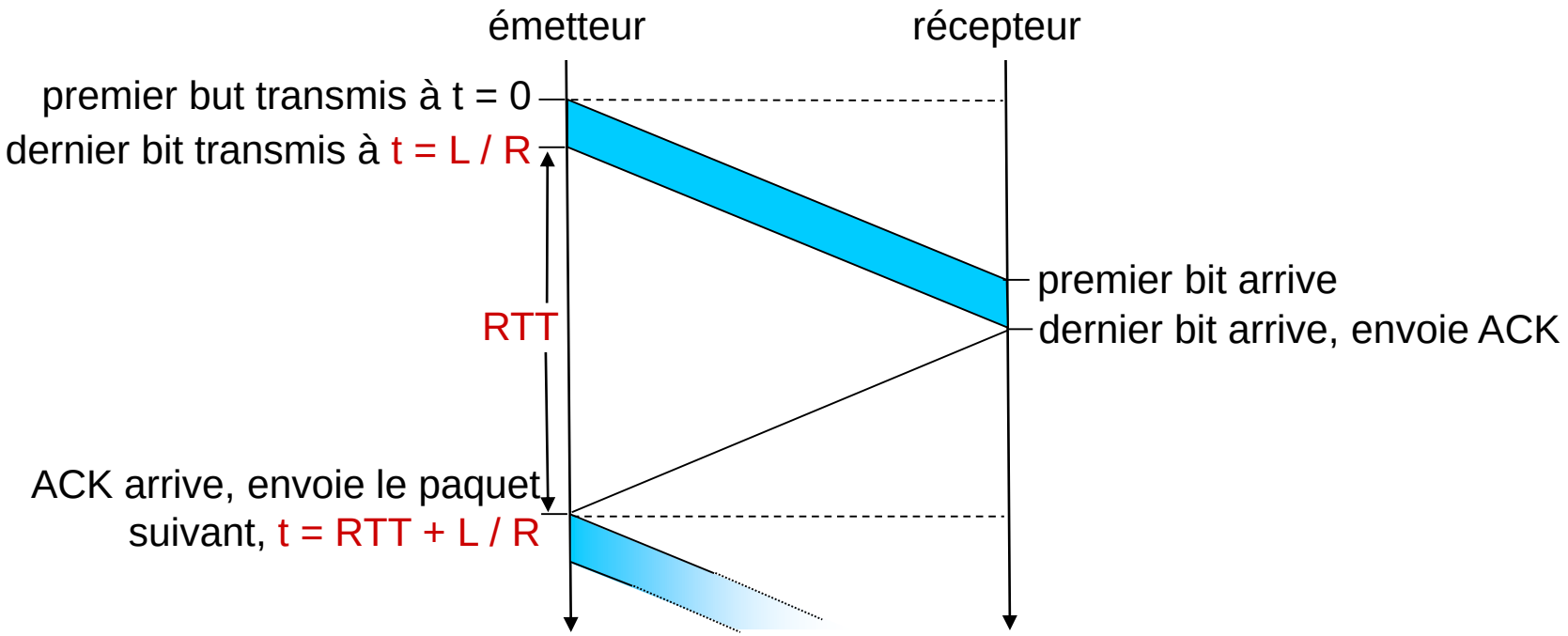
  - U_{sender}: *utilisation* – fraction du temps durant laquelle l'émetteur est occupé par la transmission

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

  - si RTT=30 msec, un paquet de 1KB chaque 30 msec: 33kB/sec débit sur un lien de 1 Gbps

- Le protocole réseau limite l'utilisation des ressources physiques!

Fonctionnement avec arrêt-et-attente

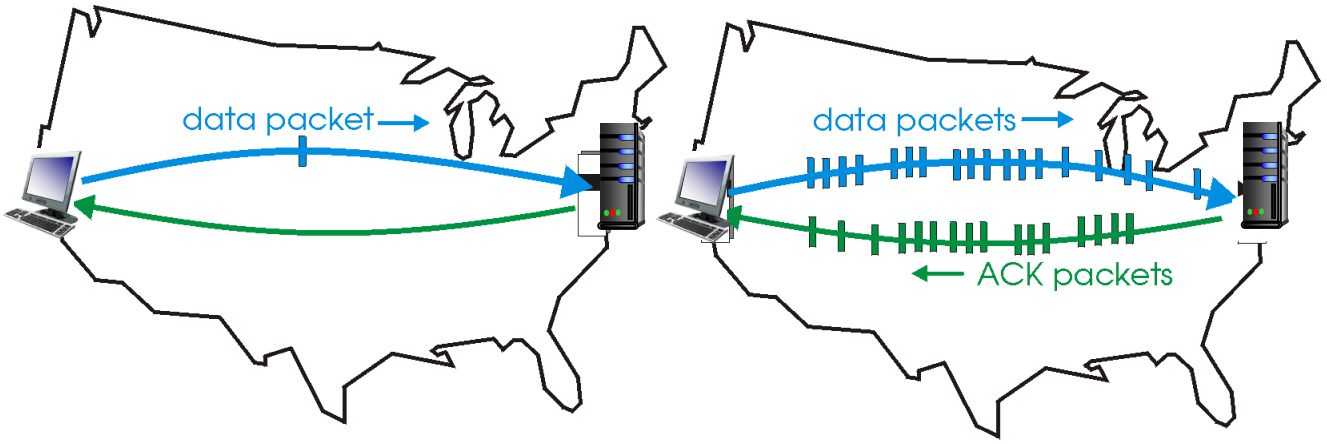


$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

Protocoles avec pipelining

pipelining: l'émetteur peut transmettre plusieurs paquets (sans attendre l'acquiescement)

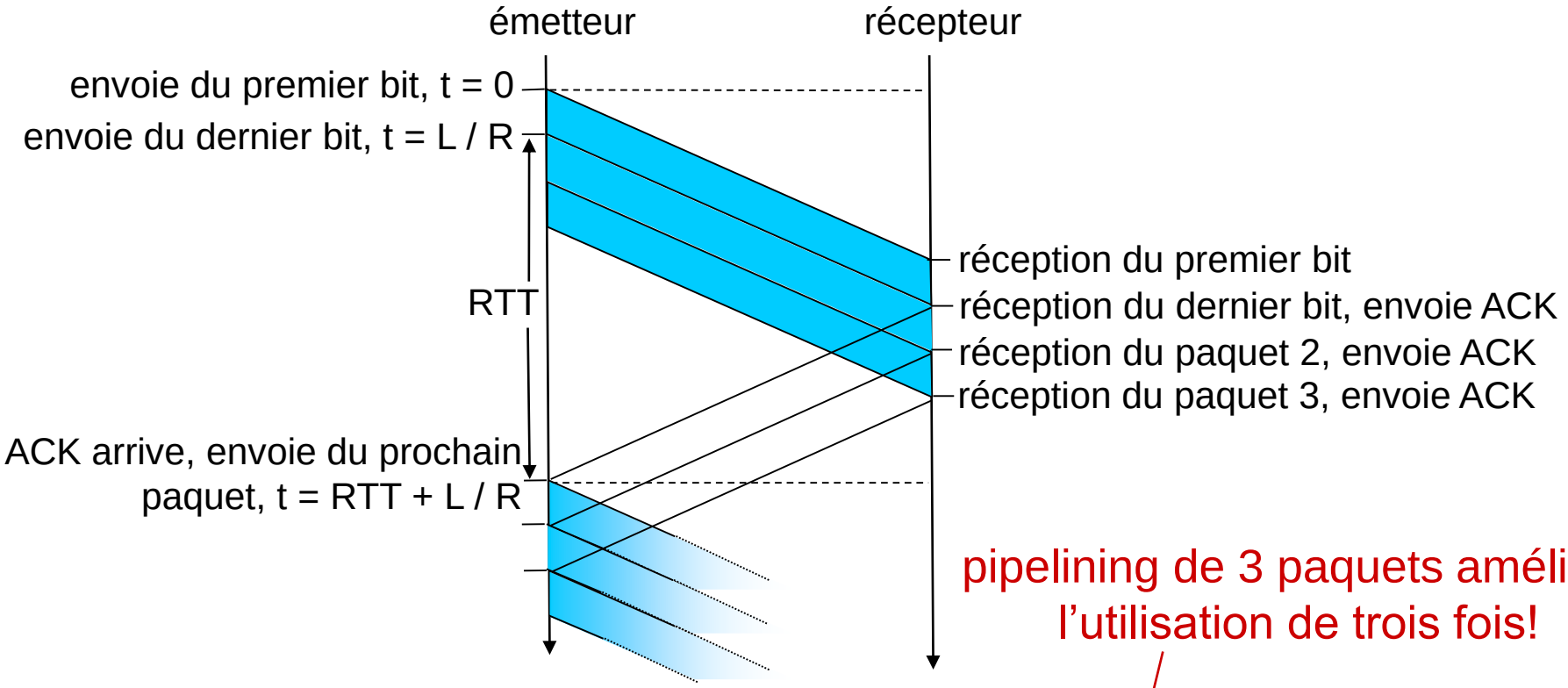
- plus de numéros de séquence
- mise en mémoire chez l'émetteur et le récepteur



(a) protocole arrêt et attente (b) protocole avec pipelining

- deux formes de protocoles de pipelining: *go-Back-N, selective repeat (répétition sélective)*

Pipelining: utilisation améliorée



pipelining de 3 paquets améliore l'utilisation de trois fois!

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

Protocoles avec pipelining

Go-back-N:

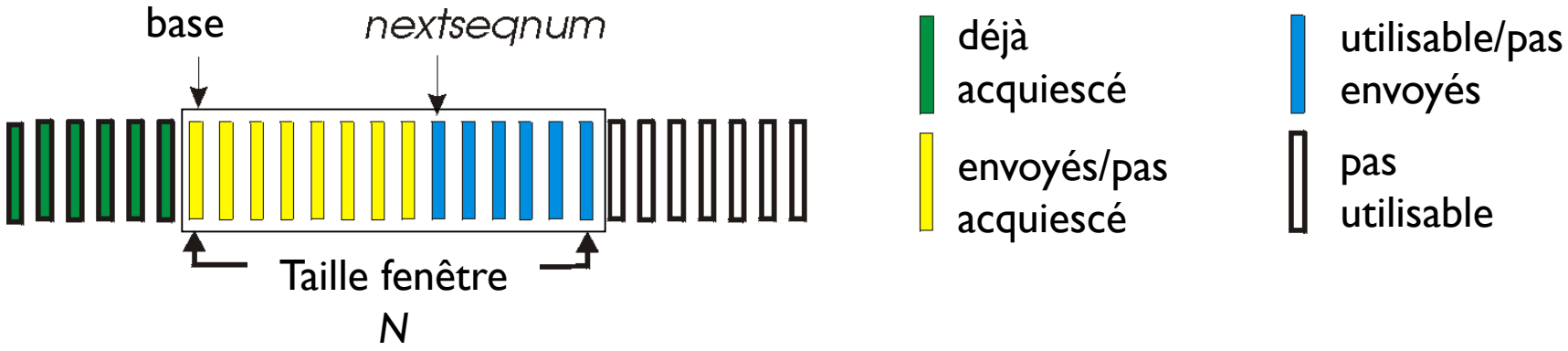
- L'émetteur peut avoir jusqu'à N paquets non acquiescés en pipeline
- Le récepteur envoie seulement des ACKs cumulatifs
  - N'acquiesce pas les paquets s'il y a un gap
- L'émetteur a un timer pour le plus ancien paquet non acquiescé
  - Si le timer expire, il retransmets tous les paquets non acquiescés

Répétition sélective:

- L'émetteur peut avoir jusqu'à N paquets non acquiescés en pipeline
- Le récepteur envoie des ACKs individuels
- L'émetteur a un timer pour chaque paquet non acquiescé
  - Si le timer expire, il retransmets tous les paquets non acquiescés

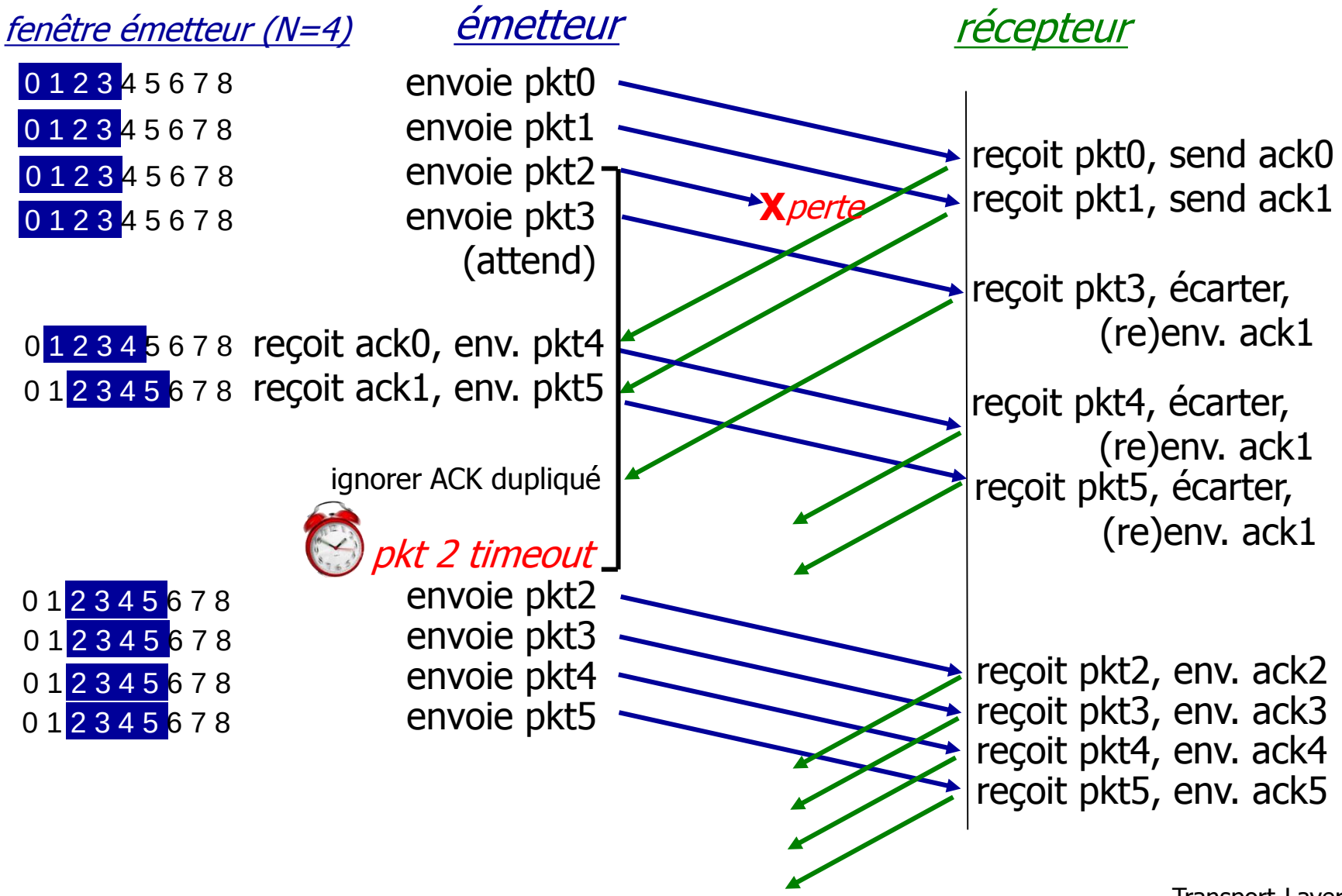
Go-Back-N: émetteur

- # seq de k-bit dans l'en-tête du paquet
- une "fenêtre" d'au plus N paquets consécutives non acquiescés



- ACK(n): acquiesce tous les paquets jusqu'au paquet n - *"ACK cumulatif"*
  - peut recevoir des ACKs dupliqués
- timer pour le paquet le plus ancien en transit
- *timeout(n):* retransmettre le paquet *n* ainsi que tous les paquets ayant un # seq plus élevé

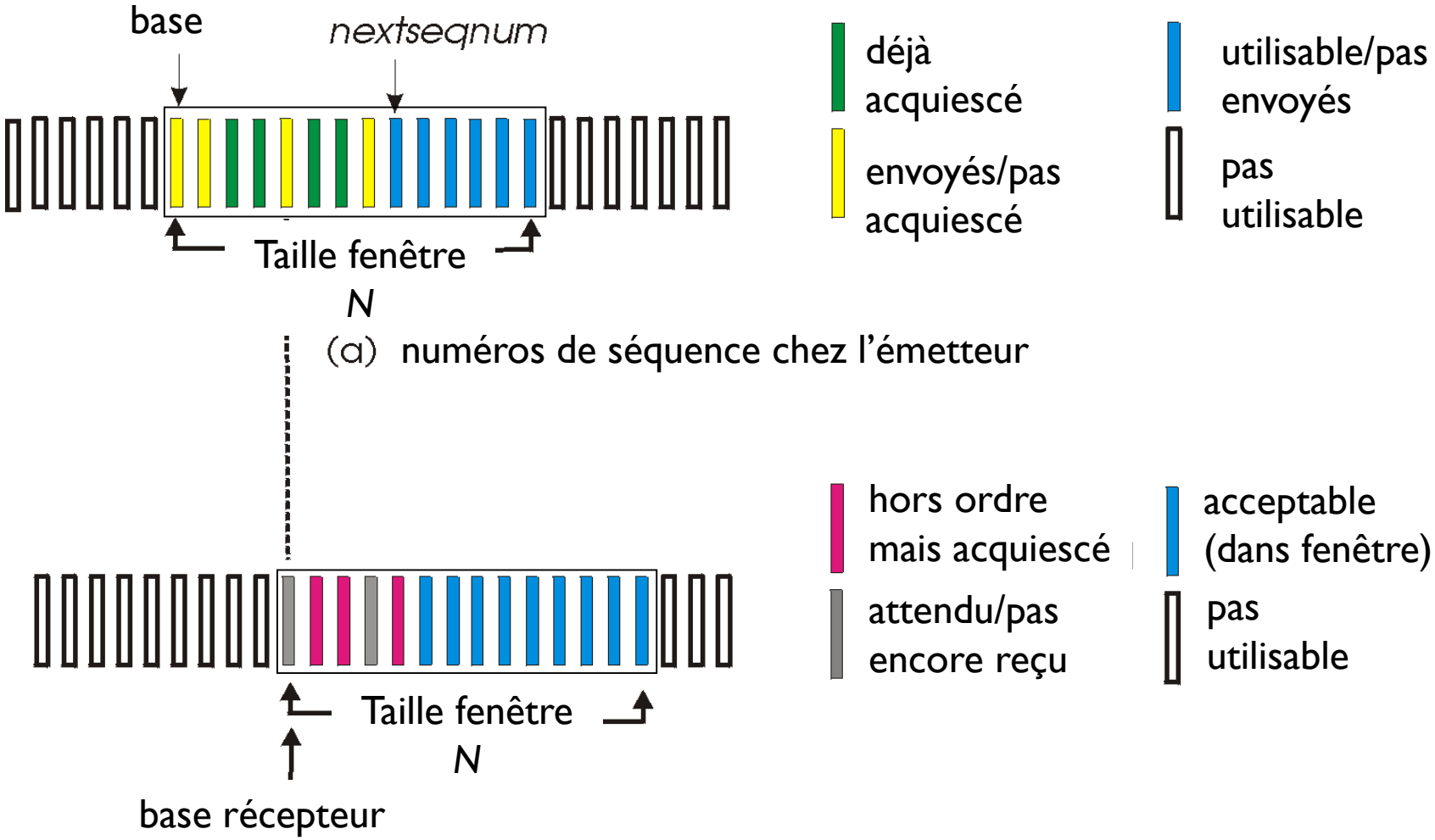
GBN en action



Répétition sélective

- le récepteur acquiesce chaque paquet correctement reçu
  - mets en mémoire les paquets, au besoin, pour les délivrer à la couche application dans le bon ordre
- L'émetteur renvoie seulement les paquets pour lesquels un ACK n'est pas reçu
  - Un timer de l'émetteur pour chaque paquet non acquiescé
- Fenêtre d'émission
  - N # séq. consécutives
  - limite les # séq. des paquets envoyés et des paquets non acquiescés

Répétition sélective: fenêtres



Répétition sélective

émetteur

données arrivent du haut:

- Si le prochain # séq dans la fenêtre, envoyer le paquet

timeout(n):

- renvoie paquet n, redémarre le timer

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

- marque paquet n comme reçu
- si n est le plus petit paquet non acquiescé, faire avancer la base de la fenêtre au prochain # séq non acquiescé

récepteur

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

- envoie ACK(n)
- hors-ordre: mémoriser
- Dans l'ordre: délivre (et délivrer les paquets mémorisés dans l'ordre), avancer la fenêtre au prochain paquet non reçu

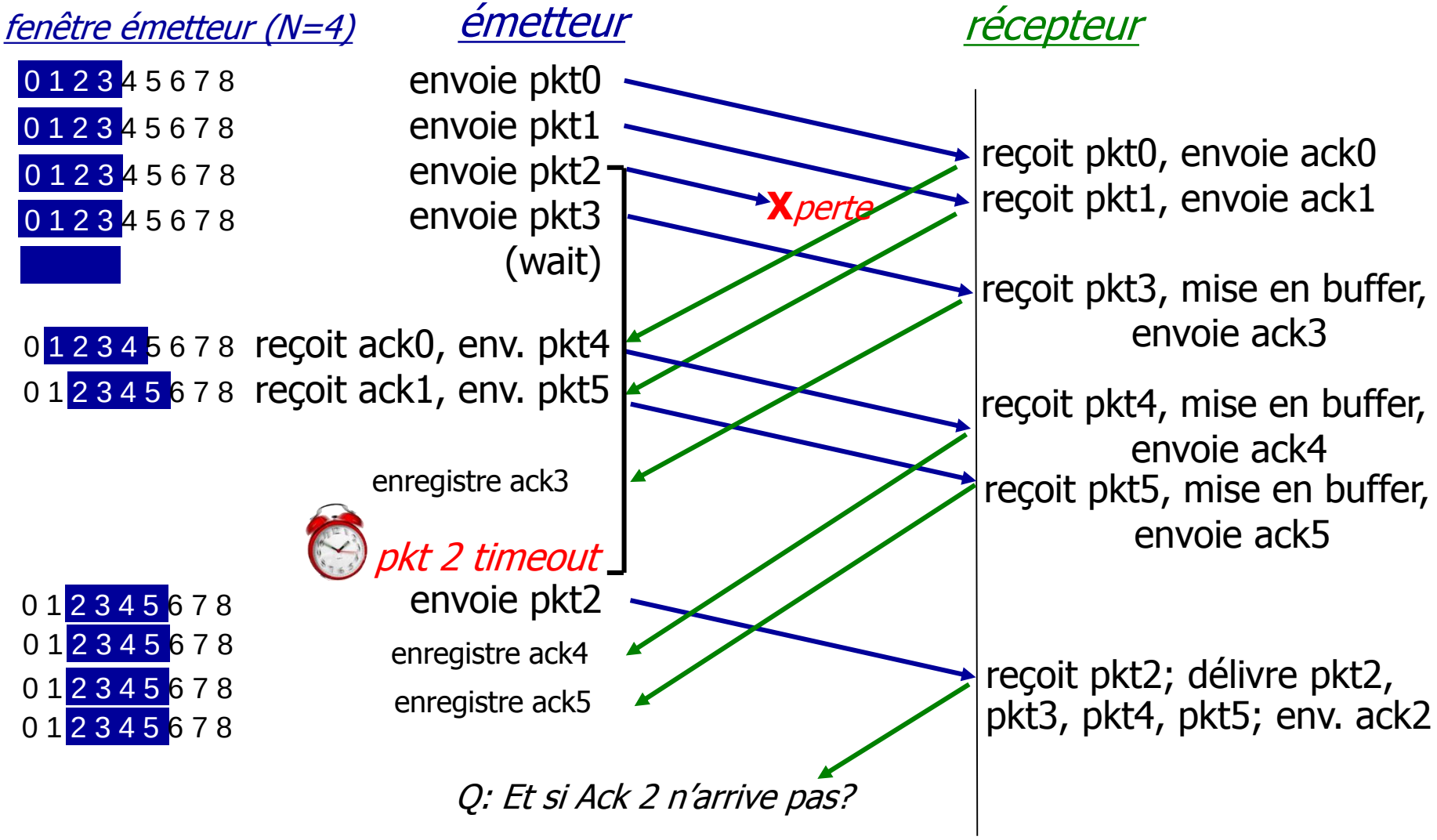
pkt n en [rcvbase-N,rcvbase-1]

- ACK(n)

Sinon:

- ignorer

La répétition sélective en action

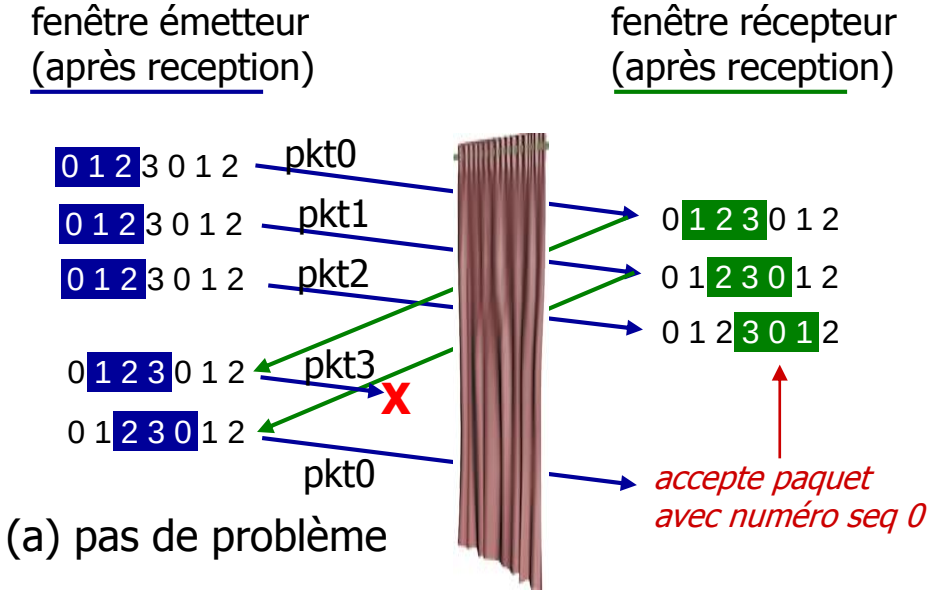


Répétition sélective dilemme

exemple:

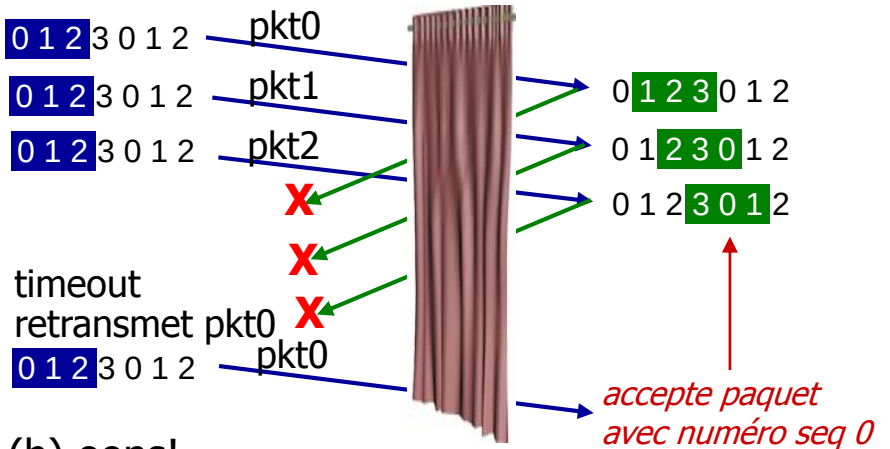
- # seq: 0, 1, 2, 3
- taille de fenêtre=3
- Le récepteur ne voit pas la différence dans les deux scénarios!
- passe des données dupliquées comme nouvelles en (b)

Q: quelle est la relation entre la taille des # séq. et la taille de la fenêtre pour corriger le problème dans (b)?



(a) pas de problème

le récepteur ne peut voir l'émetteur. Son comportement est le même pour les deux cas! problème sérieux!



(b) oops!