Cours Réseaux

Chapitre 3
Couche Transport

Université de Perpignan

Ouvrage de référence: *Analyse Structurée des Réseaux*, J. Kurose & K. Ross, Pearson Education, 2002

Chapitre 3: Couche Transport

Nos objectifs:

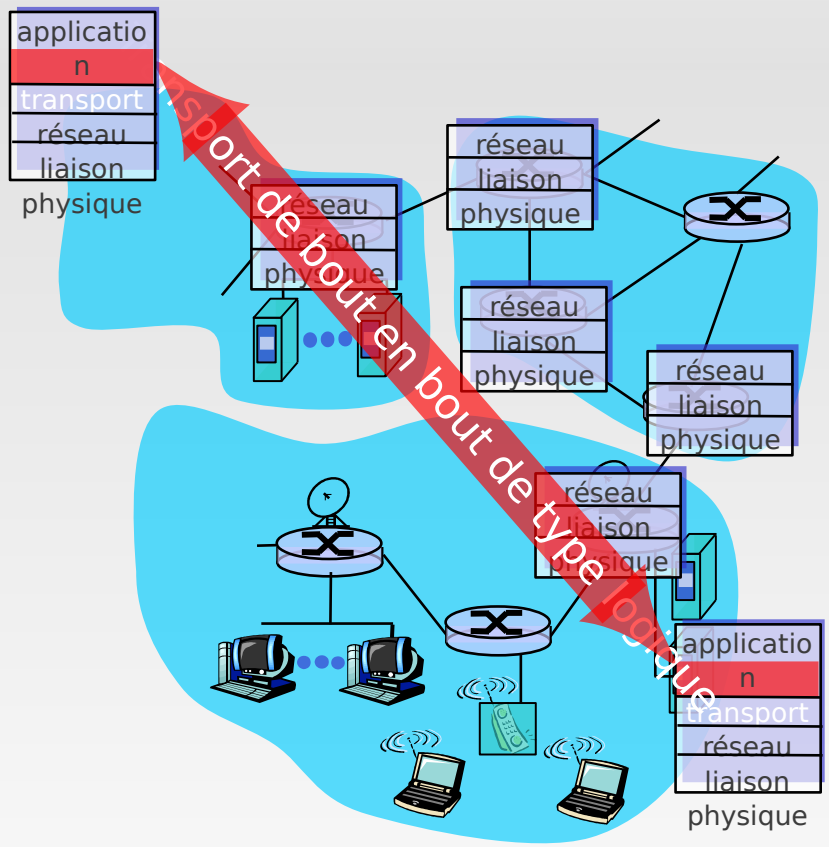
- Comprendre les principes derrière les services de la couche transport:
  - multiplexage/démultiplexage
  - transfert de donnée fiable
  - Contrôle de flux
  - Contrôle de congestion
- Apprendre les protocoles de la couche transport d'internet:
  - UDP: transport sans connexion
  - TCP: transport orienté connexion
  - contrôle de congestion de TCP

Chapitre 3: Plan

- 3.1 Services de la couche transport
- 3.2 Multiplexage et démultiplexage
- 3.3 Transport sans connexion: UDP
- 3.4 Principe du transfert de données fiable
- 3.5 Transport orienté connexion: TCP
  - Structure des segments
  - Transfert de données fiable
  - Controle de flux
  - Gestion de la connexion
- 3.6 contrôle de congestion
  - Principe
  - Controle de congestion de TCP
- 3.7 Performance de TCP

Services et protocoles de la couche Transport

- *La couche transport fournit une communication logique* entre processus tournant sur différents terminaux
- Les protocoles de transport s'exécutent sur les terminaux :
  - Coté expéditeur: il découpe les messages des processus en segments, et les passe à la couche réseau
  - Cote destinataire: il réassemble les messages à partir des segments et les fait passer à la couche application
- Plus d'un protocole de transport disponible :
  - Internet: TCP et UDP



Transport vs. Couche réseau

- *Couche réseau:* communication logique entre terminaux
- *Couche transport:* communication logique entre processus
  - Est liée à la couche réseau et étend les services de cette couche réseau

Analogie foyer:

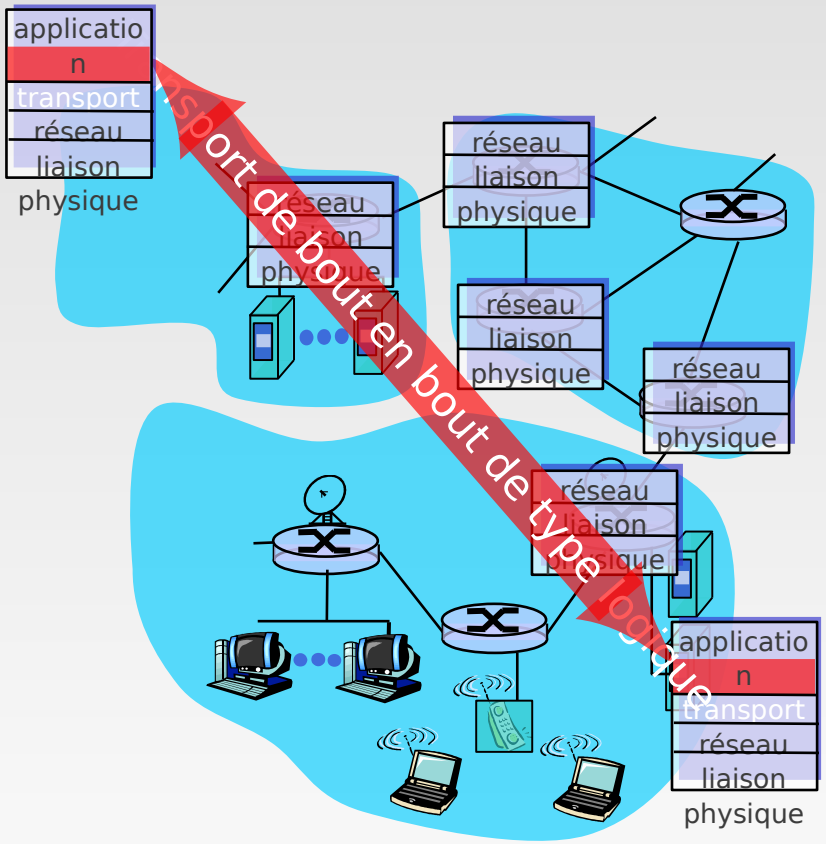
12 enfants envoient des lettres a 12 autres enfants

- processus = enfant
- messages des app = lettres dans des enveloppes
- terminaux = maisons
- protocole = Ann et Bill
- Protocole de la couche réseau = service postal

La couche transport : Introduction

Deux grands protocoles de transport :

- UDP (User Datagram Protocol) :
  - non fiable
  - sans connexion
- TCP (Transmission Control Protocol) :
  - fiable (contrôle de flux, de congestion, numéro de séquence, accusés de réception)
  - Convertit le service du protocole IP (service non fiable de la couche réseau) en un service fiable



Chapitre 3: Plan

- 3.1 Services de la couche transport
- 3.2 Multiplexage et démultiplexage
- 3.3 transport sans connexion: UDP
- 3.4 Principe du transfert de donnee fiable
- 3.5 Transport orienté connexion: TCP
  - Structure des segments
  - Transfert de données fiable
  - Controle de flux
  - Gestion de la connexion
- 3.6 contrôle de congestion
  - Principe
  - Controle de congestion de TCP
- 3.7 Performance de TCP

Multiplexage/démultiplexage

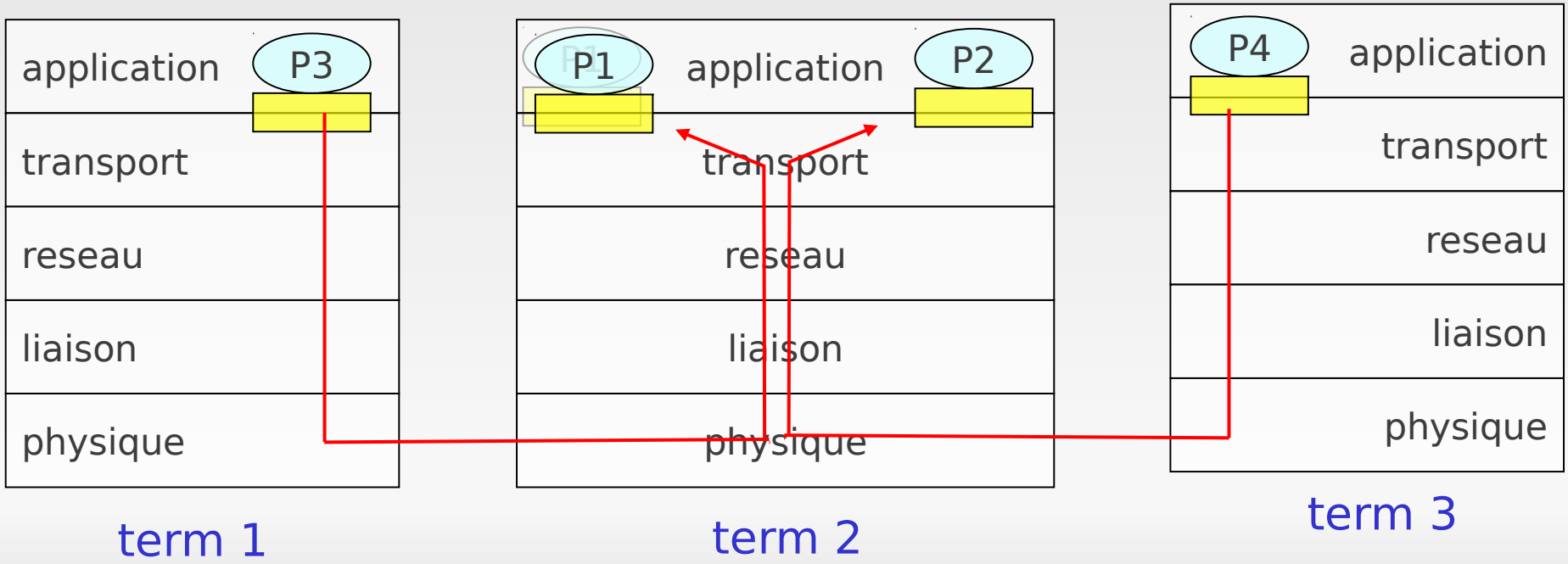
Démultiplexage sur le terminal dest.:

Délivre le segment à la socket correspondant au processus destinataire

Multiplexage sur le terminal exp.:

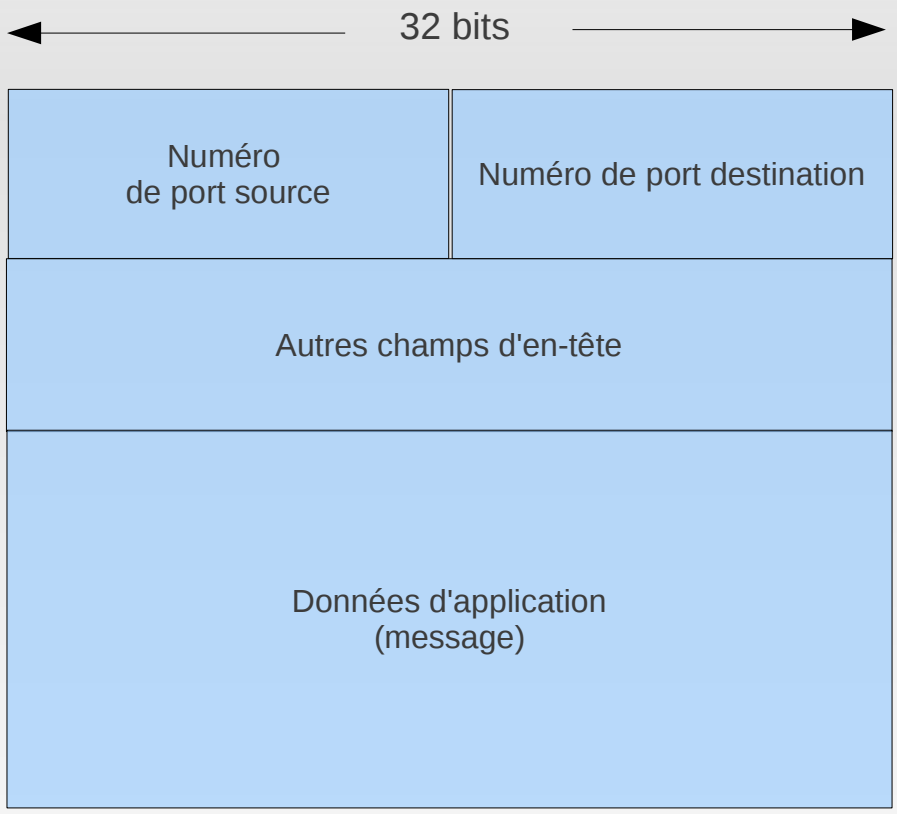
Rassemble les données provenant de multiple sockets, enveloppant les données dans une entête (utilisées plus tard pour le demultiplexage)

= socket = processus



Comment ca marche le démultiplexage

- Le terminal reçoit des datagrammes IP
  - Chaque datagramme a une adresse IP source et une adresse IP de destination
  - Chaque datagramme contient 1 segment de la couche transport
  - Chaque segment a un numéro de port source et un numéro de port de destination
- Le terminal utilise l'adresse IP et le numéro de port pour diriger le segment vers la socket appropriée



Format d'un segment TCP/UDP

Démultiplexage UDP (sans connexion)

- Crée des sockets avec un numéro de port:

```
DatagramSocket mySocket1 = new
   DatagramSocket(12534);
DatagramSocket mySocket2 = new
   DatagramSocket(12535);
```

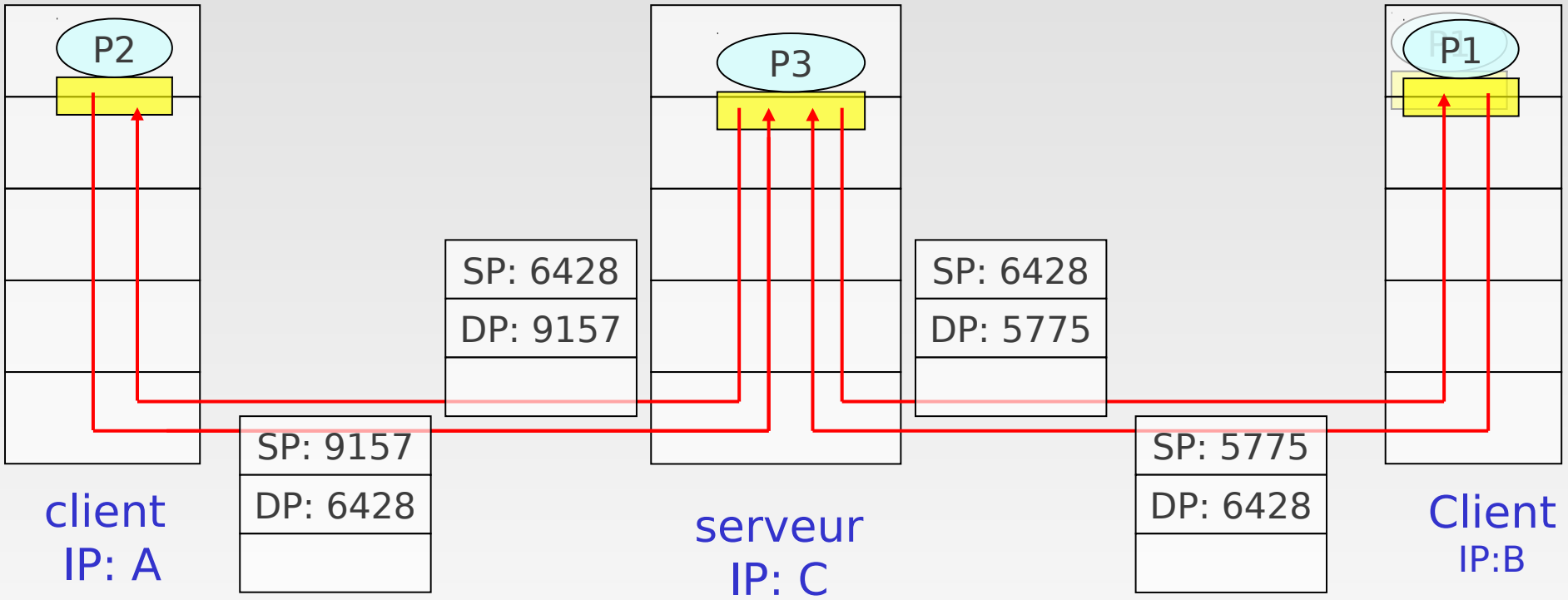
- Une socket UDP est identifiée par:

(adresse IP de dest., numéro de port de dest.)

- Quand un terminal reçoit un segment UDP, Il
  - vérifie le port de destination port dans les segments
  - dirige les segments UDP vers la socket avec ce numéro de port
- Des datagrammes IP avec des adresses IP sources et des numéros de port sources identiques sont redirigés vers la même socket

Démultiplexage UDP (cont.)

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

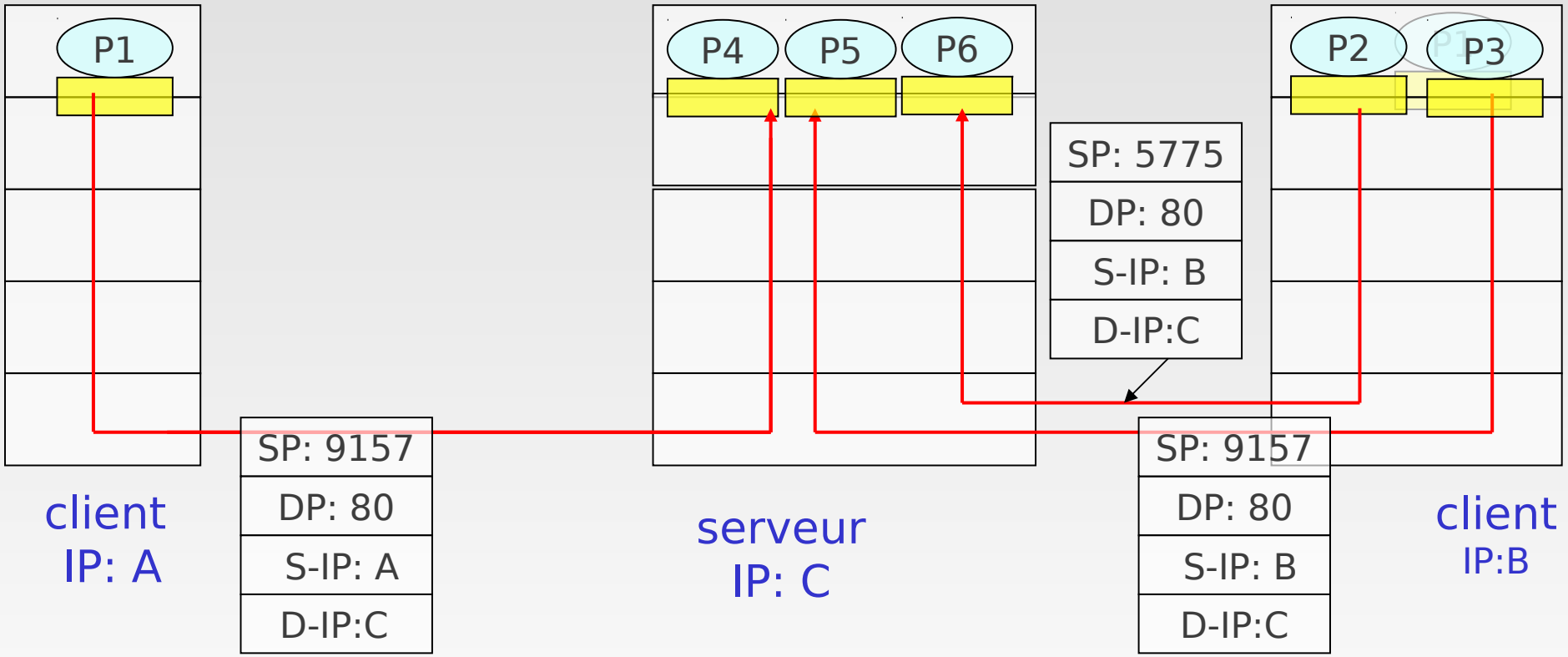


SP fournit “adresse de retour”

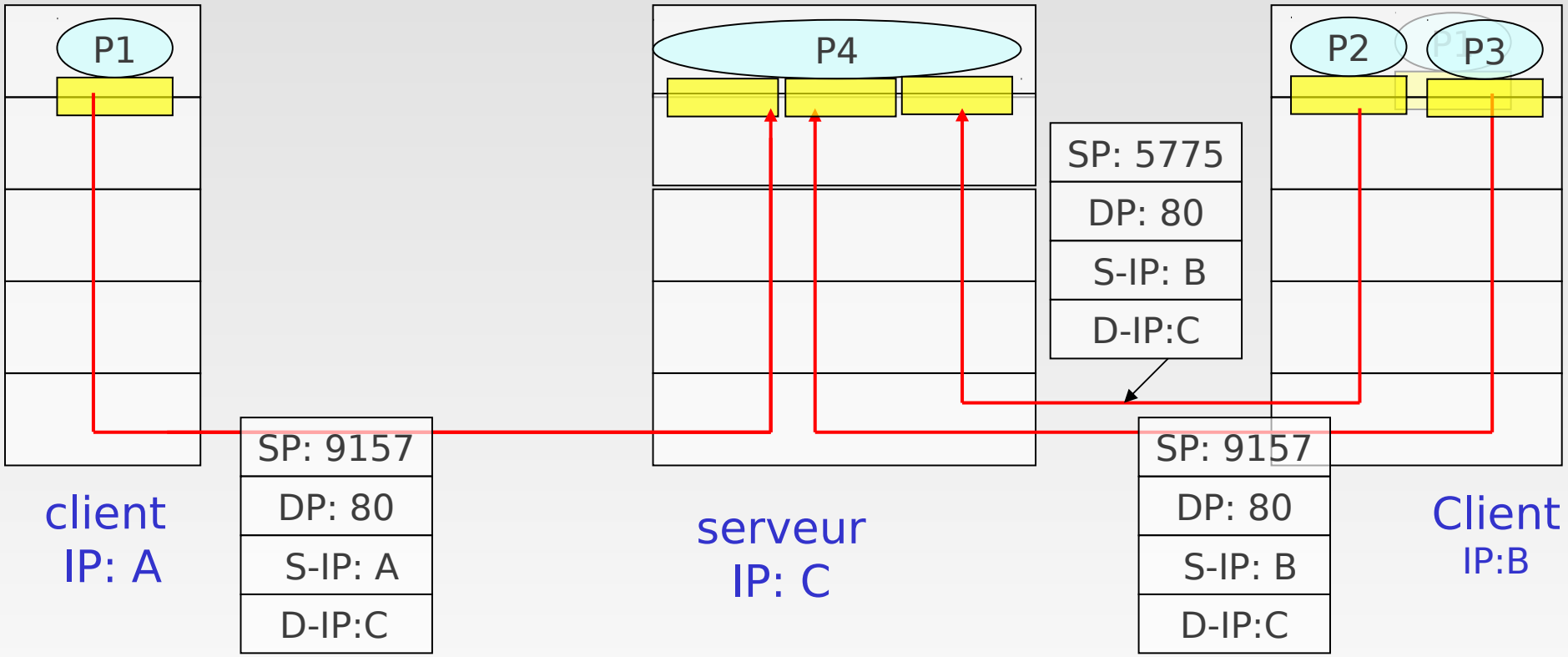
Démultiplexage TCP (orienté connexion)

- Socket TCP identifiée par un quadruplet:
  - Adresse IP source
  - Numéro de port source
  - Adresse IP de dest.
  - Numéro de port de dest.
- Le terminal destinataire utilise les quatre valeurs pour diriger le segment à la socket appropriée
- Un processus du terminal peut supporter de nombreuse socket TCP simultanément:
  - chaque socket est identifiée par son propre 4-tuple
- Les serveurs web ont des sockets différentes pour chaque connexion client
  - Une connexion HTTP non persistante aura différentes sockets pour chaque requête

Démultiplexage orienté connexion (cont.)



Démultiplexage orienté connexion: Web serveur threadé



Chapitre 3: Plan

- 3.1 Services de la couche transport
- 3.2 Multiplexage et démultiplexage
- 3.3 Transport sans connexion: UDP
- 3.4 Principe du transfert de données fiable
- 3.5 Transport orienté connexion: TCP
  - Structure des segments
  - Transfert de données fiable
  - Controle de flux
  - Gestion de la connexion
- 3.6 contrôle de congestion
  - Principe
  - Controle de congestion de TCP
- 3.7 Performance de TCP

UDP: User Datagram Protocol [RFC 768]

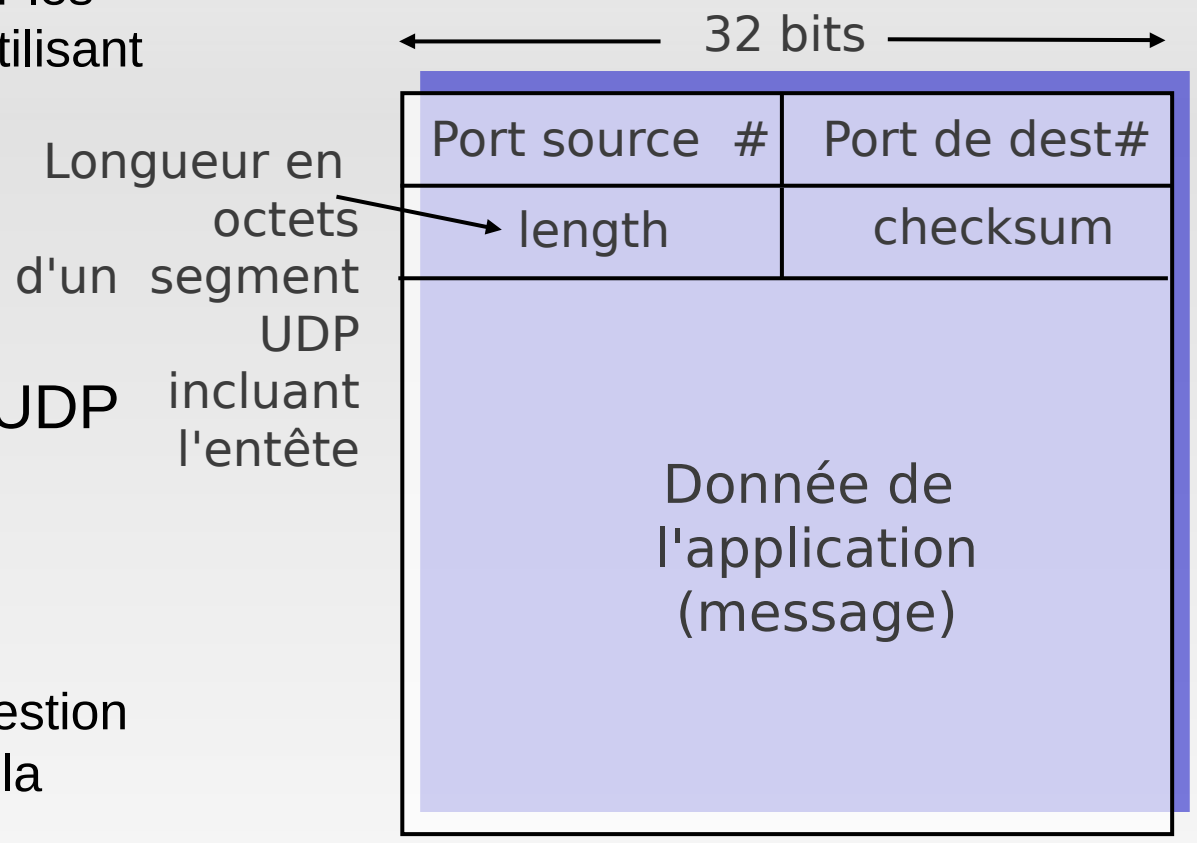
- *Protocole de transport Internet "sans fioritures"*
- Service "au meilleur effort", les segments UDP peuvent être:
  - perdu
  - Délivrés en désordre à l'application destinataire
- *Sans connexion:*
  - pas de "*poignée de mains*" ou "*présentation*" entre expéditeur et destinataire UDP
  - Chaque segment UDP est géré indépendement des autres.

Pourquoi utiliser UDP?

- Pas d'établissement de connexion (qui peut ajouter du délai)
- simple: pas de données à maintenir concernant la connexion (côte expéditeur et destinataire)
- Léger: petite entête
- Pas de contrôle de congestion: UDP peut envoyer des données aussi rapidement qu'on le souhaite

UDP: cont.

- UDP est souvent utilisé par les applications multimédias utilisant du *streaming*
  - Tolérant aux pertes
  - Sensible au taux de transmission
- D'autres utilisations d'UDP dans
  - DNS
  - SNMP
- Transfert basé sur UDP: gestion de la fiabilité au niveau de la couche application
  - Recouvrement d'erreur spécifique à l'application !



Format de segment UDP

La somme de contrôle (checksum) d'UDP

Objectif: détecter des “erreurs” (e.g., bits erronés) dans la transmission d'un segment

Expéditeur:

- Traite le contenu des segments comme une suite d'entiers codés sur 16-bit
- *checksum*: addition avec complément à 1 du contenu du segment
- L'expéditeur met la somme de contrôle (checksum) dans le champs “checksum” de l'entête UDP

Destinataire:

- Calcule la somme de contrôle du segment reçu
- Vérifie si la somme de contrôle calculée est égale à la valeur stockée dans le champs “checksum”:
  - NON - erreur détectée
  - OUI – pas d'erreur détectée. Mais *peut contenir des erreurs tout de même?* On verra ça plus tard ….

La somme de contrôle d'UDP

Expéditeur

0110011001100110
+ 0101010101010101
+ 0000111100001111
= 1100101011001010
0011010100110101

(complément à 1)

Destinataire

0110011001100110
+ 0101010101010101
+ 0000111100001111
= 1100101011001010
+ 0011010100110101
= 1111111111111111

(<=Ok)

= 1111111**0**1111111

(<=Erreur)

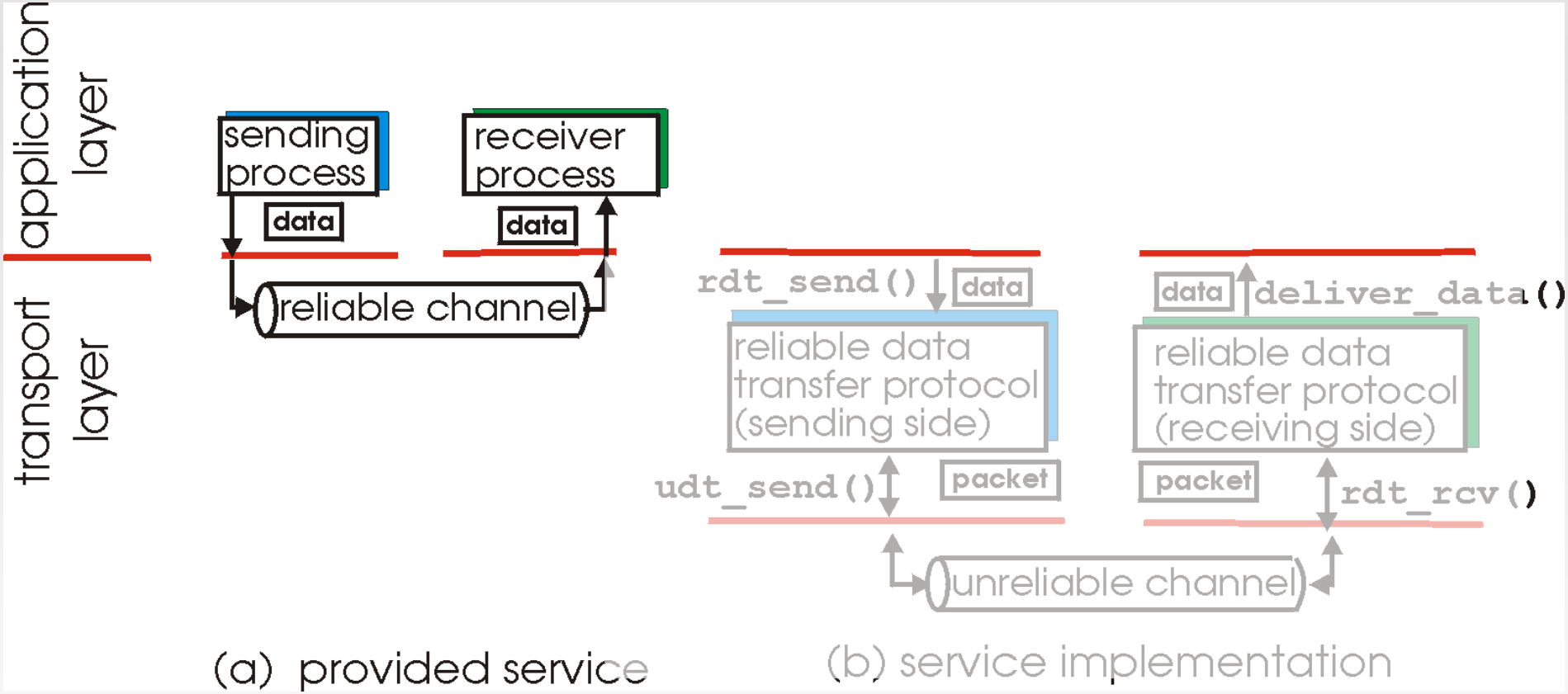
- *Remarque: les additions sont faites modulo 2^16*

Chapitre 3: Plan

- 3.1 Services de la couche transport
- 3.2 Multiplexage et démultiplexing
- 3.3 Transport sans connexion: UDP
- 3.4 Principe du transfert de données fiable
- 3.5 Transport orienté connexion: TCP
  - Structure des segments
  - Transfert de données fiable
  - Controle de flux
  - Gestion de la connexion
- 3.6 contrôle de congestion
  - Principe
  - Controle de congestion de TCP
- 3.7 Performance de TCP

Principes du transfert de données fiable

- Aspect important des couches application, transport et liaison.
- top-10 des sujets important concernant les réseaux!

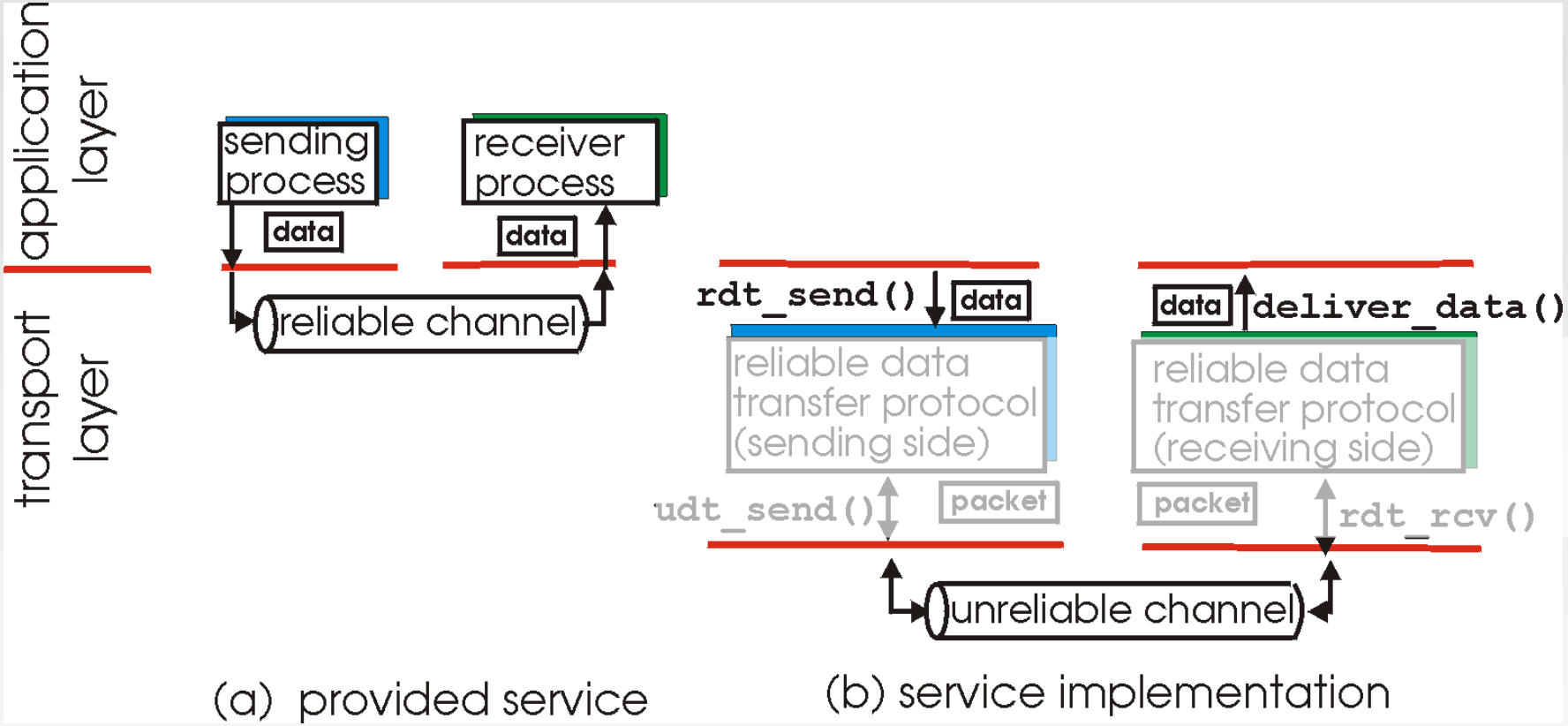


- Caractéristique d'un canal non fiable déterminera la complexité du transfert de données fiable.

  (rdt= reliable data transfert)

Principes du transfert de données fiable

- Aspect important des couches application, transport et liaison.
- top-10 des sujets important concernant les reseaux!

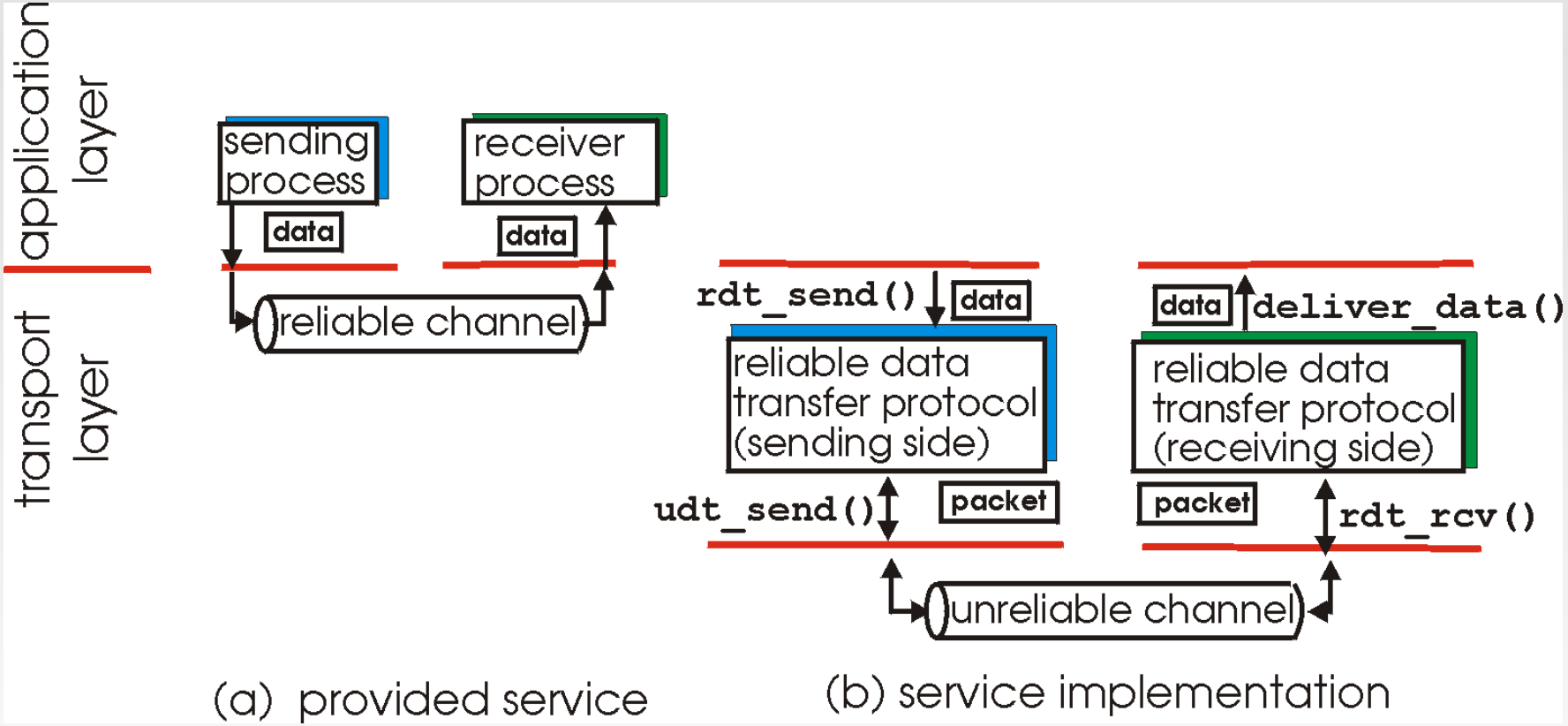


- Caractéristique d'un canal non fiable déterminera la complexité du transfert de données fiable.

  (rdt= reliable data transfert)

Principes du transfert de données fiable

- Aspect important des couches application, transport et liaison.
- top-10 des sujets important concernant les réseaux!



- Caractéristique d'un canal non fiable déterminera la complexité du transfert de données fiable.

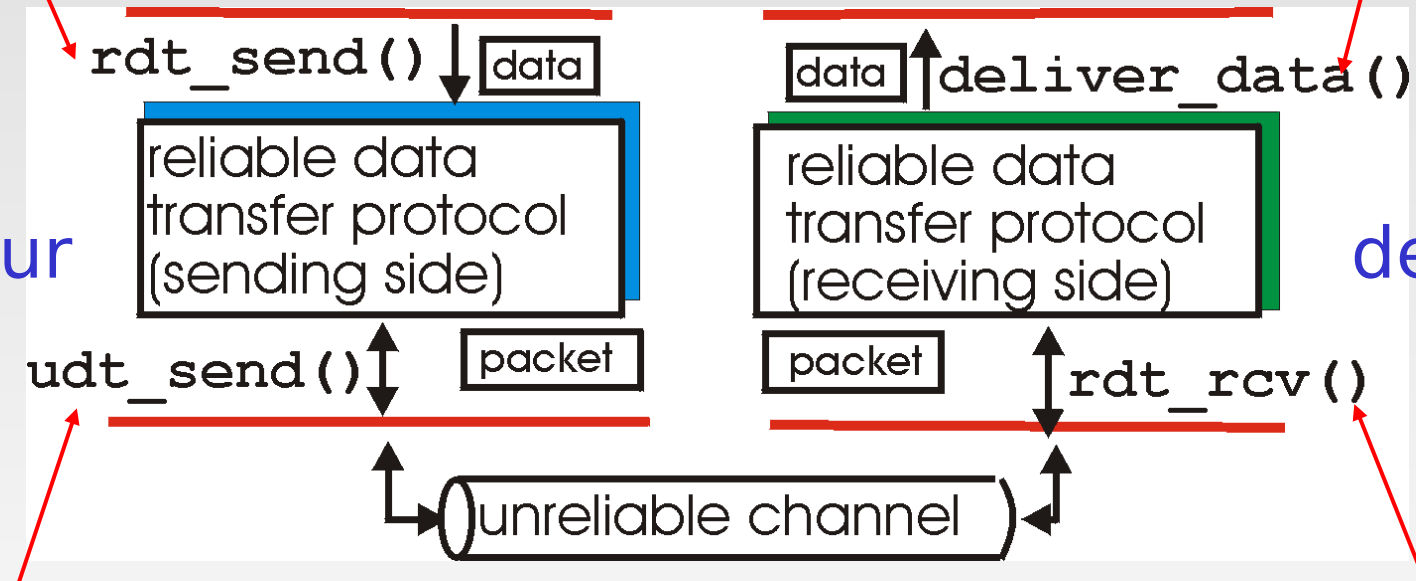
  (rdt= reliable data transfert)

Reliable data transfer: premier pas

rdt_send(): appel par le haut, (e.g., par l'appli.). Récupère les données provenant de la couche supérieure pour les envoyer au destinataire

deliver_data(): appelé par **rdt** pour délivrer les données à la couche supérieure

pôle expéditeur



pôle destinataire

udt_send(): appellé par rdt, pour transférer un paquet sur un canal non sûr au destinataire

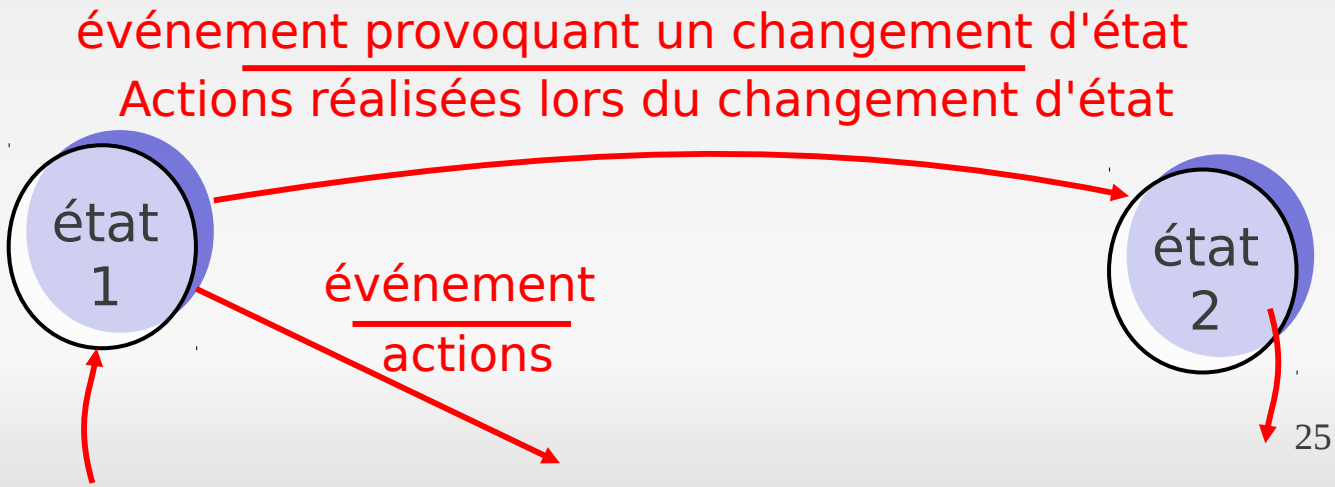
rdt_rcv(): appelle lorsqu'un paquet est reçu du côté destinataire

Transfert de données fiable: premier pas

Nous allons:

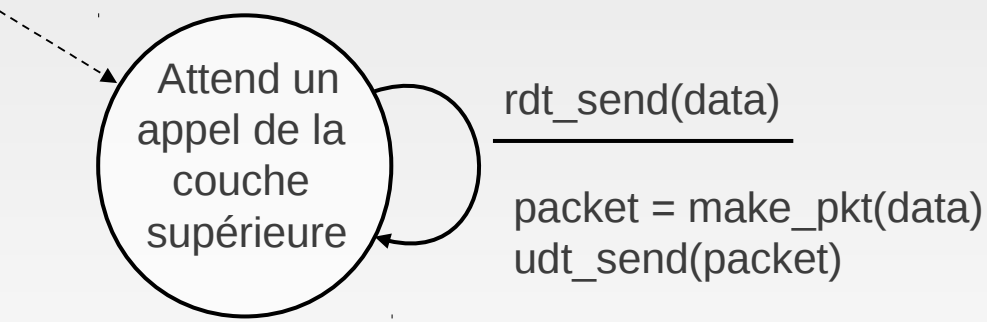
- Nous allons développer pas a pas les pôles expéditeur et destinataire d'un transfert de données fiable (rdt)
- Considérons seulement un transfert de données unidirectionnel
  - Mais les informations de contrôle navigueront dans les deux directions!
- On modélisera le *rdt* via machine à état fini (FSM=Finite State Machine) pour l'expéditeur et le destinateur

État: lorsque l'on est dans cet “état” le prochain état est déterminé uniquement par le prochain événement

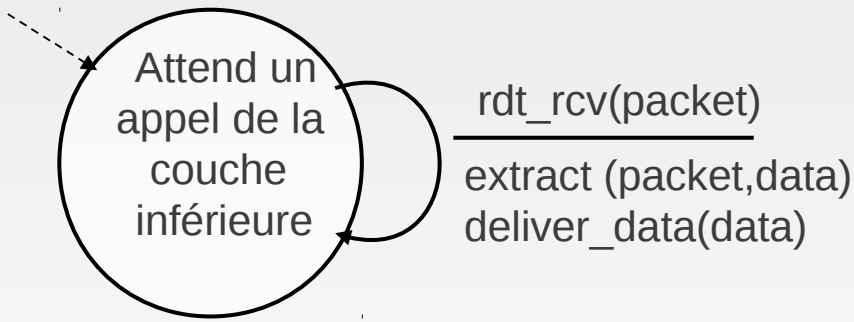


Rdt1.0: transfert fiable à travers un canal fiable

- Le canal de transmission est parfaitement fiable
  - Pas d'erreurs sur les bits lors de la transmission
  - Pas de perte de paquet
- Séparer FSMs pour l'expéditeur et destinataire:
  - L'expéditeur envoie des données dans le canal
  - Le destinataire lit les données sortant du canal



Expéditeur



Destinataire

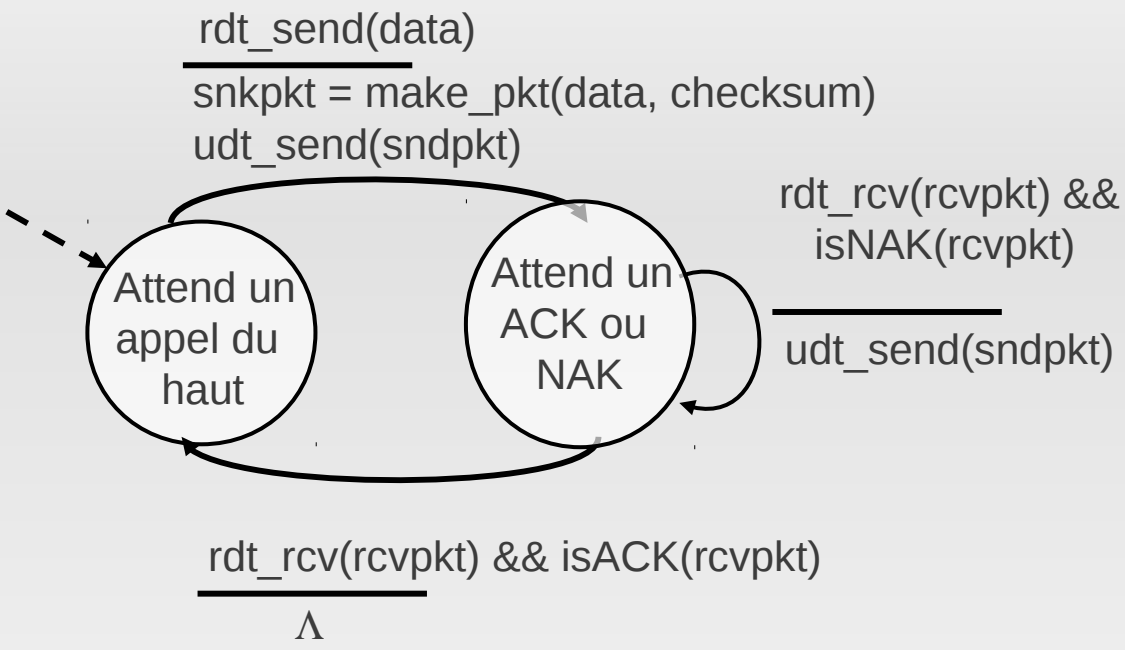
Rdt2.0: canal avec des erreurs

- Le canal de transmission peut modifier certains bits dans les paquets
  - checksum pour détecter des erreurs eventuelles
- *LA* question: comment recupérer des paquets erronés:
  - *Accusé de réception/validité (Note ACKs=acknowledgements ():* le destinataire dit explicitement à l'expéditeur que le paquet reçu était correct
  - *Accusé de réception négatif, d'invalidité (NAKs):* le destinataire dit explicitement à l'expéditeur que le paquet contenait des erreurs
  - L'expéditeur retransmet pkt (=paquet) lors d'une réception d'un NAK
- Nouveau mécanisme dans **`rdt2.0`** (au delà de **`rdt1.0`**):
  - détection d'erreur
  - Le destinataire répond par des messages de contrôle (ACK,NAK) expéditeur->destinaire

Rdt2.0: canal avec des erreurs

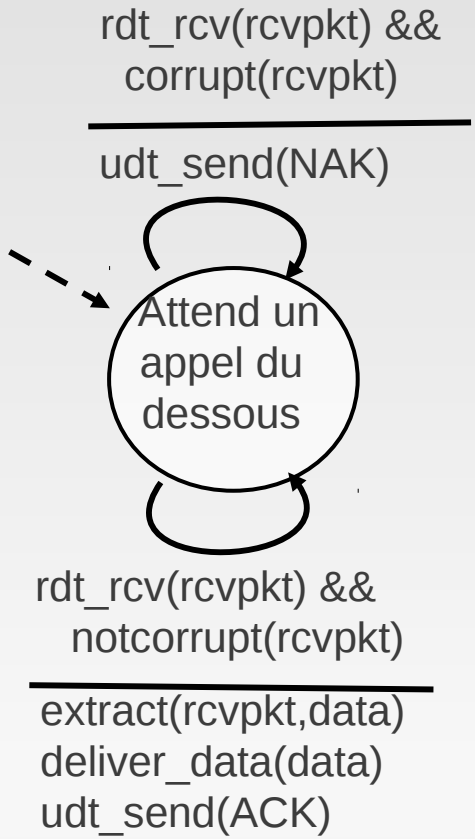
- Le canal de transmission peut modifier certains bits dans les paquets
  - checksum pour détecter des erreurs éventuelles
- *LA* question: comment récupérer des paquets erronés:
  - *Accusé de réception/validité (Note ACKs=acknowledgements ():* le destinataire dit explicitement à l'expéditeur que le paquet reçu était correct
  - *Accusé de réception négatif, d'invalidité (NAKs):* le destinataire dit explicitement à l'expéditeur que le paquet contenait des erreurs
  - L'expéditeur retransmet pkt (=paquet) lors d'une réception d'un NAK
- Nouveau mécanisme dans **`rdt2.0`** (au delà de **`rdt1.0`**):
  - détection d'erreur
  - Le destinataire répond par des messages de contrôle (ACK,NAK) expéditeur->destinaire

rdt2.0: spécification du FSM

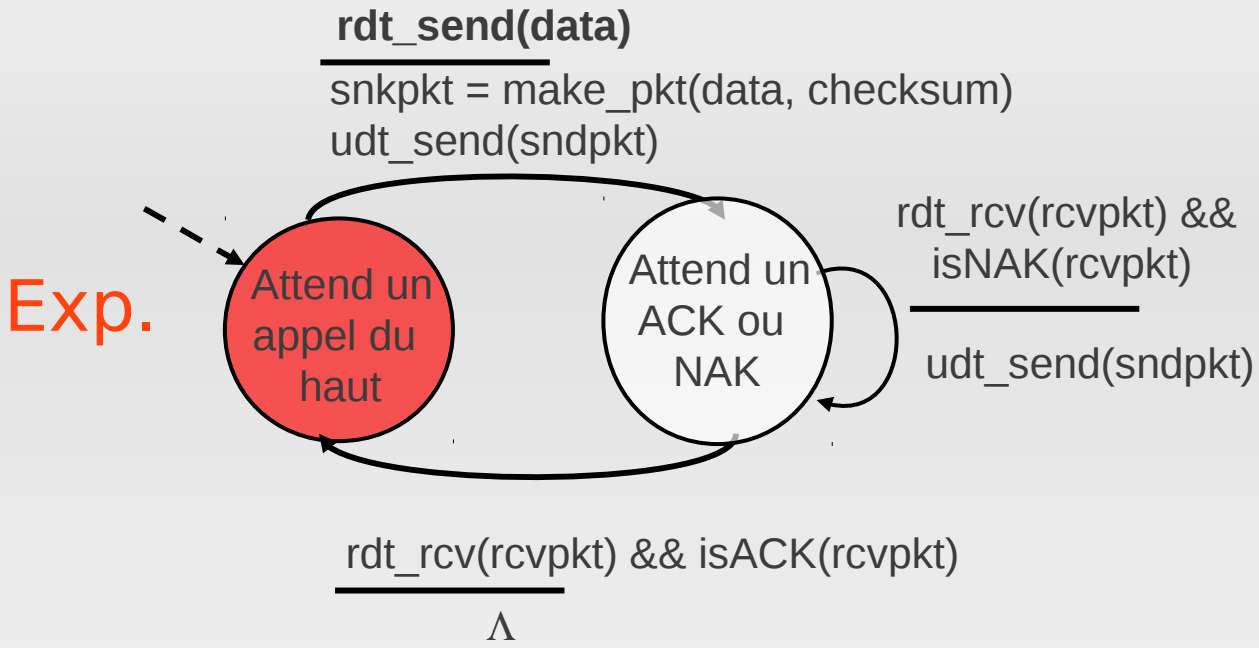


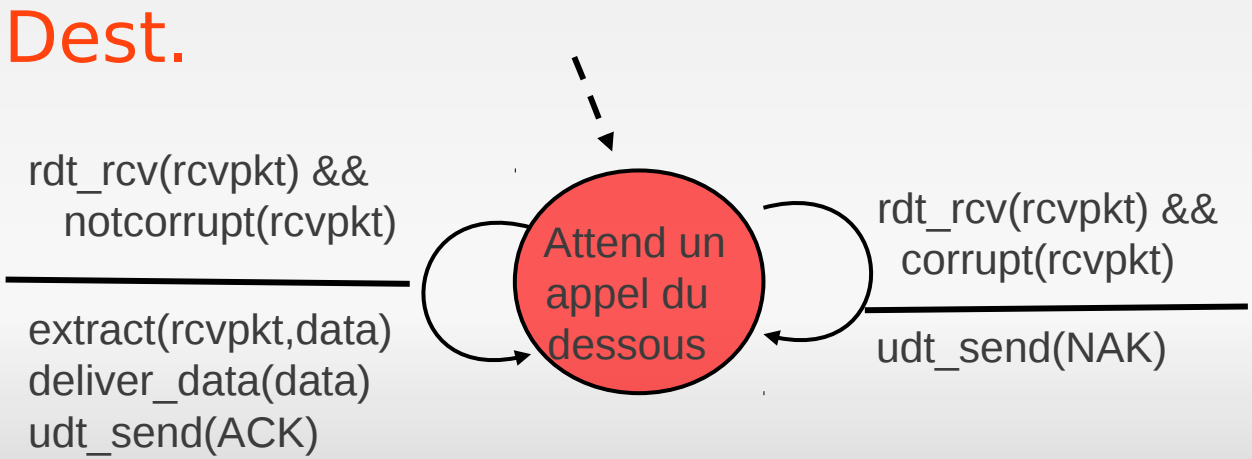
expéditeur

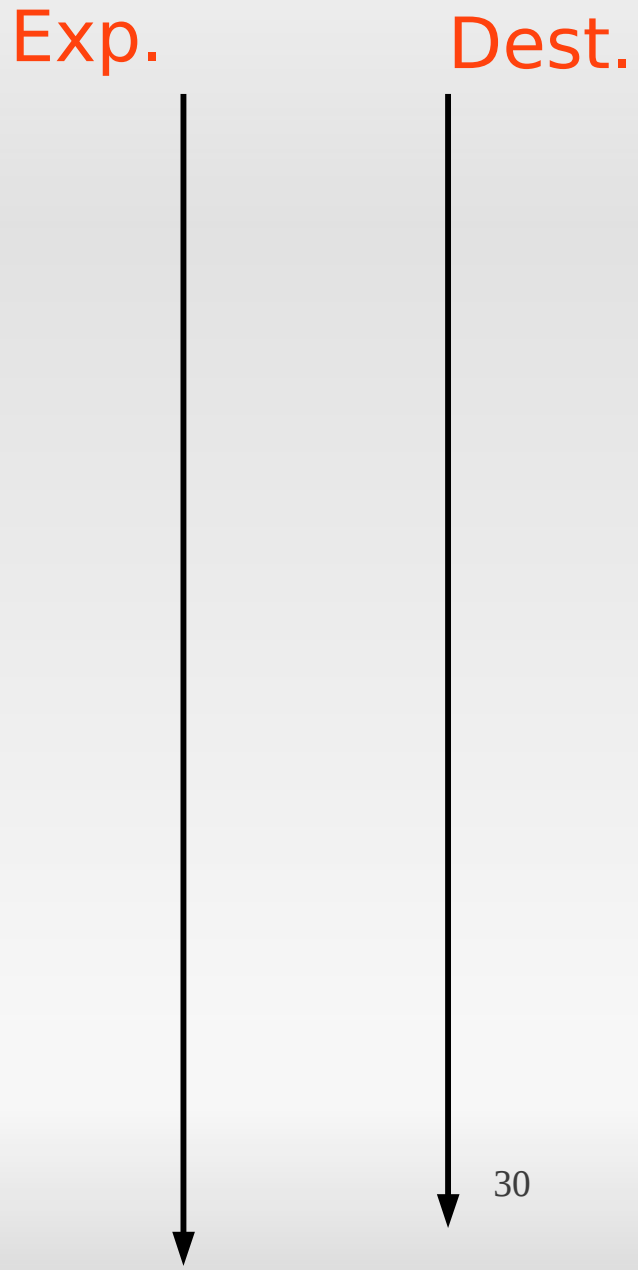
destinataire



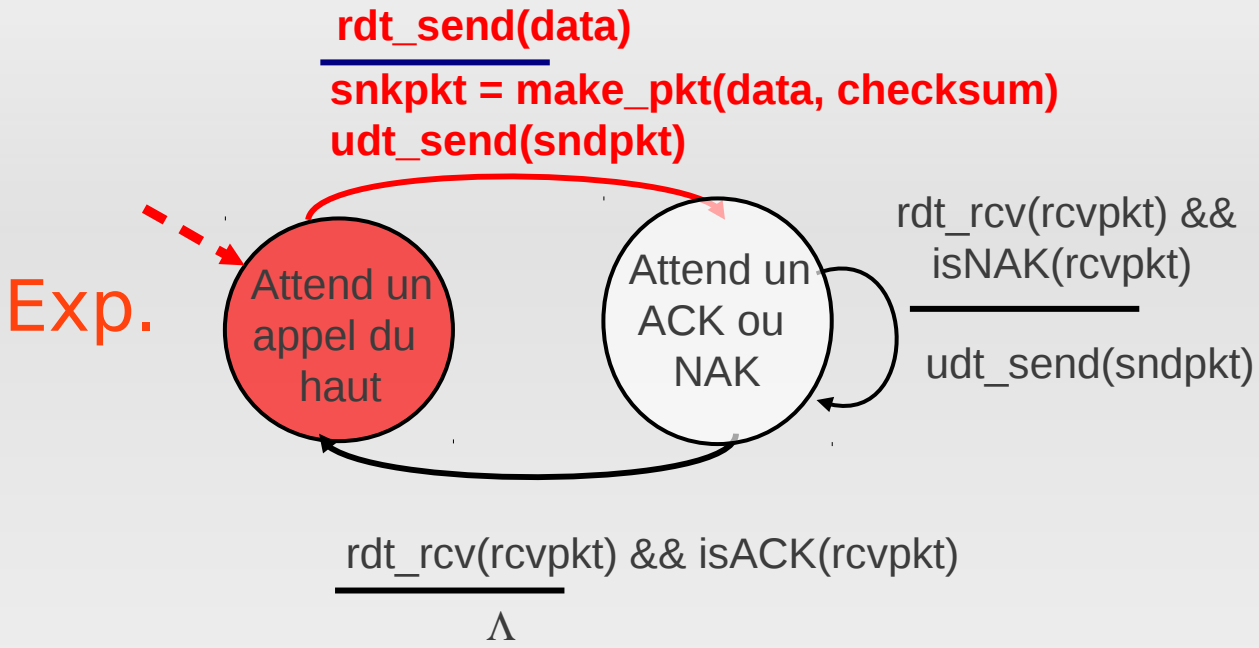
rdt2.0: exemple de déroulement (1/10)

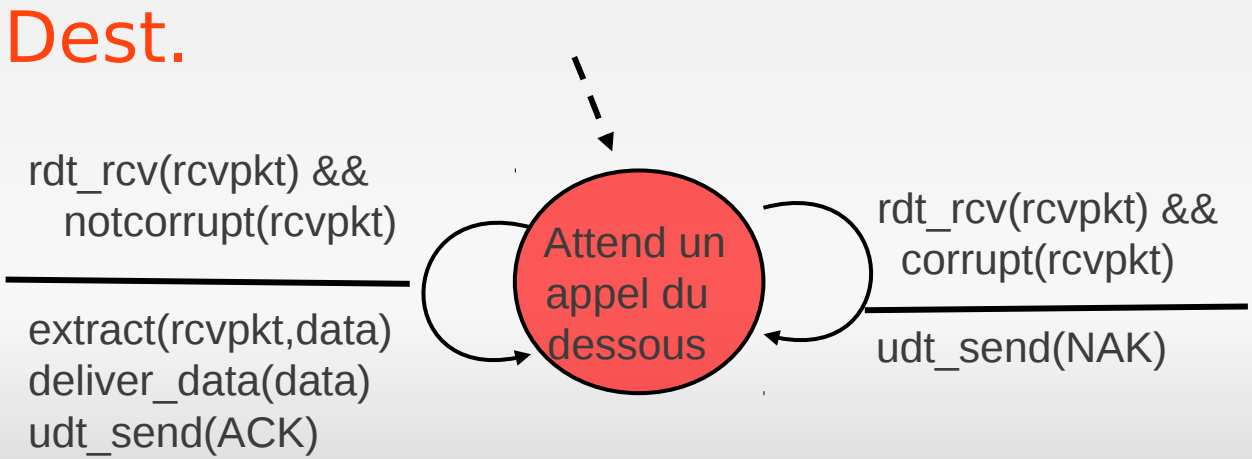


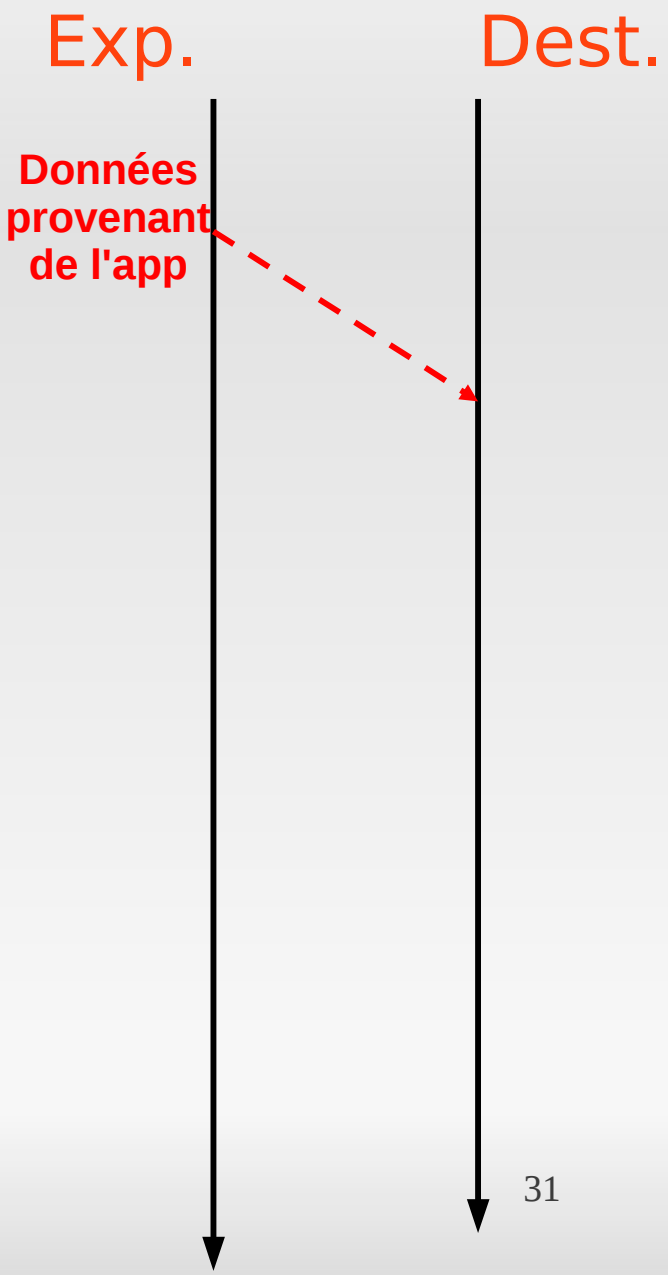




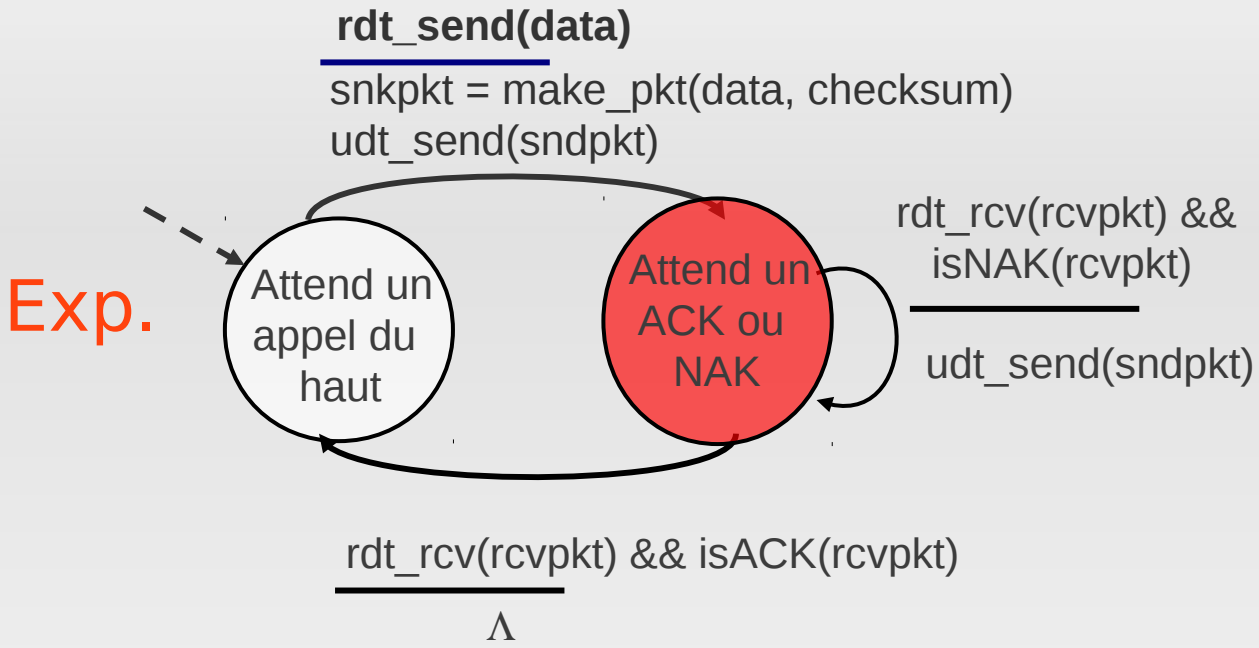
rdt2.0: exemple de déroulement (2/10)

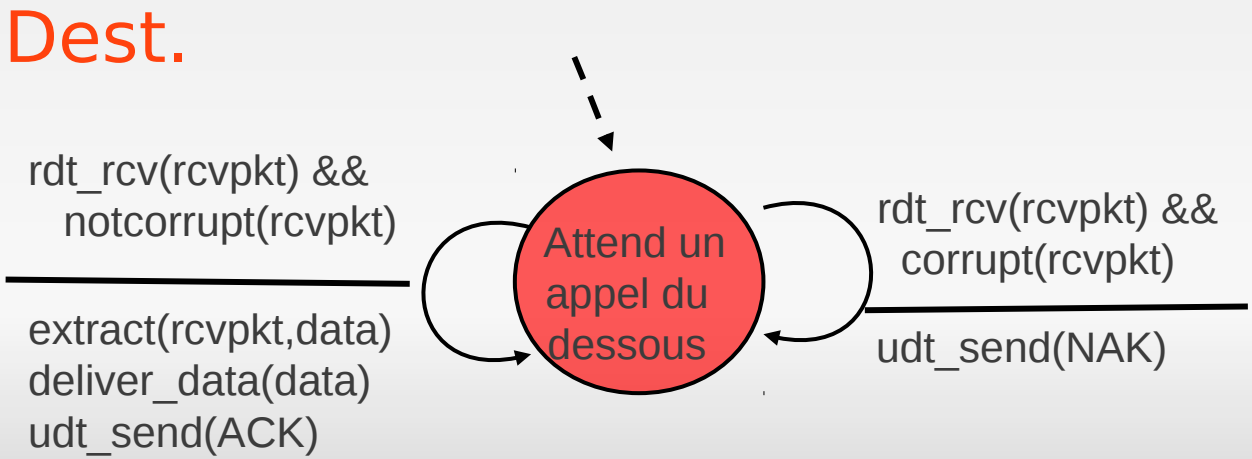


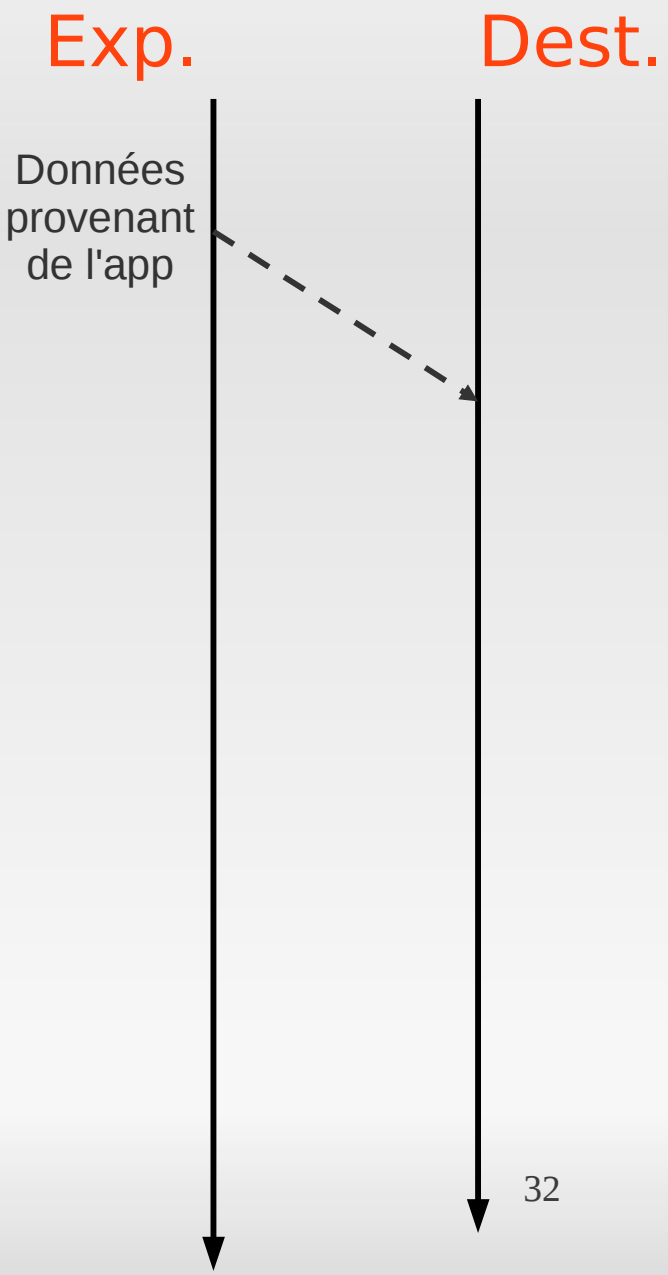




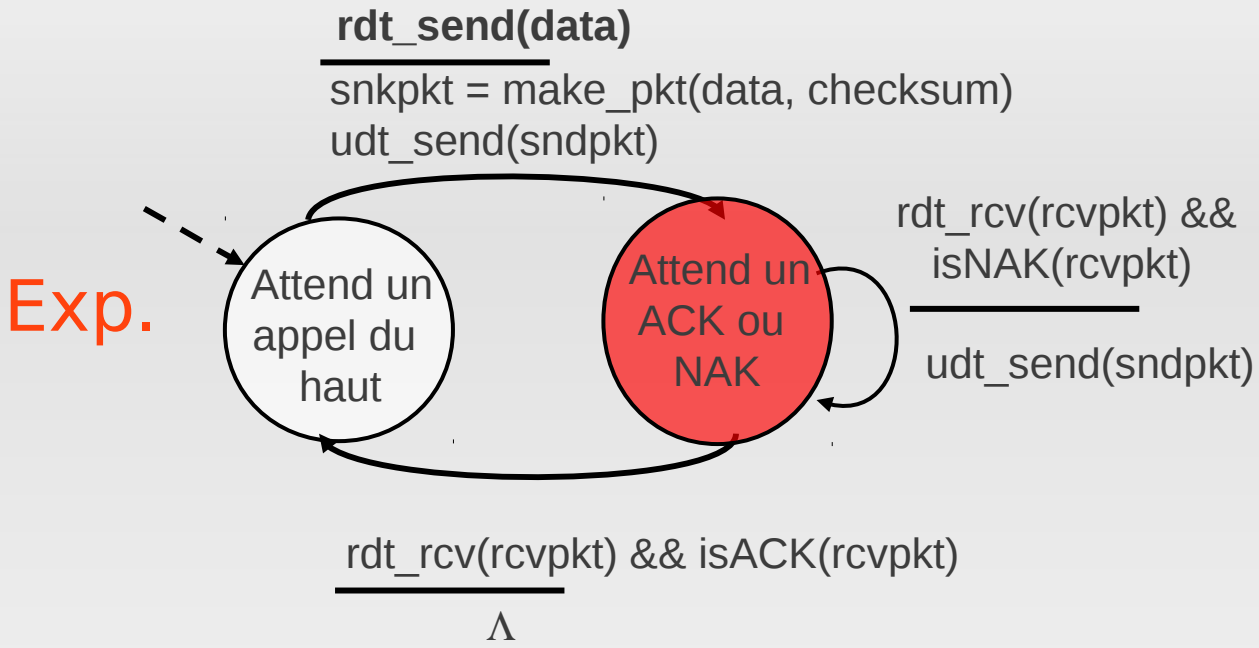
rdt2.0: exemple de déroulement (3/10)

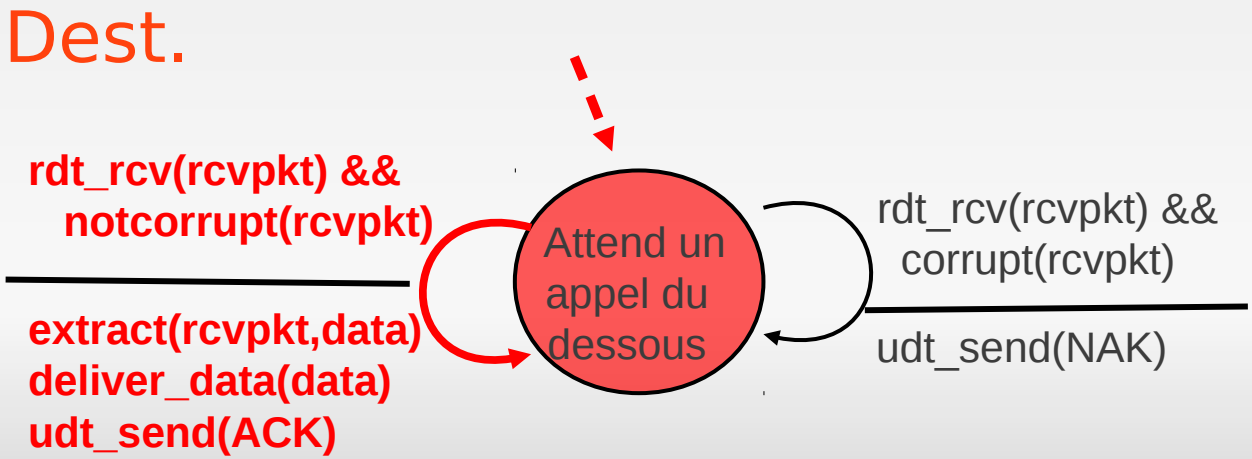


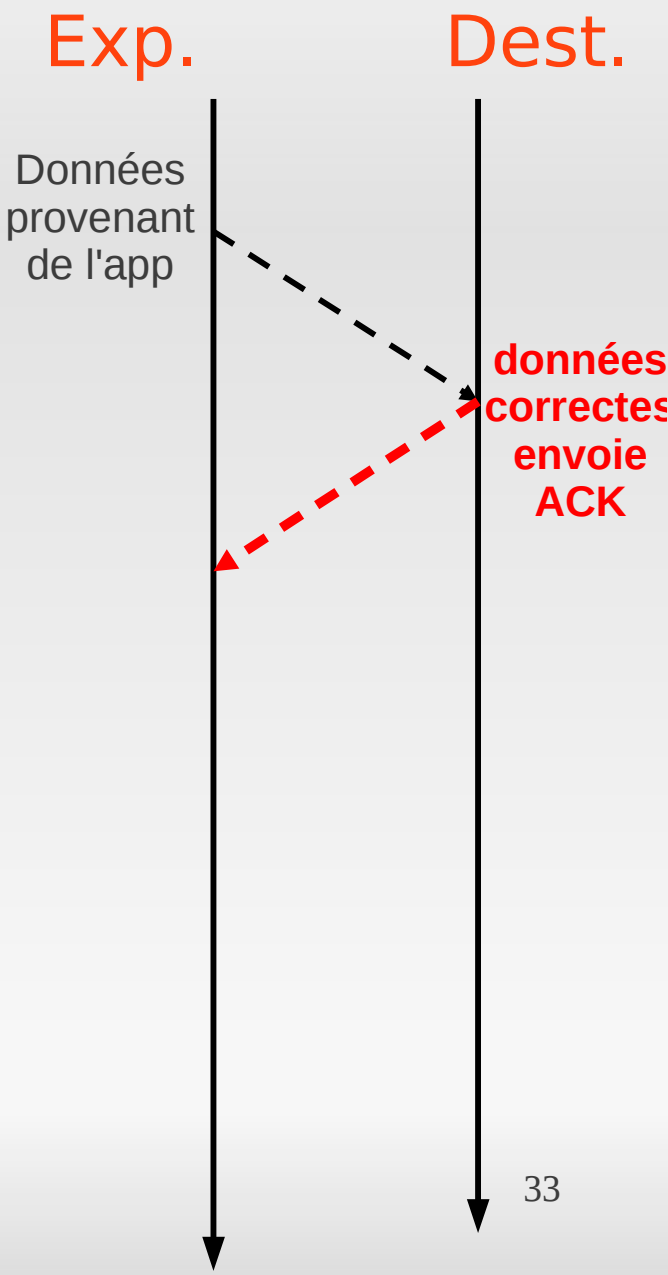




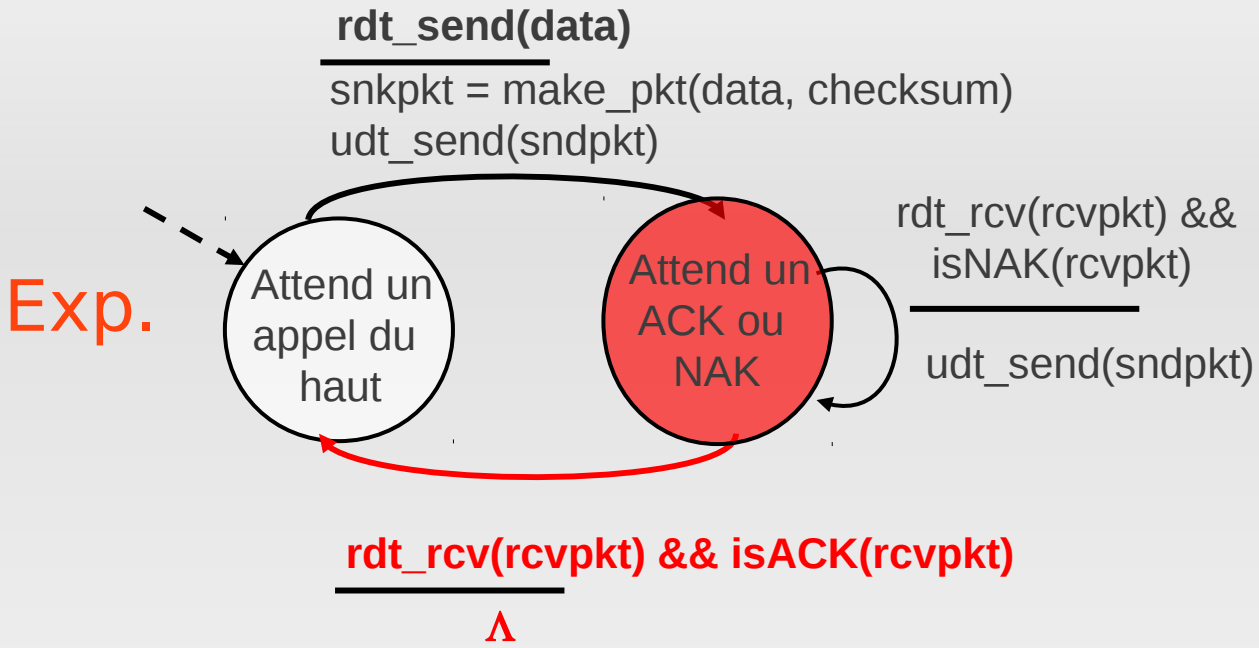
rdt2.0: exemple de déroulement (4/10)

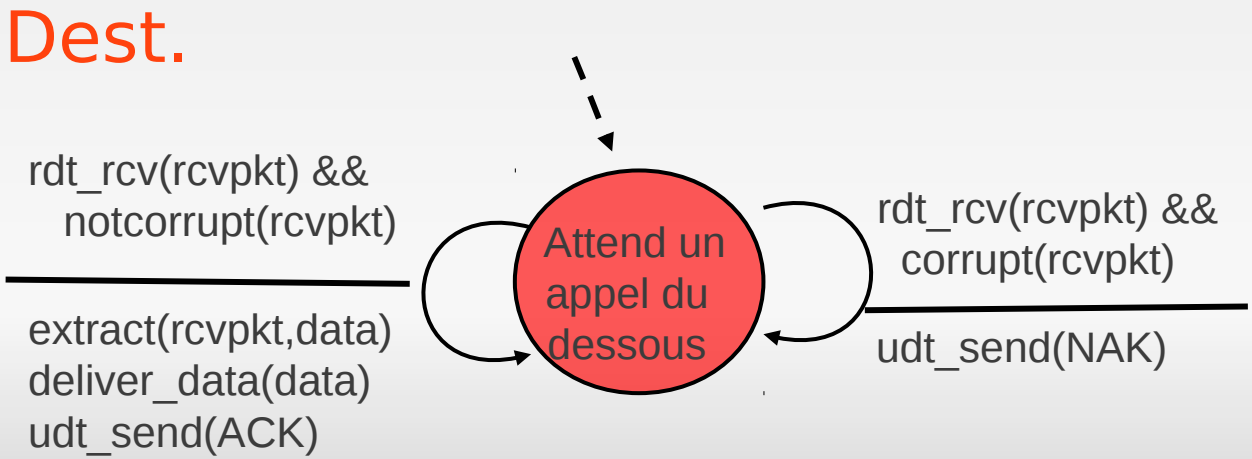


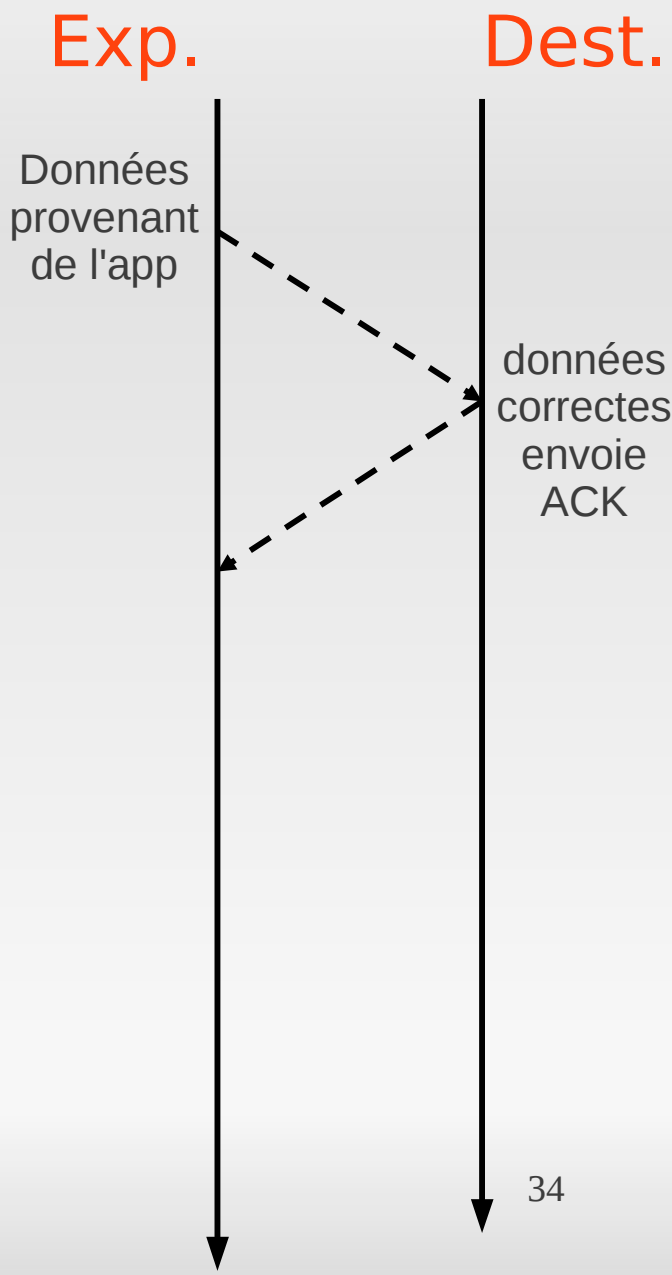




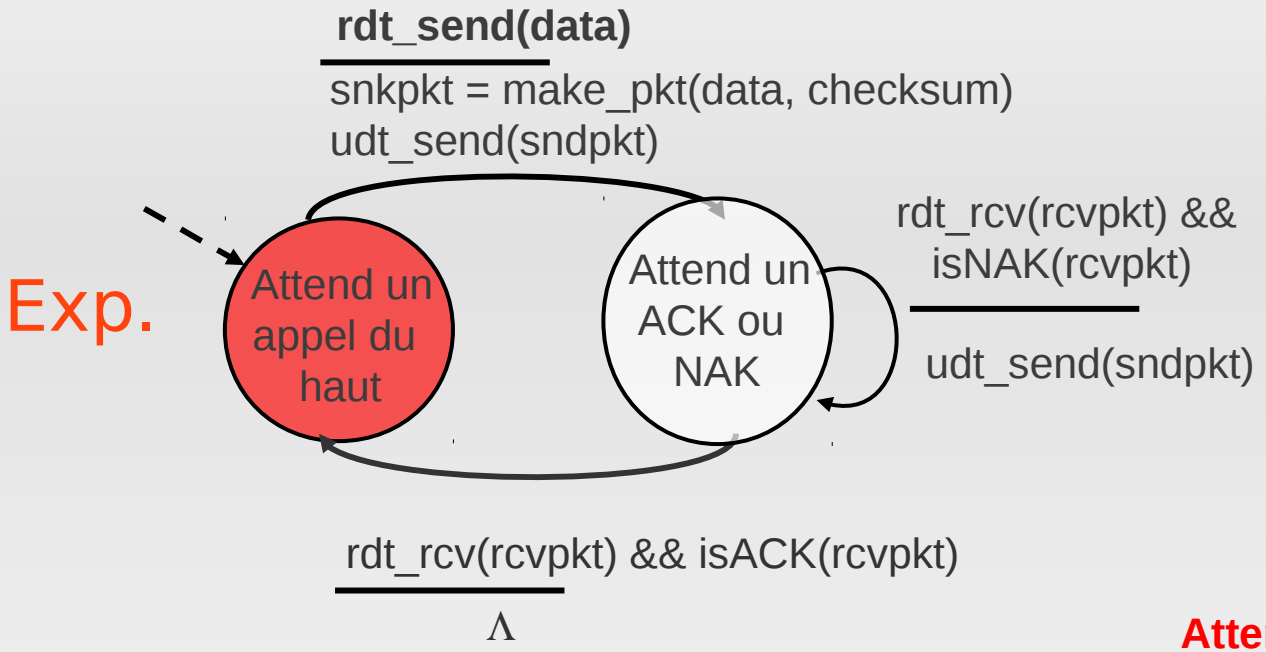
rdt2.0: exemple de déroulement (5/10)

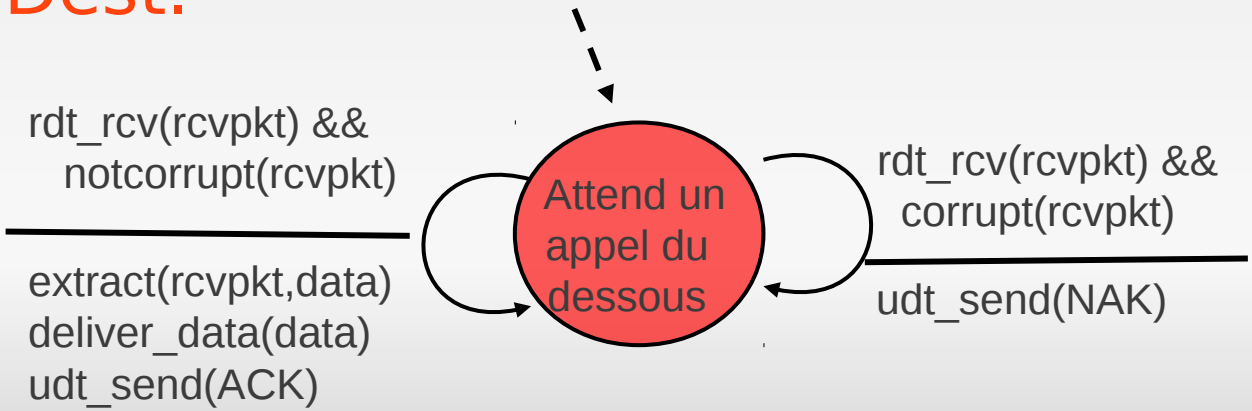


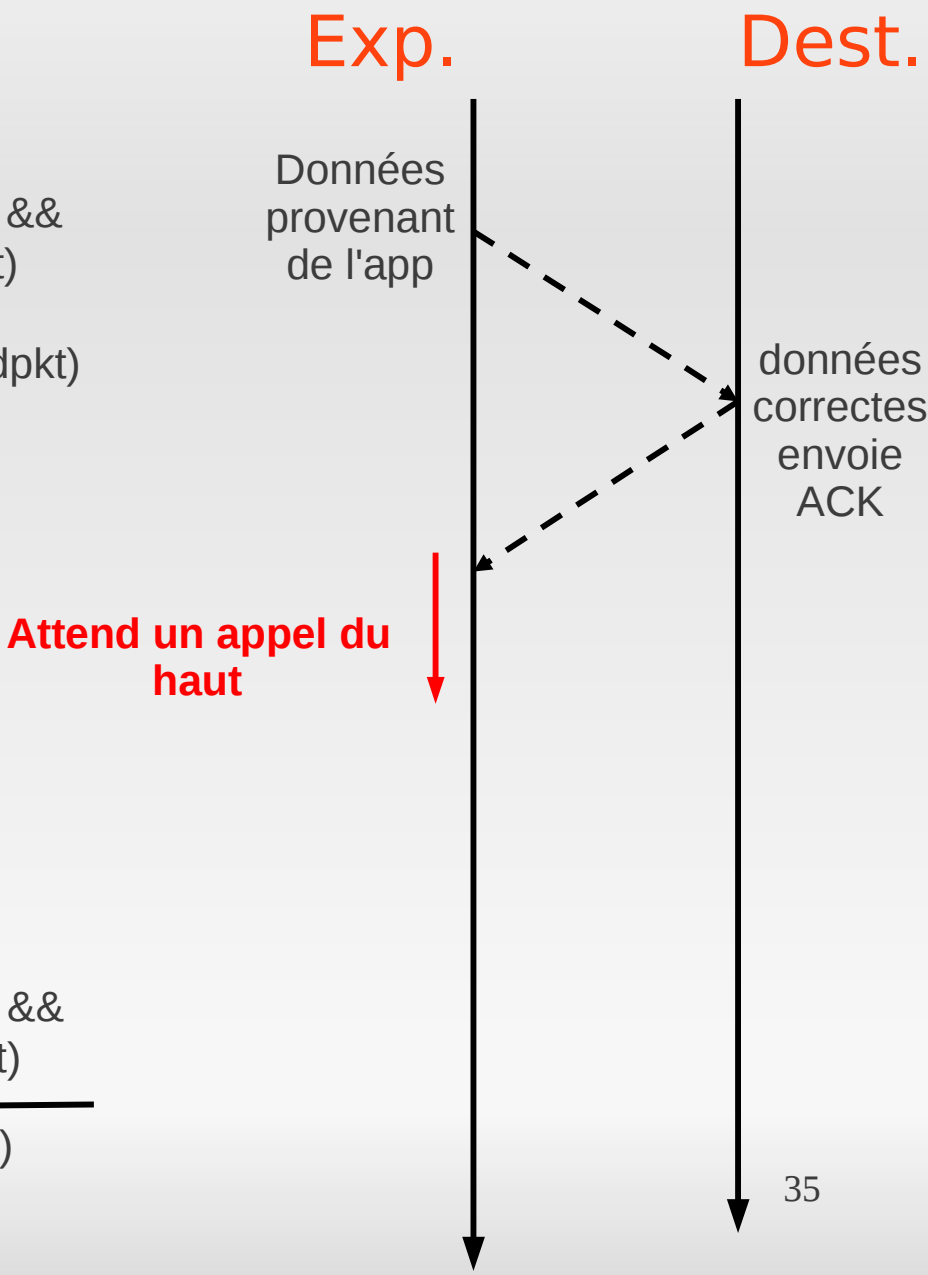




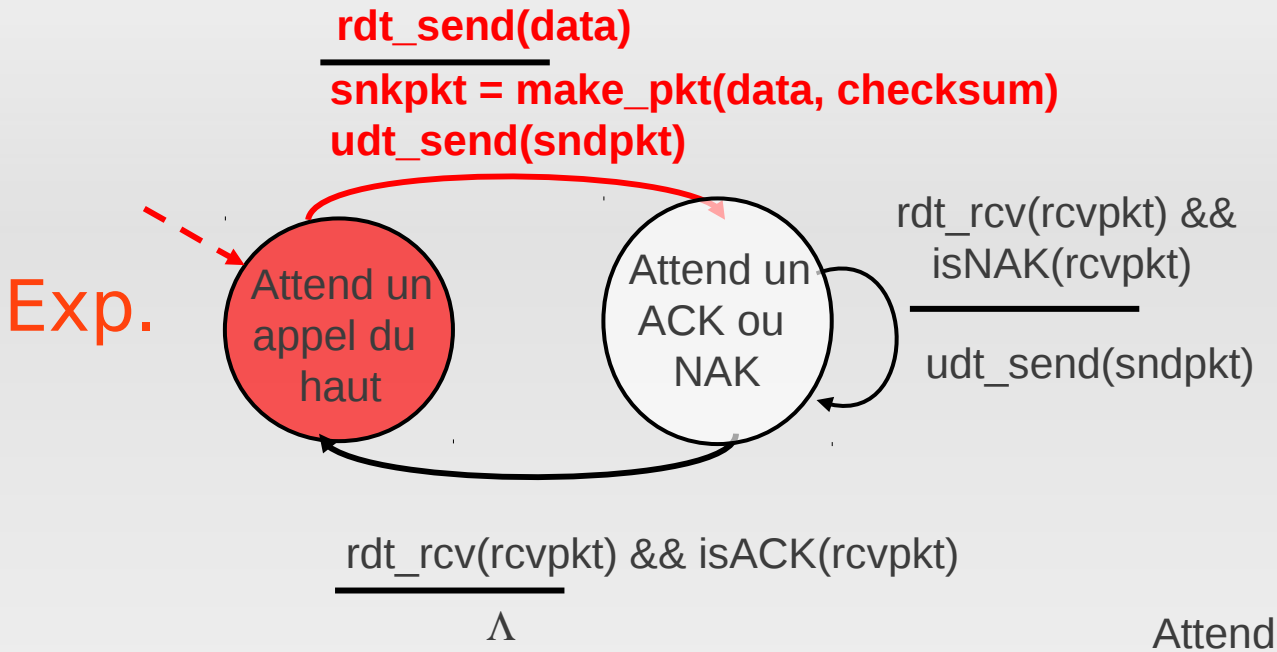
rdt2.0: exemple de déroulement (6/10)

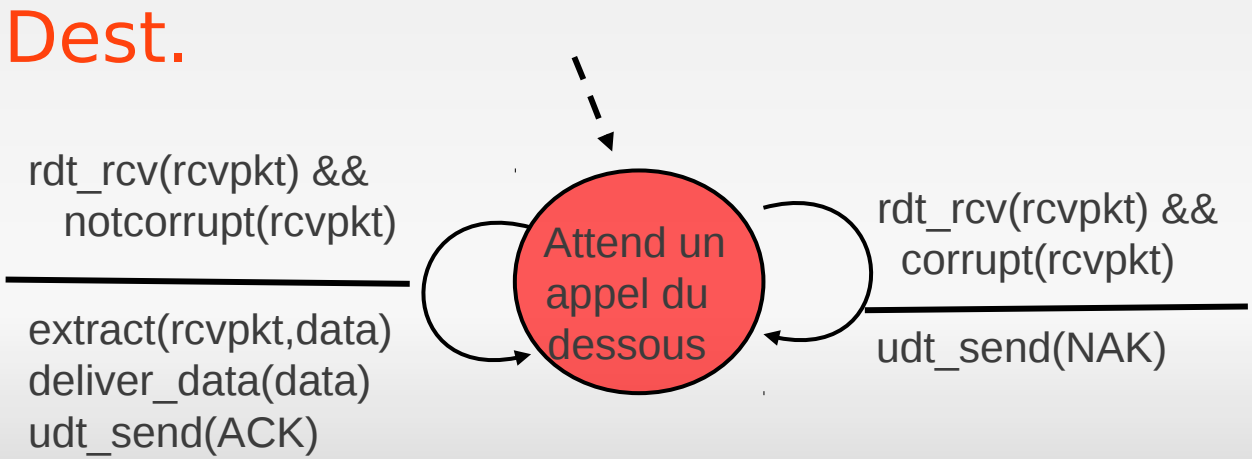


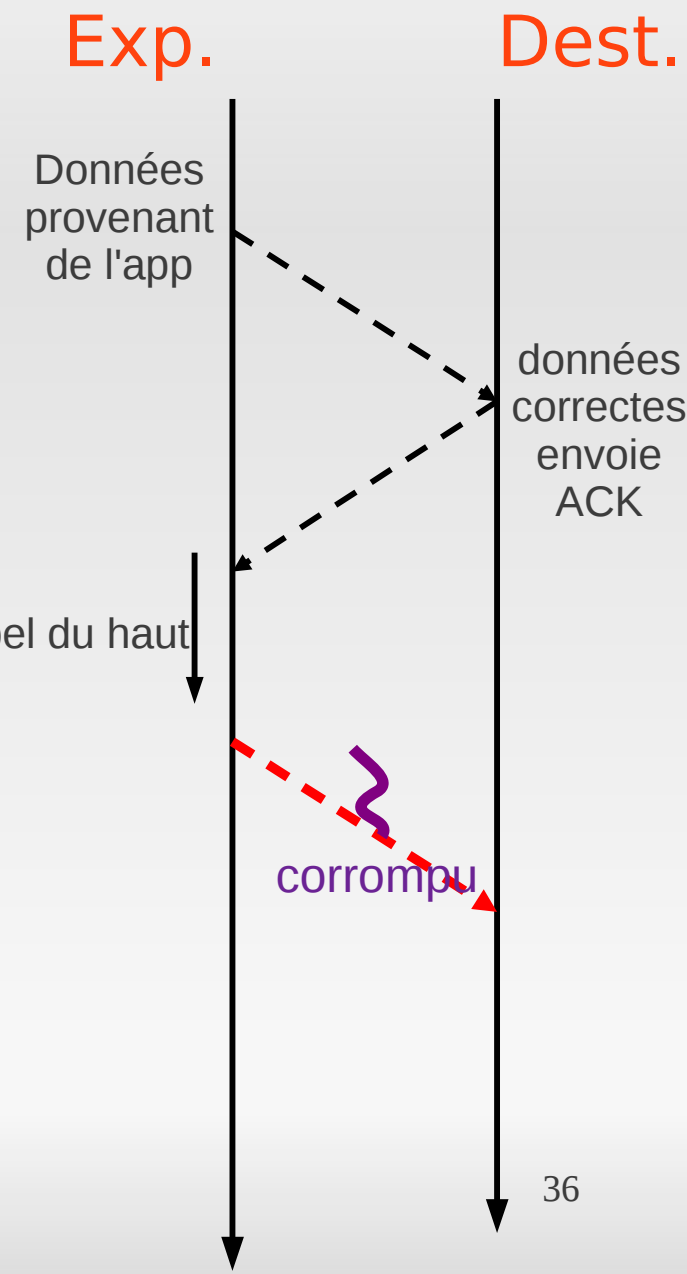




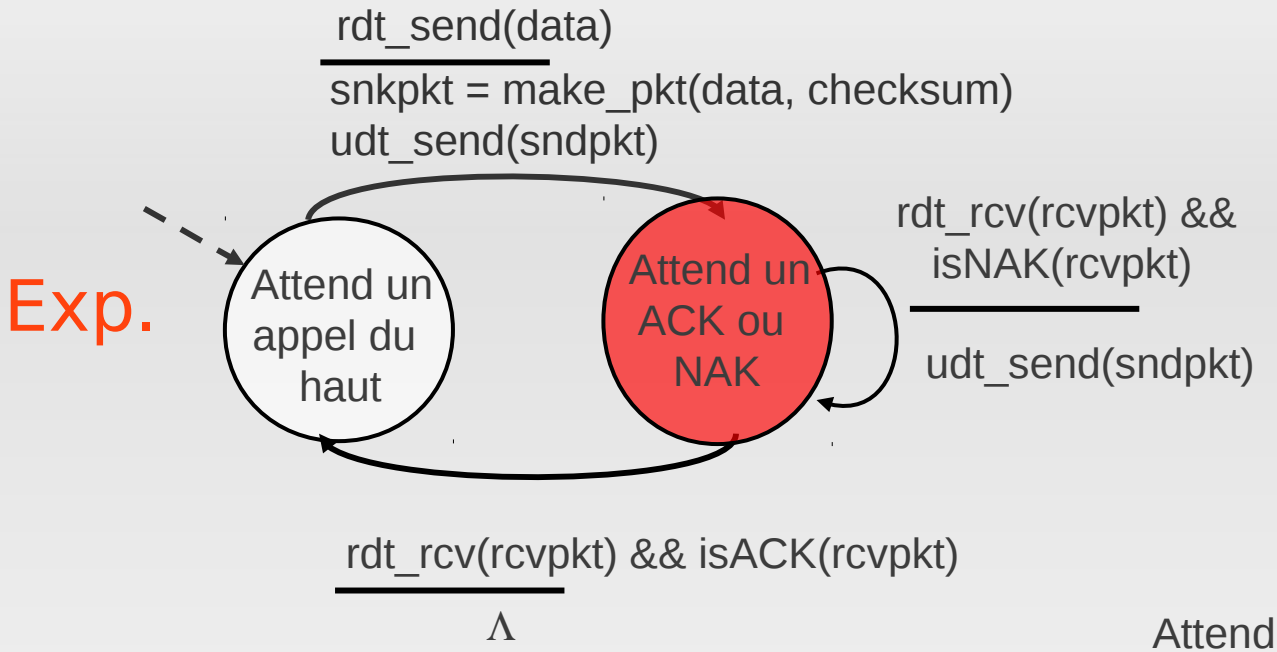
rdt2.0: exemple de déroulement (7/10)

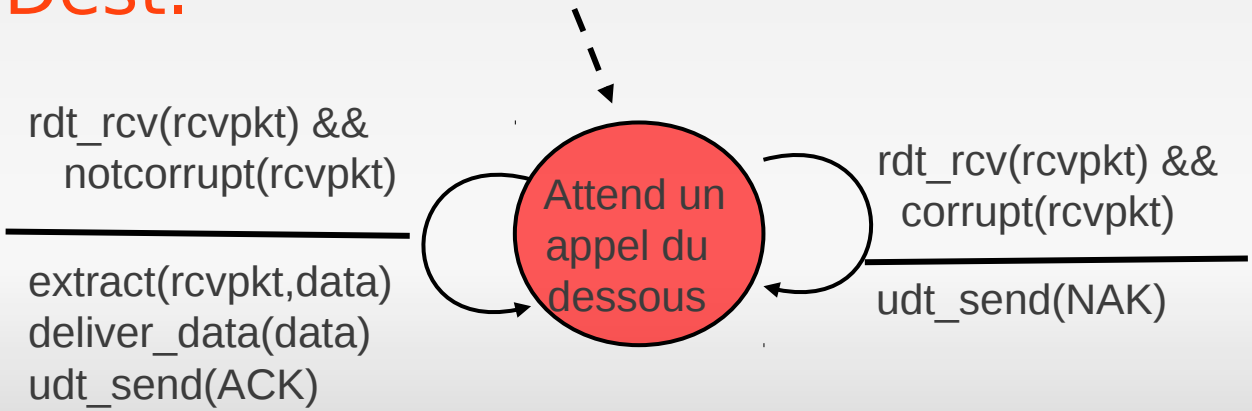


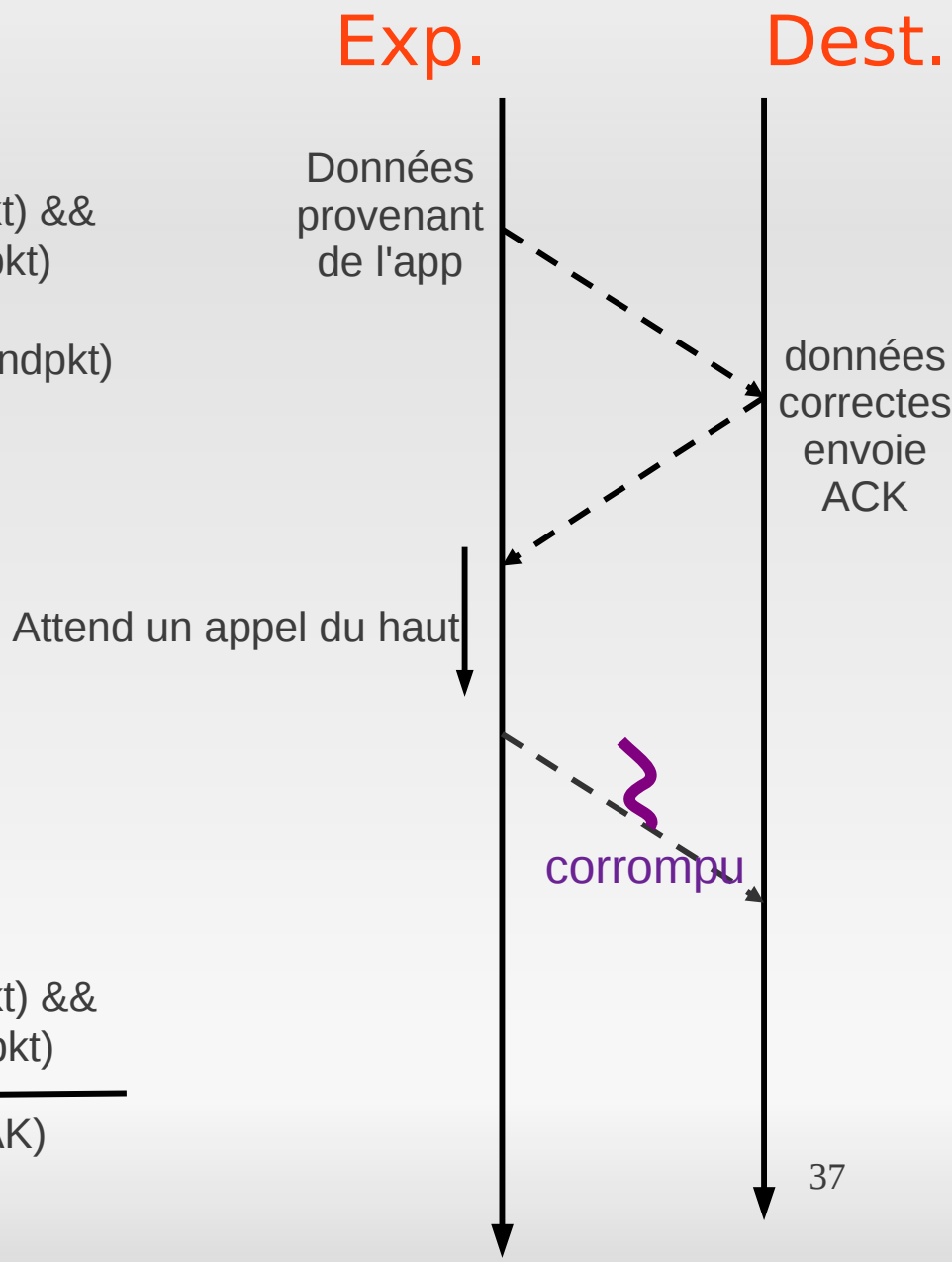




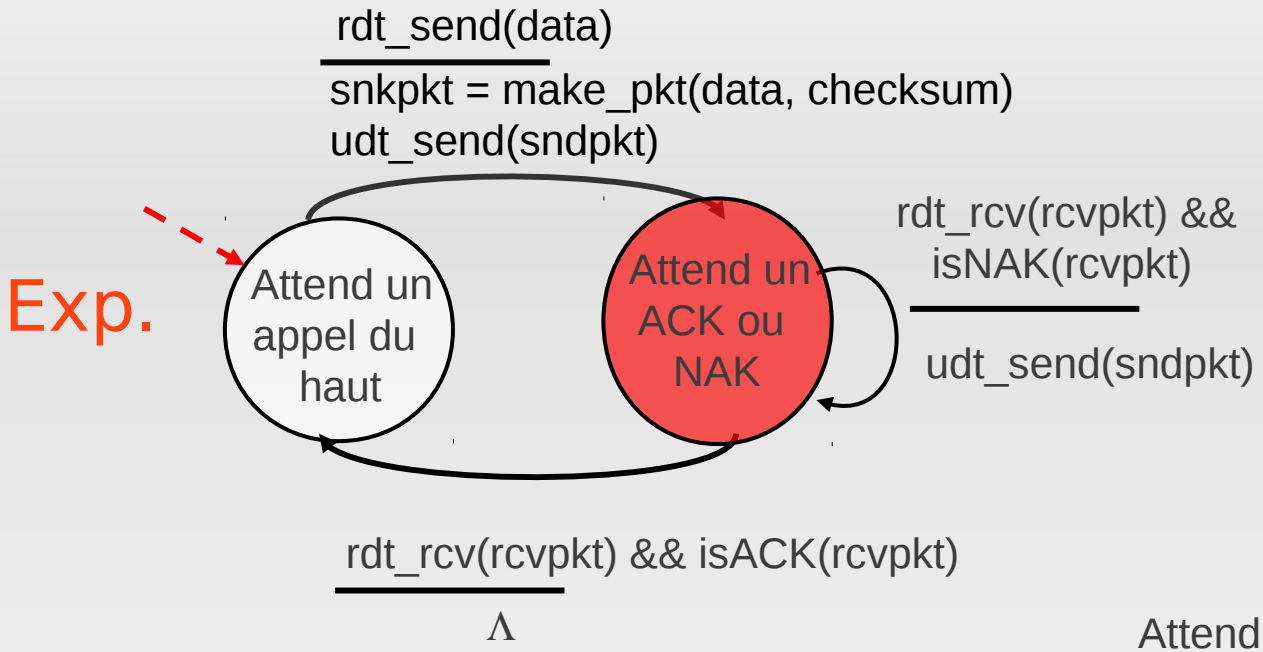
rdt2.0: exemple de déroulement (8/10)

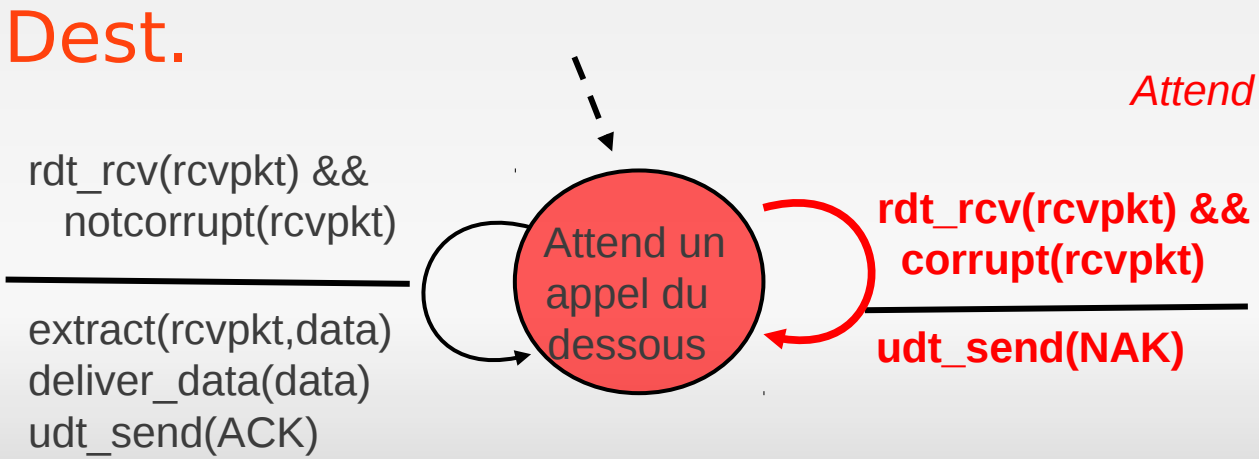


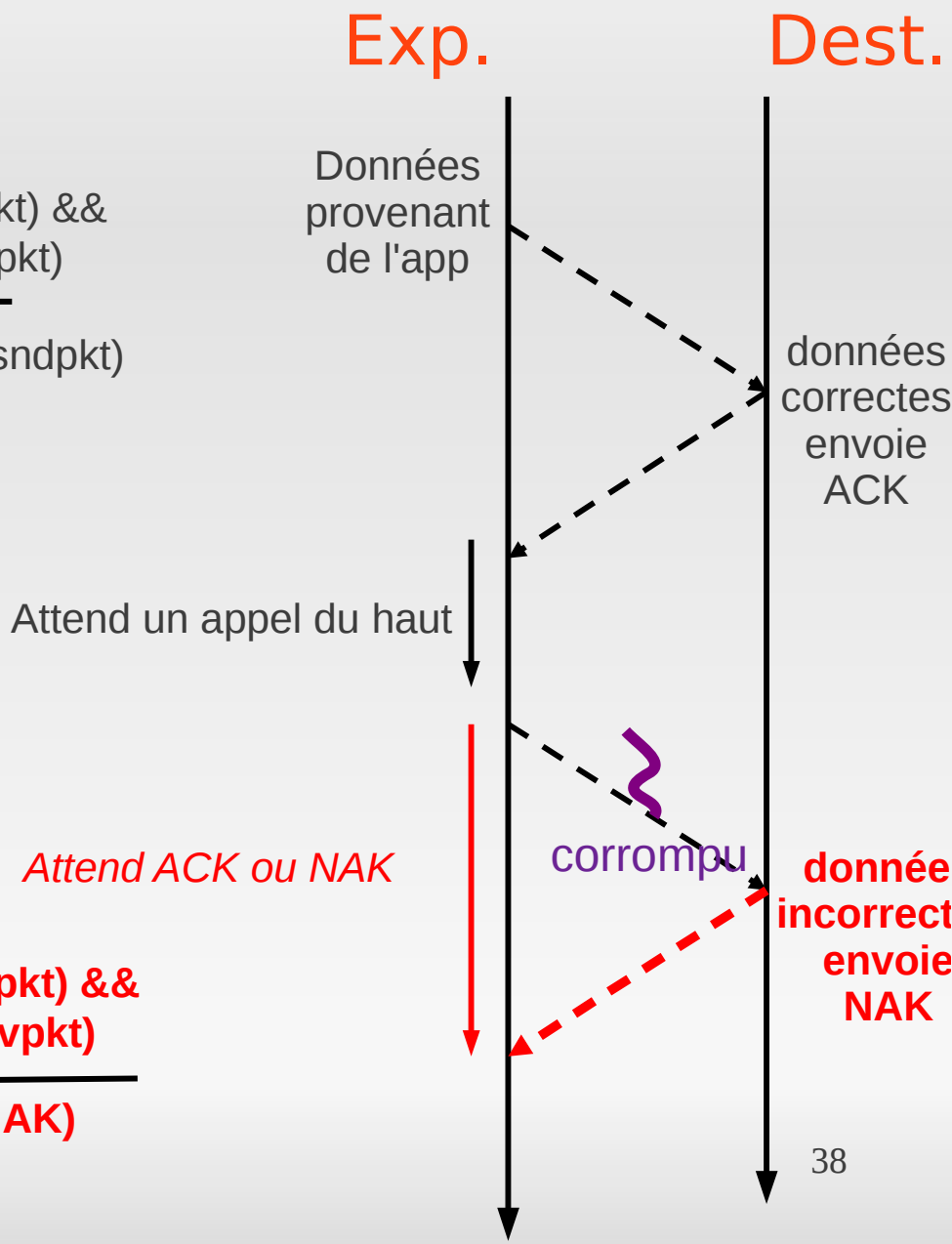




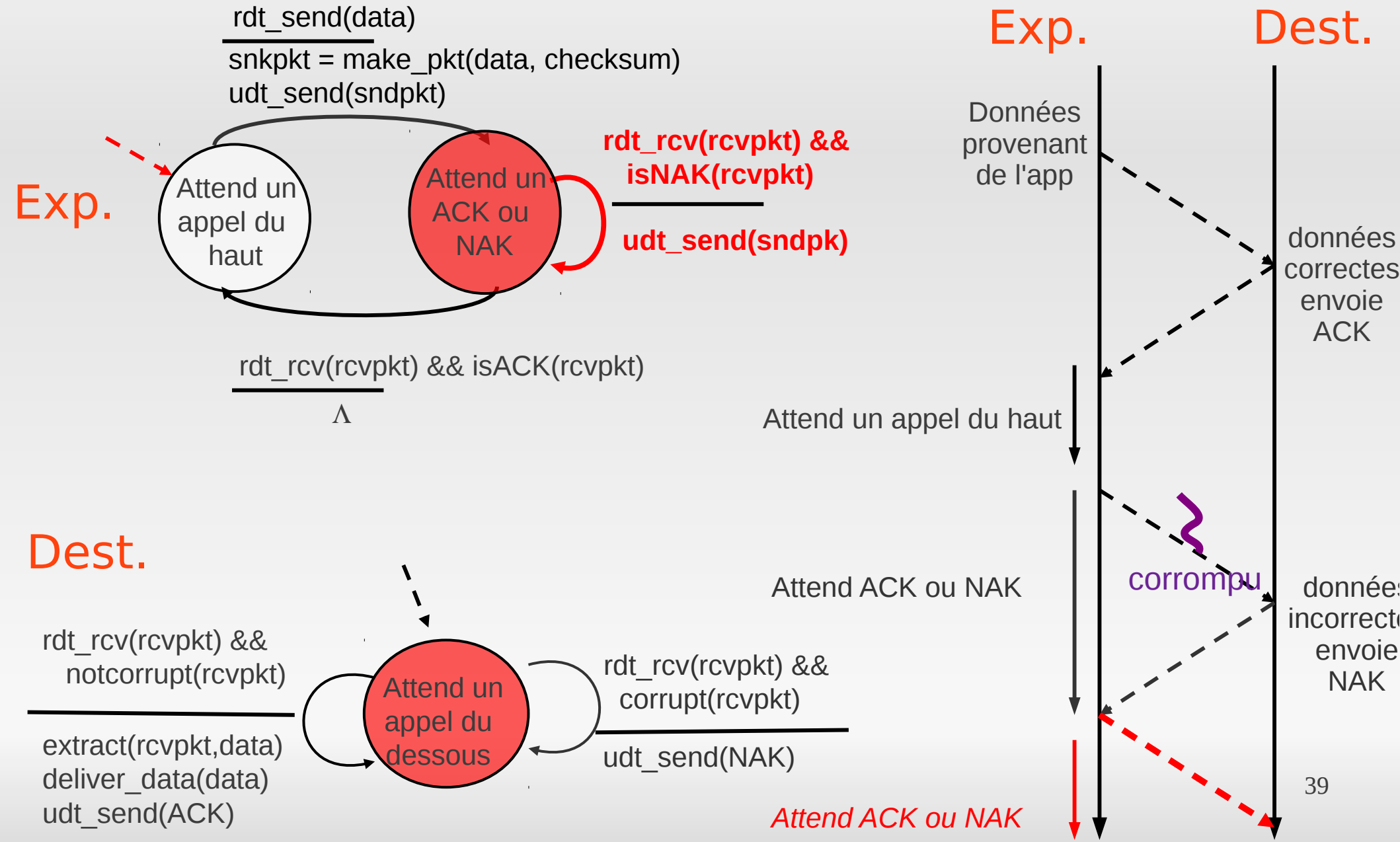
rdt2.0: exemple de déroulement (9/10)







rdt2.0: exemple de déroulement (10/10)



rdt2.0 a une faille fatale!

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

- L'expéditeur ne sait pas ce qu'il s'est passé au pôle destinataire!
- Ne peut pas retransmettre: car ce la produirait un possible duplicata

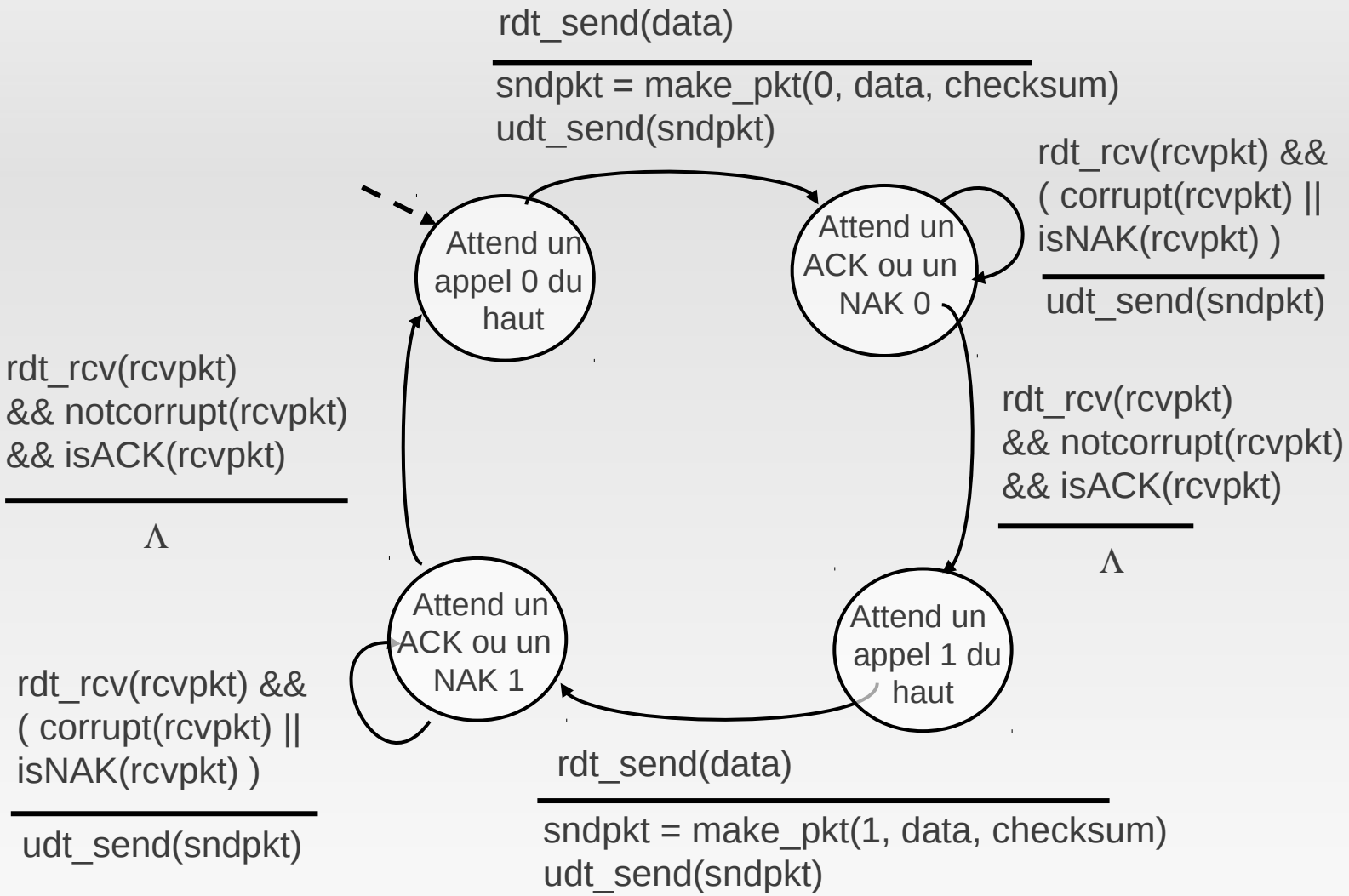
Gérer les duplicatas:

- L'expéditeur retransmet le paquet courant si ACK/NAK abîmer
- L'expéditeur ajoute un *numéro de séquence pour chaque paquet*
- Le destinataire rejette (ne délivre pas au dessus) des paquets dupliqués

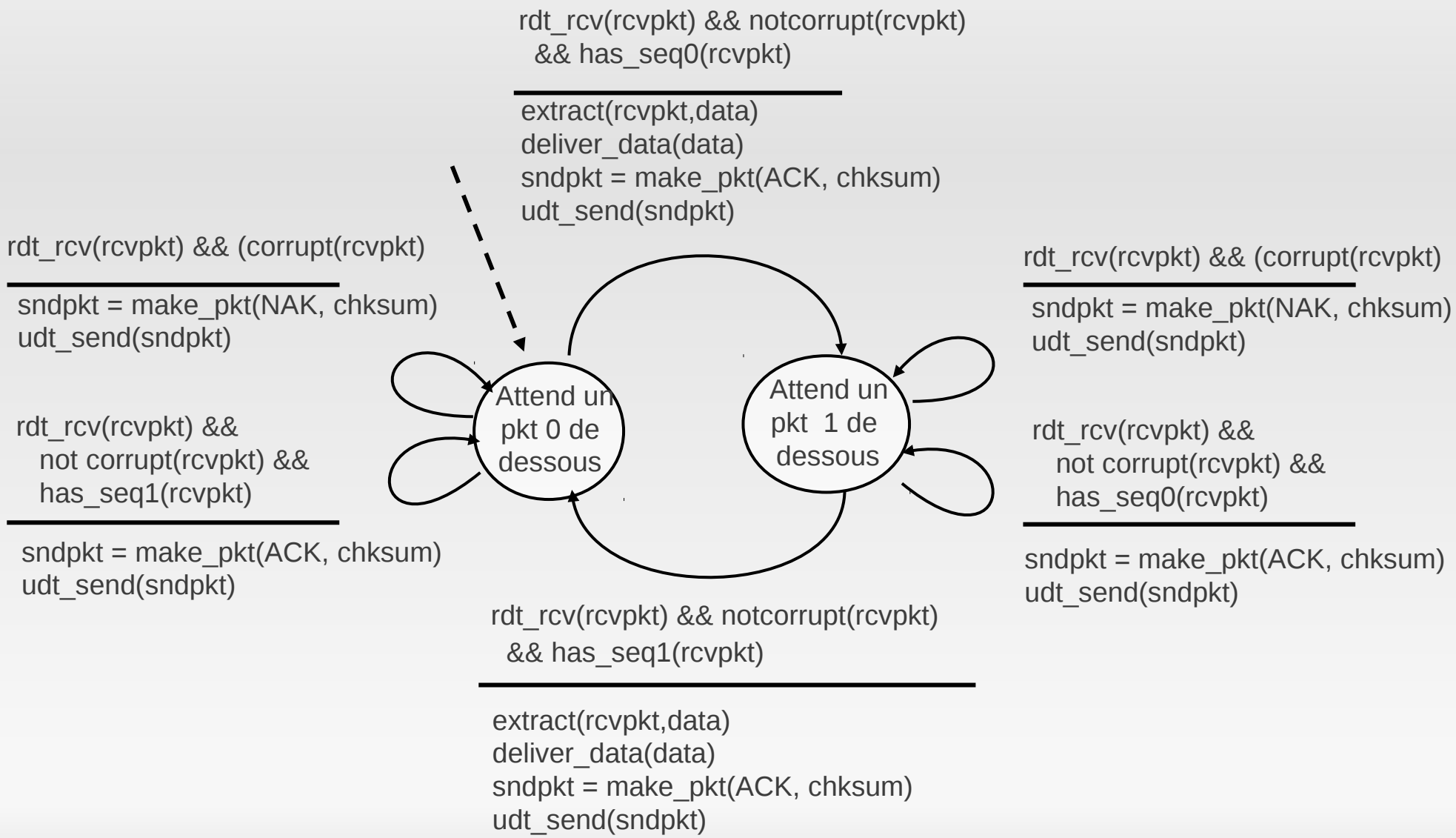
« stop and wait »

L'expéditeur envoie un paquet, puis attend une réponse du destinataire

rdt2.1: expéditeur, gestion des ACK/NAKs abîmés



rdt2.1: destinataire, gestion des ACK/NAKs corrompus



rdt2.1: discussion

Expéditeur:

- Seq. # ajouter au pkt
- Deux numéros de séquence (0,1) suffisent. Pourquoi?
- Doit vérifier si le ACK/NAK est corrompu
- Deux fois plus d'état
  - Etat doit “se souvenir” si le pkt courant a un seq # valant 0 ou 1

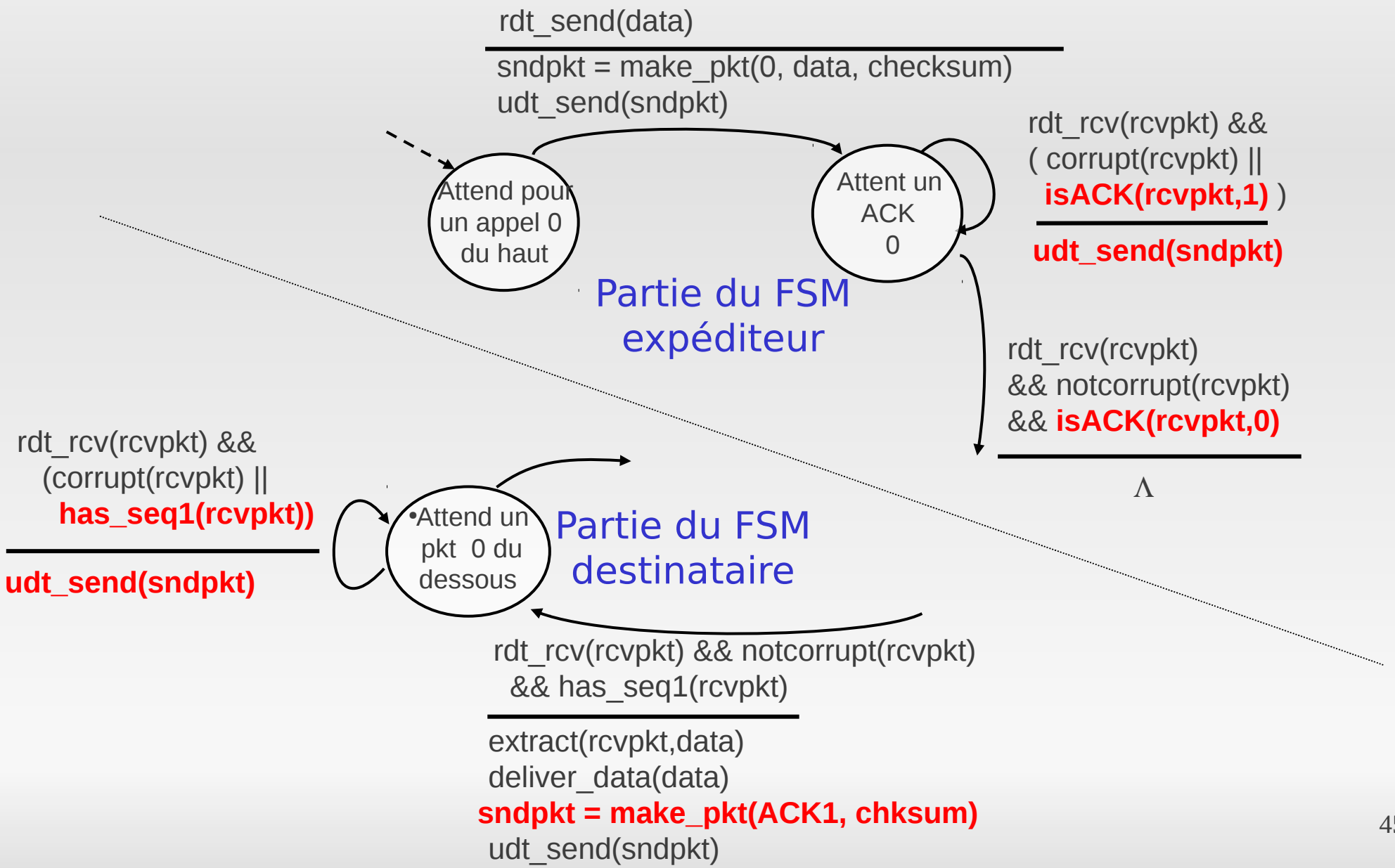
Destinataire:

- Doit vérifier si le paquet est dupliqué
  - Etat indique si un pkt 0 ou 1 est attendu
- note: destinataire ne peut *pas* savoir si sont dernière ACK/NAK a été reçu correctement par l'expéditeur

rdt2.2: un protocole sans NAK

- Même fonctionnalité que rdt2.1, mais n'utilise que des ACKs
- A la place d'un NAK, le destinataire envoie un ACK pour le dernier pkt reçu sans erreur
  - Le destinataire doit *explicitement* inclure un numéro de seq # du pkt qui a été ACKé
- Des ACK dupliqués arrivant à l'expéditeur provoque une même action qu'un NAK: *retransmission du paquet courant*

rdt2.2: Expéditeur, Destinataire (partiel)



rdt3.0: Cannaux avec erreur *et* pertes

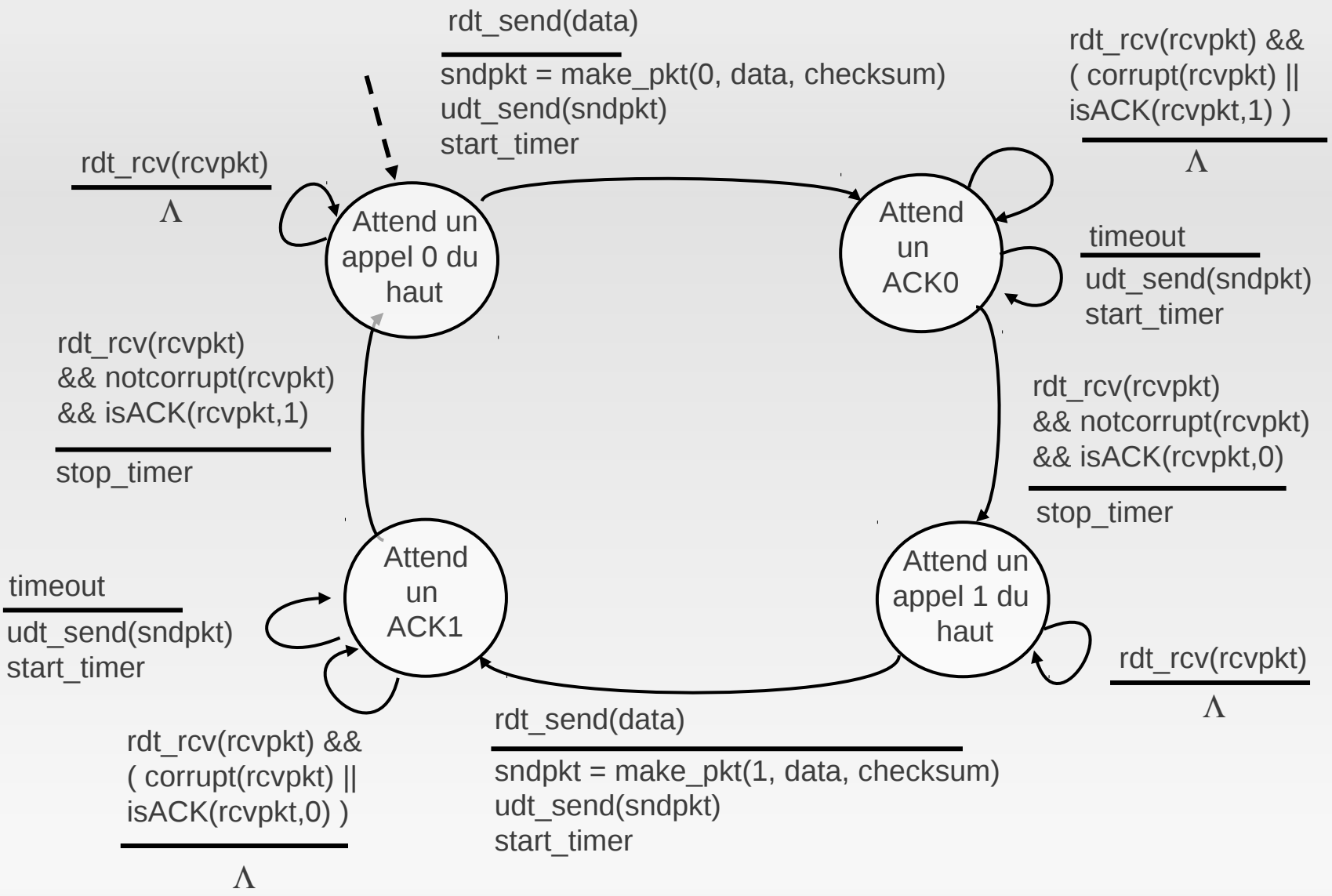
Nouvelle hypothèse : Le canal de transmission peut aussi perdre des paquets (données ou ACKs)

- checksum, seq. #, ACKs, retransmissions sera nécessaire, mais pas suffisante

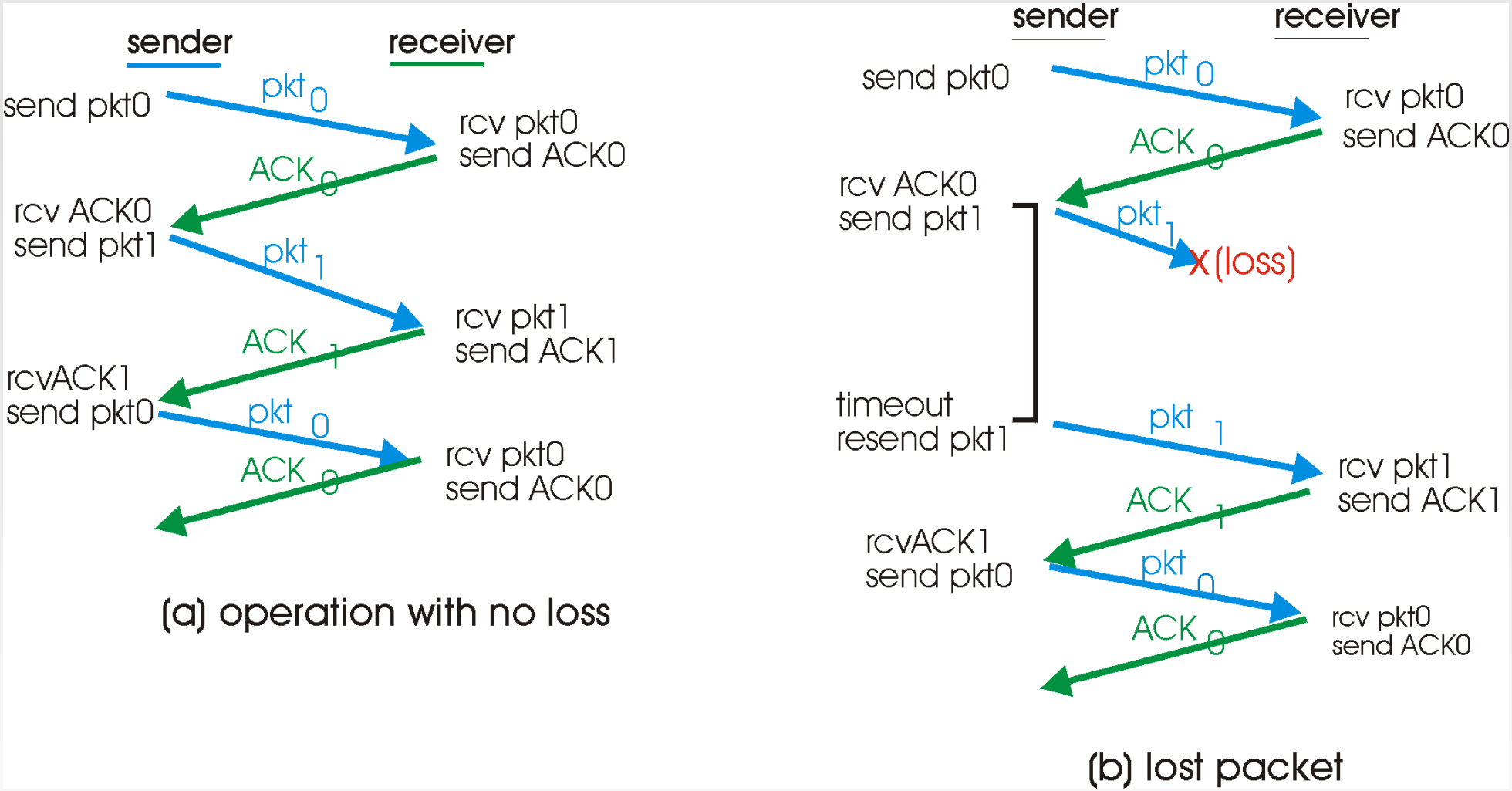
Approche adoptée: l'expéditeur attend durant un temps "*raisonnable*" l'ACK attendu

- Il le retransmet si aucun ACK n'est reçu durant cette période
- Si le pkt (ou ACK) est juste retardé (mais pas perdu):
  - La retransmission sera dupliquée, mais l'utilisation de numéro de sequence déjà permet de remédier à ca
  - Le destinataire doit spécifier le numéro de séquence d'un paquet ayant été ACKé
- Nécessite un chronomètre

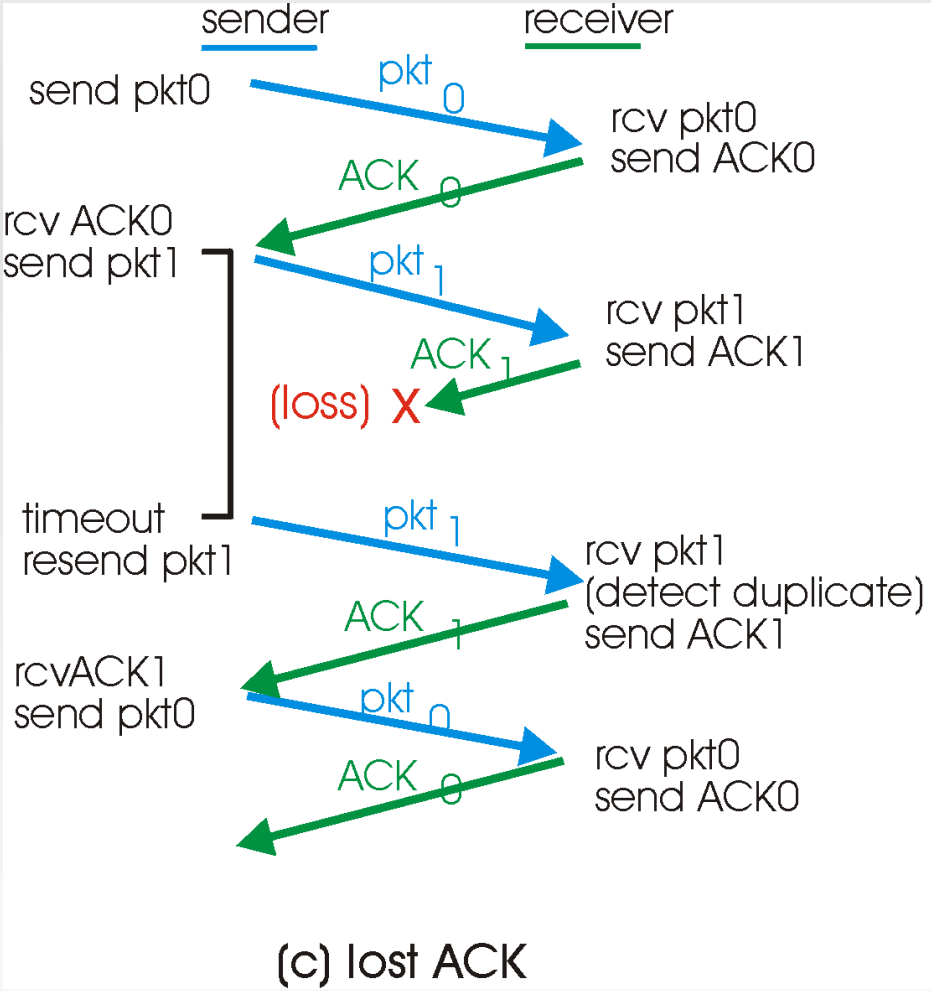
rdt3.0 expéditeur



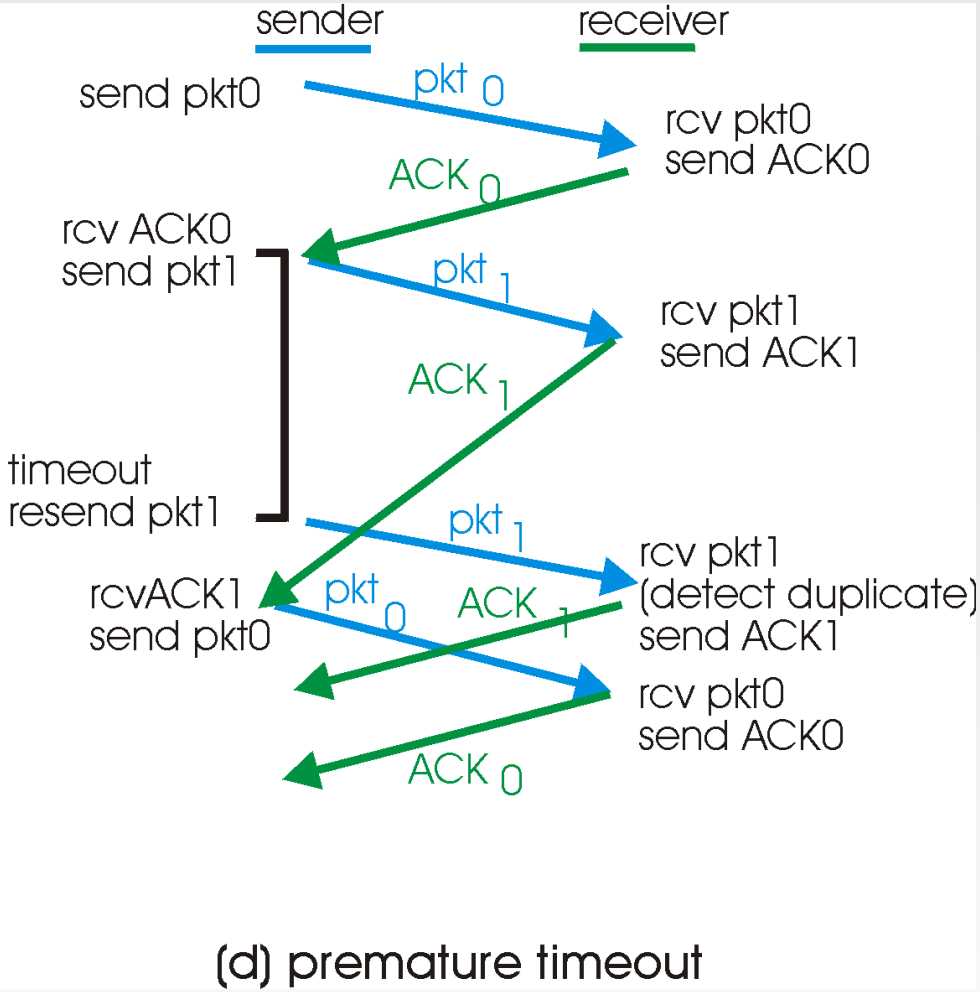
rdt3.0 en action



rdt3.0 en action



(c) lost ACK



(d) premature timeout

Performance de rdt3.0

- rdt3.0 fonctionne, mais sa performance est minable
- exemple: une liaison a 1 Gbps , 15 ms de délai de propagation, paquet de 1KB:

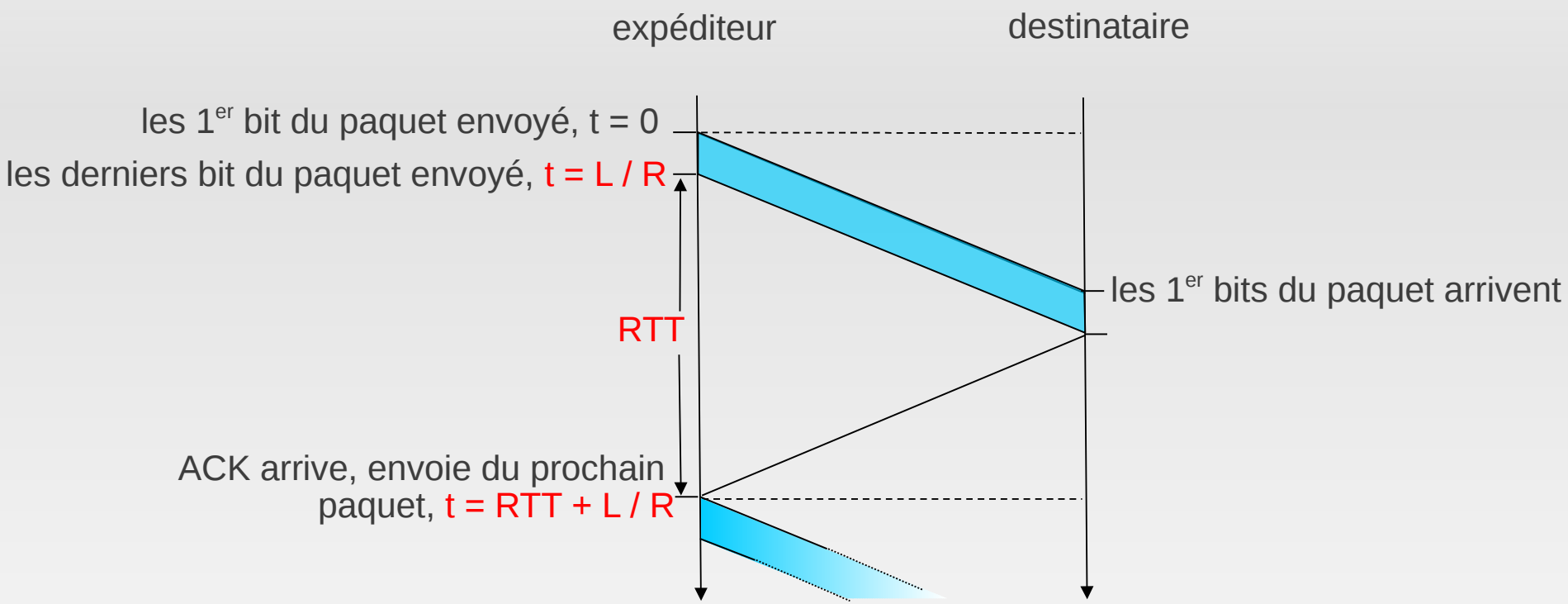
$$T_{transm} = \frac{L \text{ (long en bits du pkt)}}{R \text{ (taux de transmission, bps)}} = \frac{8kb/pkt}{10^{**}9 \text{ b/sec}} = 8 \text{ microsec}$$

- U_{exp}: utilisation – fraction du temps ou l'expéditeur est occupé à envoyer

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

- 1KB pkt chaque 30 msec -> 33kB/sec de débit sur un lien a 1 Gbps
- Protocole réseau réduit l'utilisation des ressources physiques!

Fonctionnement de rdt3.0

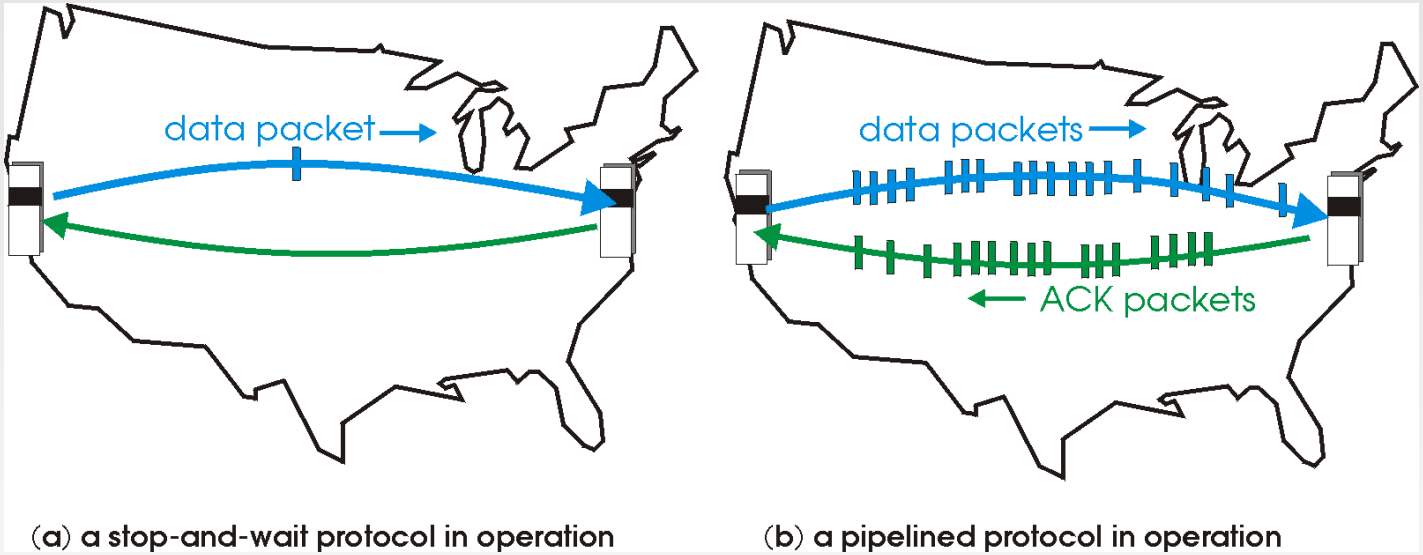


$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

Protocole pipeliné

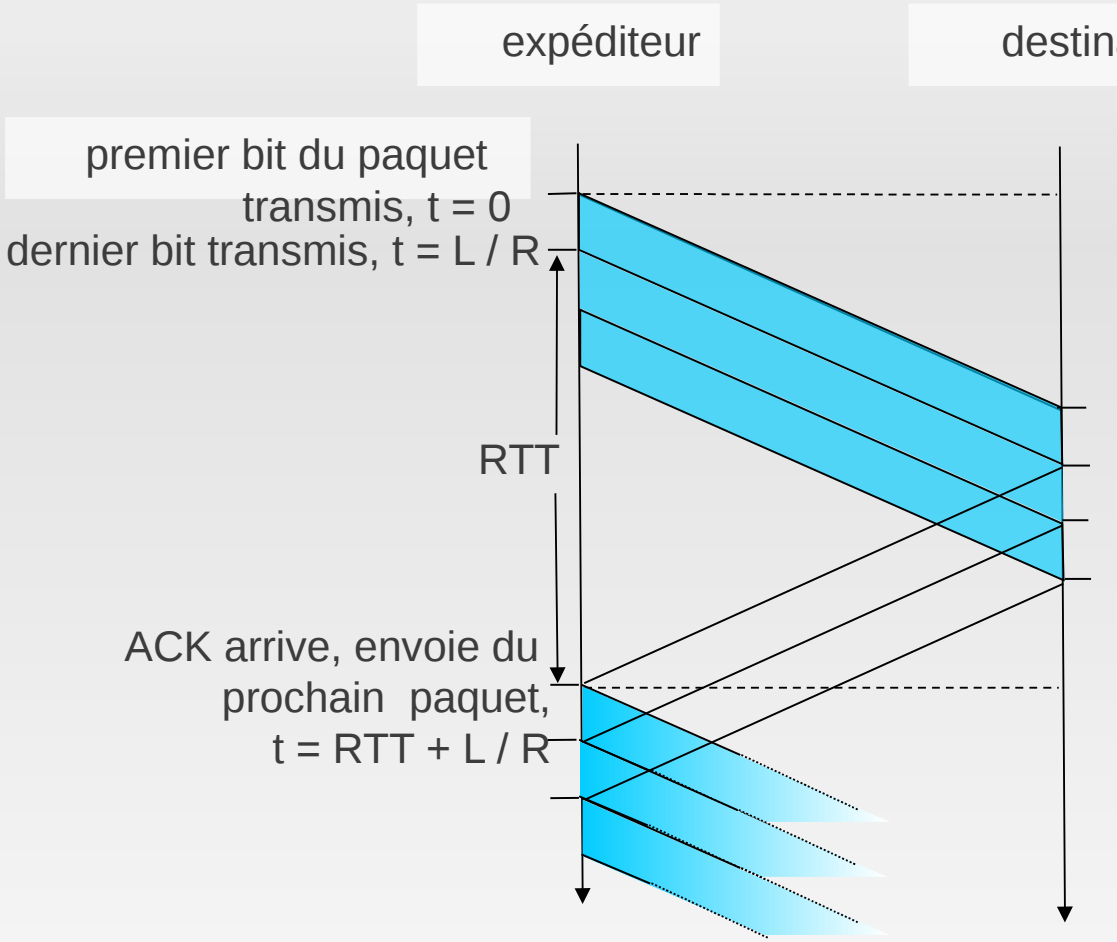
Pipeline: l'expéditeur permet l'envoie multiple de paquets en instance d'accusé de réception :

- L'espace des numéro de séquence doit être augmenté
- La mise en tampon au niveau expéditeur et du receveur



- Deux formes génériques des protocoles pipelinés: *go-Back-N, répétition sélective*

Pipeline: utilisation amélioré



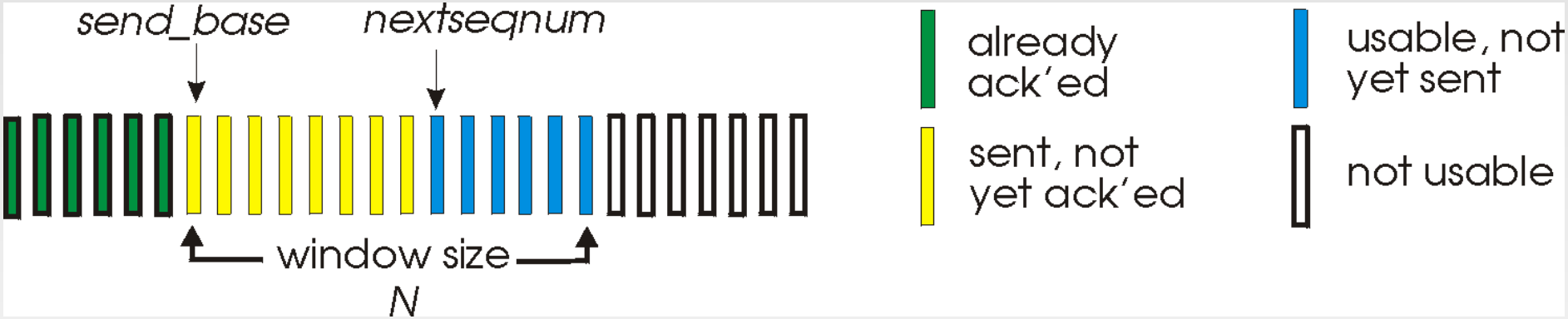
Améliore l'utilisation par un facteur 3!

$$U_{sender} = \frac{3 * L / R}{RTT + L / R} = \frac{.024}{30.008} = 0.0008$$

Go-Back-N

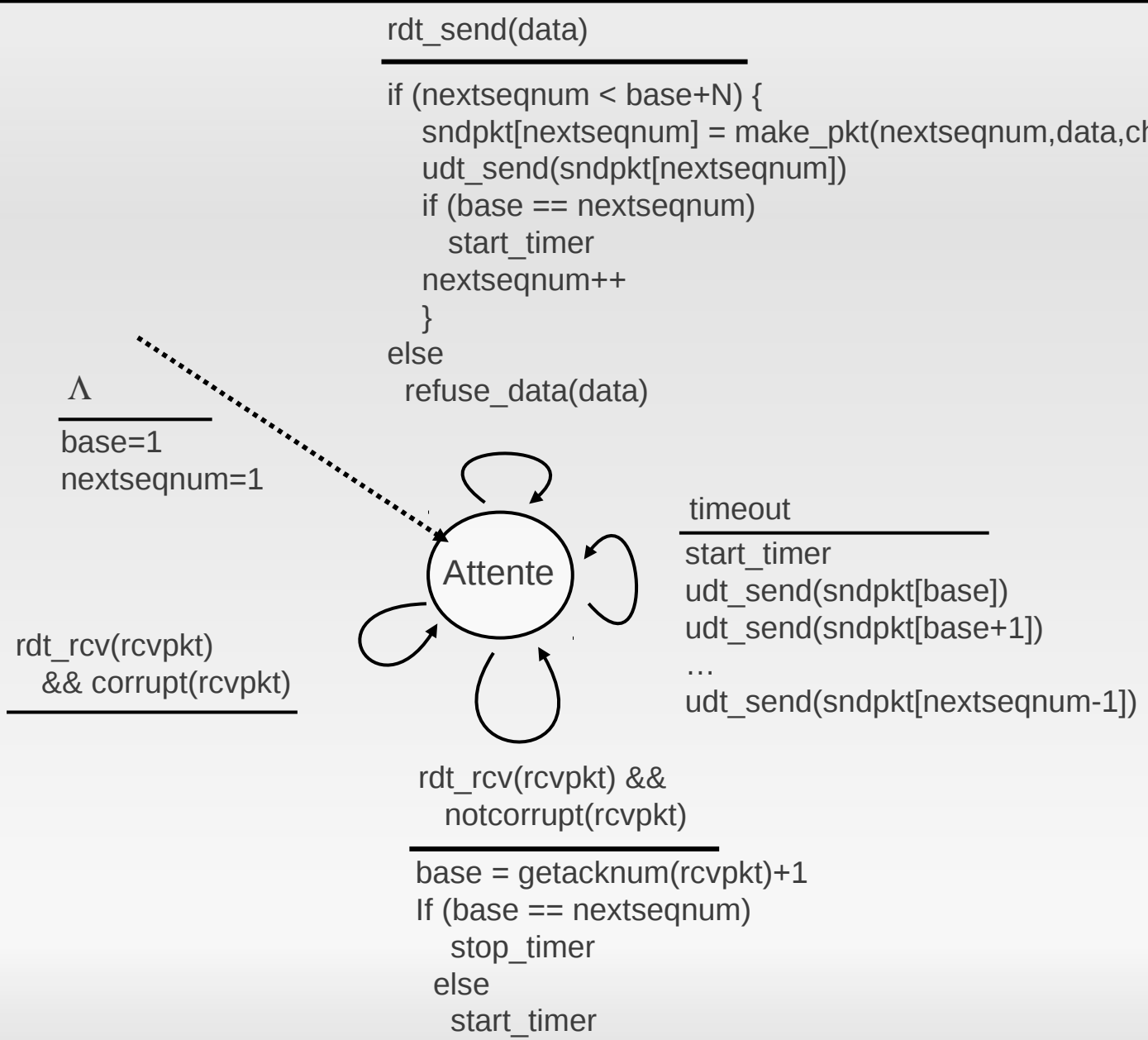
Sender:

- numéro de séquence sur K-bit dans l'entête du paquet
- "*fenêtre*" jusqu'a N, paquets non acquités simultanés

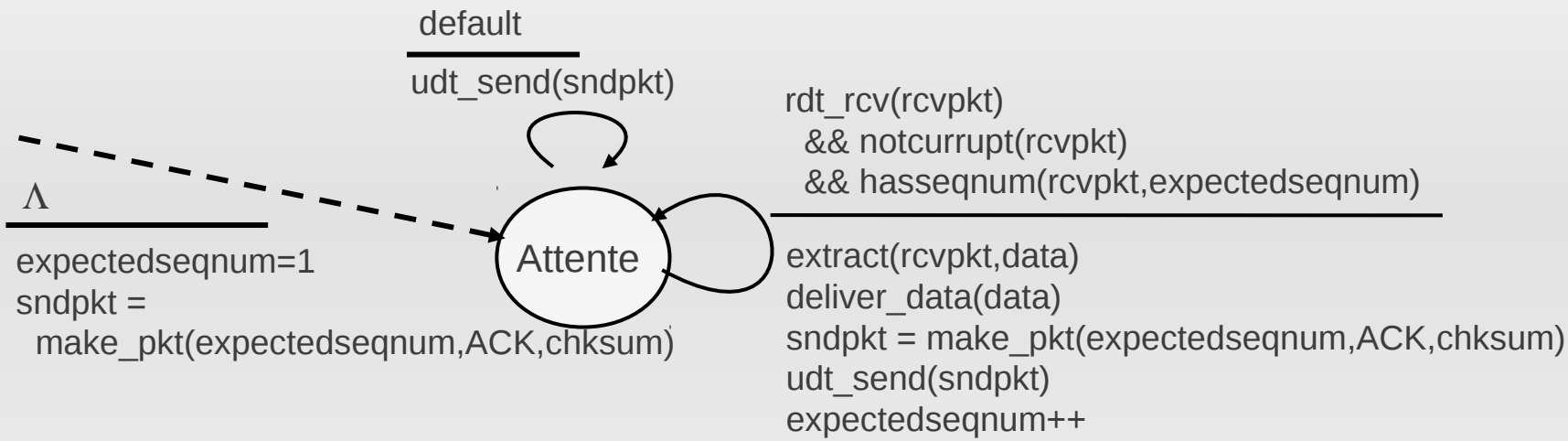


- ACK(n): accuse reception de tous les paquets jusqu'au numéro n (On parle alors d'"ACK cumulatif")
  - Peut recevoir des ACKs dupliqués (cf. destinataire)
- Minuterie (ou timer) pour chaque paquet envoyé
- *timeout(n):* retransmet le paquet *n* et ceux de plus grand numéro de séquence # pkts dans la fenêtre

GBN: FSM étendu de l'expéditeur



GBN: FSM étendu du destinataire

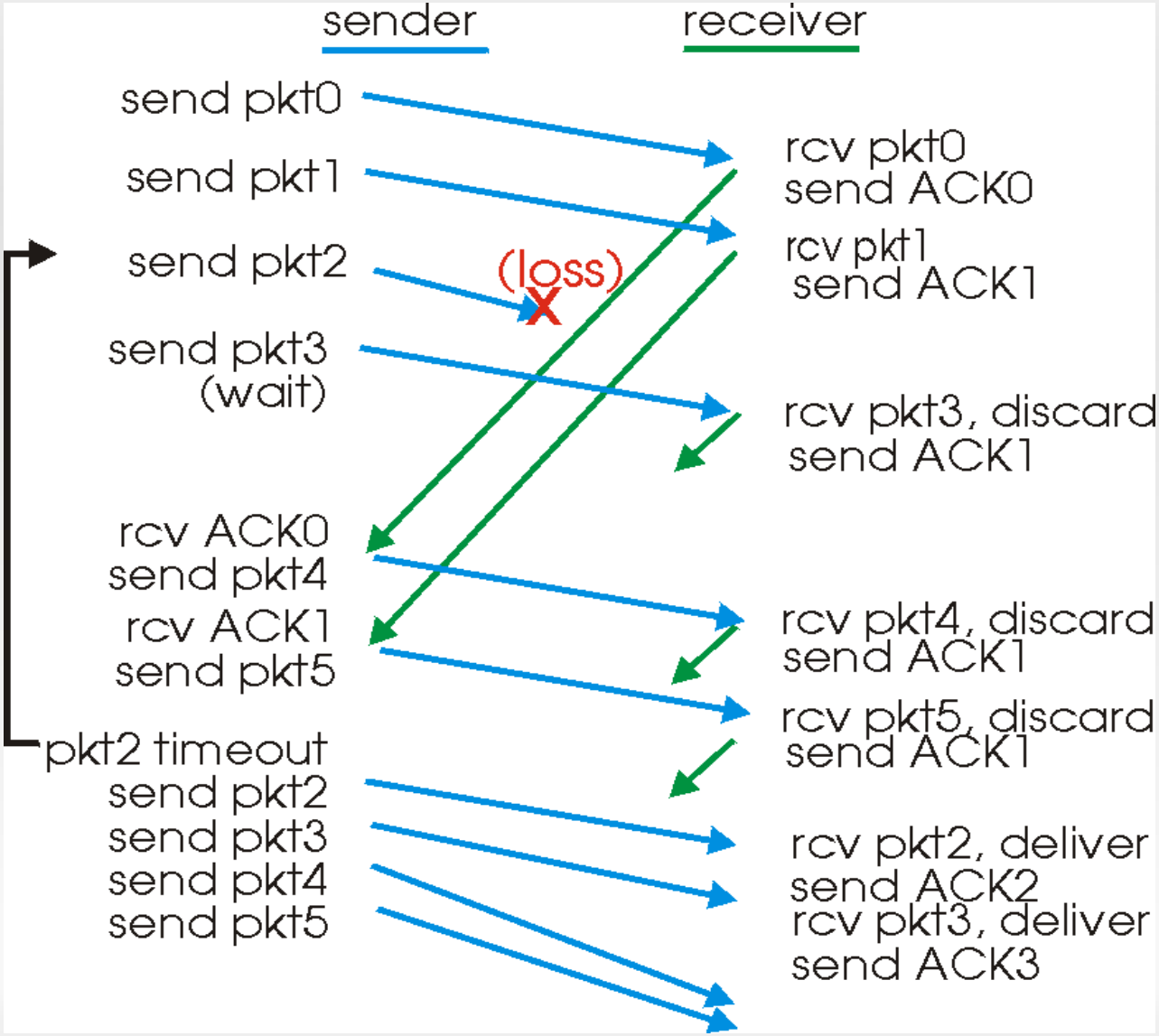


ACK-seulement: envoie toujours ACK pour des paquets correctement reçu avec le plus haut # seq *dans-ordre*

- peut génère des ACKs dupliqués
- Ont besoin seulement de mémorise **`expectedseqnum`**

- Paquet hors-séquence:
  - rejeter (ne pas mettre en tampon) -> pas de tampon utilisateur!
  - Re-ACK paquet avec le plus haut seq # en séquence

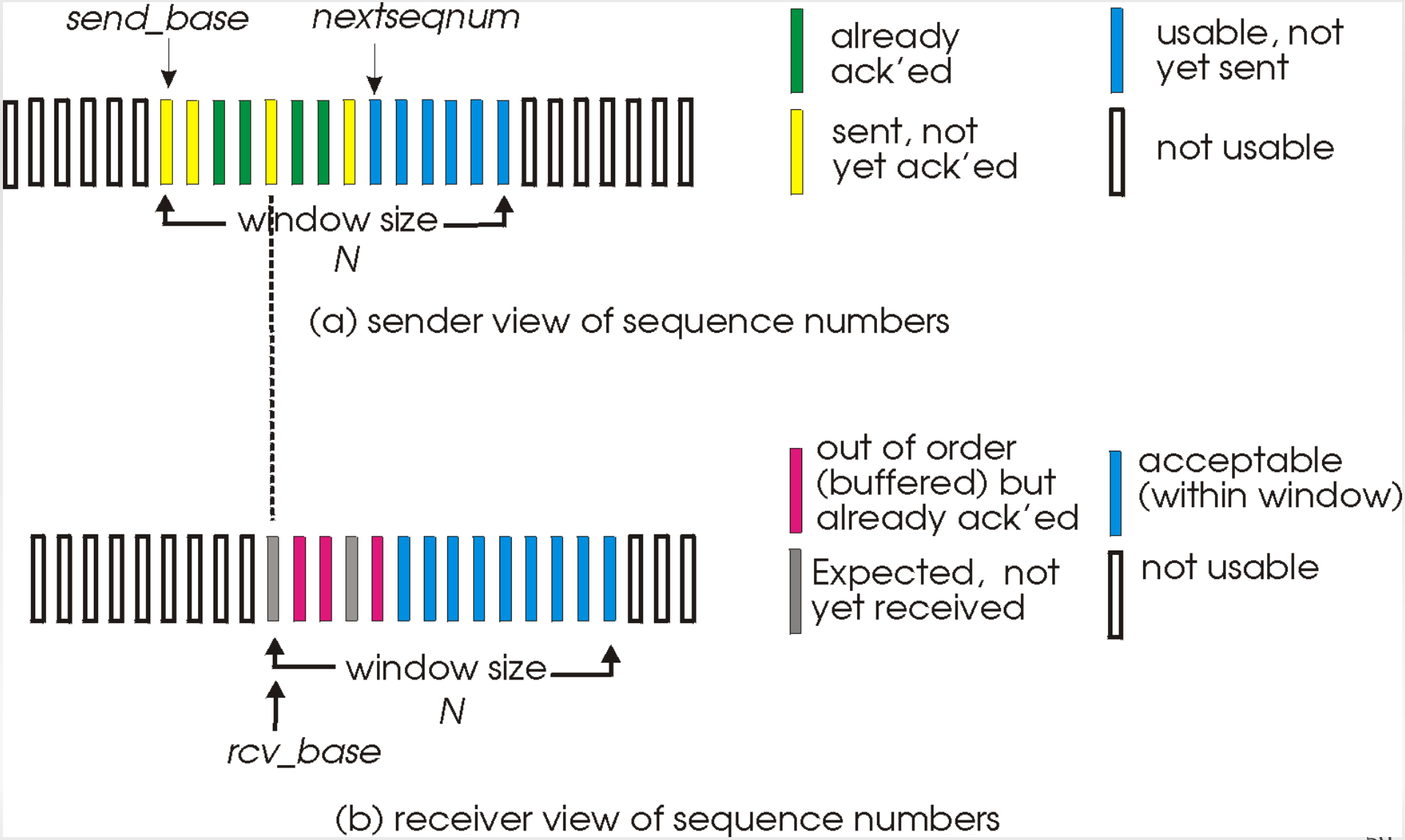
GBN en action



Répétition Sélective

- Le destinataire *accuse réception* individuellement tous les paquets recus correctement
  - Paquets dans le tampon, au besoin, au final pourra être délivré en-séquence à la couche supérieure
- L'expéditeur ré-envoie seulement les paquets avec un ACK non reçu
  - expéditeur lance un *timer* pour chaque paquet non acquité
- Fenêtre de l'expéditeur
  - N numéro de seq. consécutifs
  - Limite encore les numéro de seq. des paquets envoyés mais non acquités

Répétition sélective: fenêtre de l'expéditeur et du destinataire



Répetition sélective

expéditeur

data from above :

- Si il y a un numéro de seq. disponible dans la fenêtre send pkt

timeout(n):

- renvoyer pkt n, relancer le minuteur

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

- marquer pkt n comme reçu
- si n est le paquet de plus petit numéro, avancer la base de la fenêtre jusqu'au prochain numéro de paquet non-acquité

destinataire

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

- envoyer ACK(n)
- Hors séquence: mise en tampon
- en-séquence: délivrer (aussi délivrer les paquet du tampon en-séquence), avancer la fenêtre jusqu'au prochain paquet non acquité

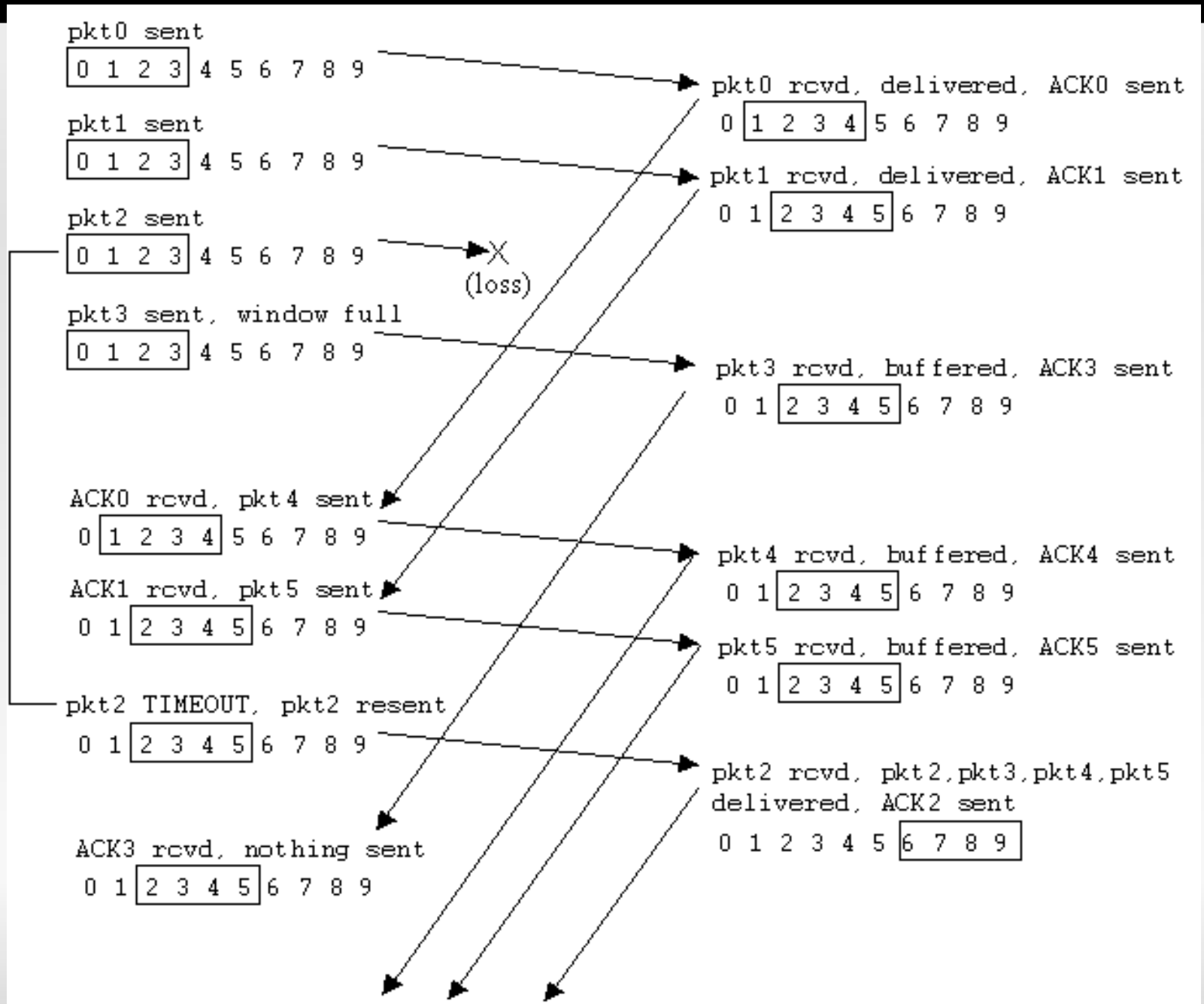
pkt n dans [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignorer

Répétition sélective en action



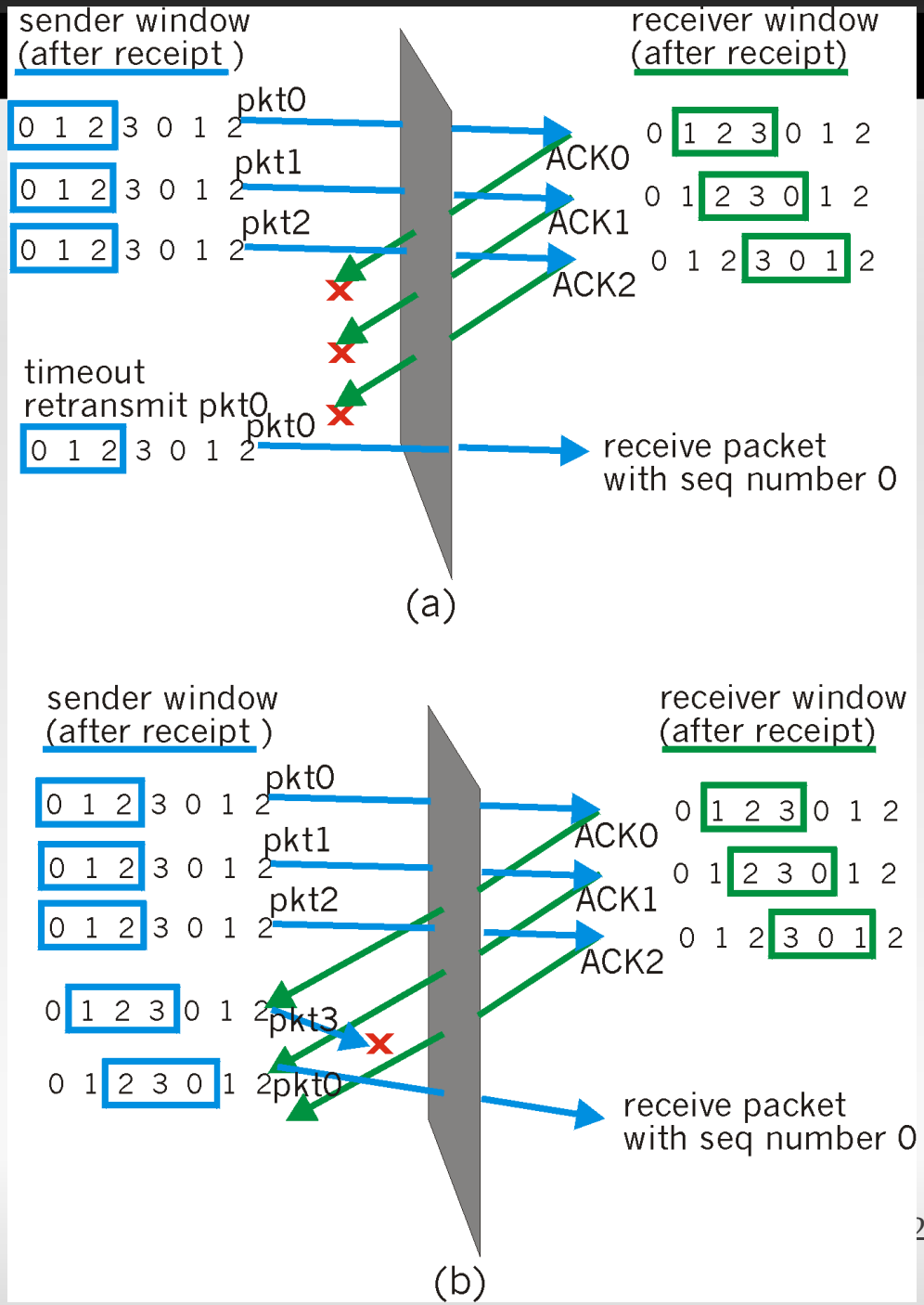
Répétition sélective: dilemme

Exemple:

- seq #'s: 0, 1, 2, 3
- Taille de la fenêtre=3

- Le destinataire voit aucune différence dans deux scénarios!
- Des données dupliquées passent pour nouvelles dans (a)

Q: Quel relation y a-t-il entre seq # size et la taille de la fenêtre?

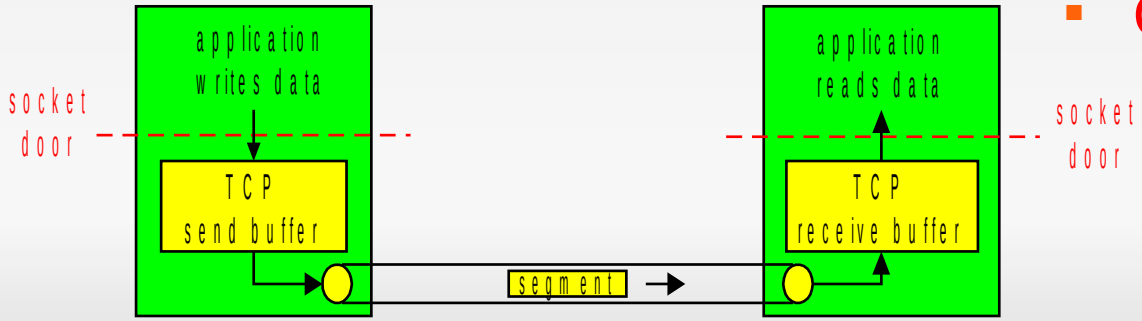


Chapitre 3: Plan

- 3.1 Services de la couche transport
- 3.2 Multiplexge and démultiplexage
- 3.3 Transport sans connexion: UDP
- 3.4 Principe du transfert de données fiable
- 3.5 Transport orienté connexion: TCP
  - Structure des segments
  - Transfert de données fiable
  - Controle de flux
  - Gestion de la connexion
- 3.6 contrôle de congestion
  - Principe
  - Controle de congestion de TCP
- 3.7 Performance de TCP

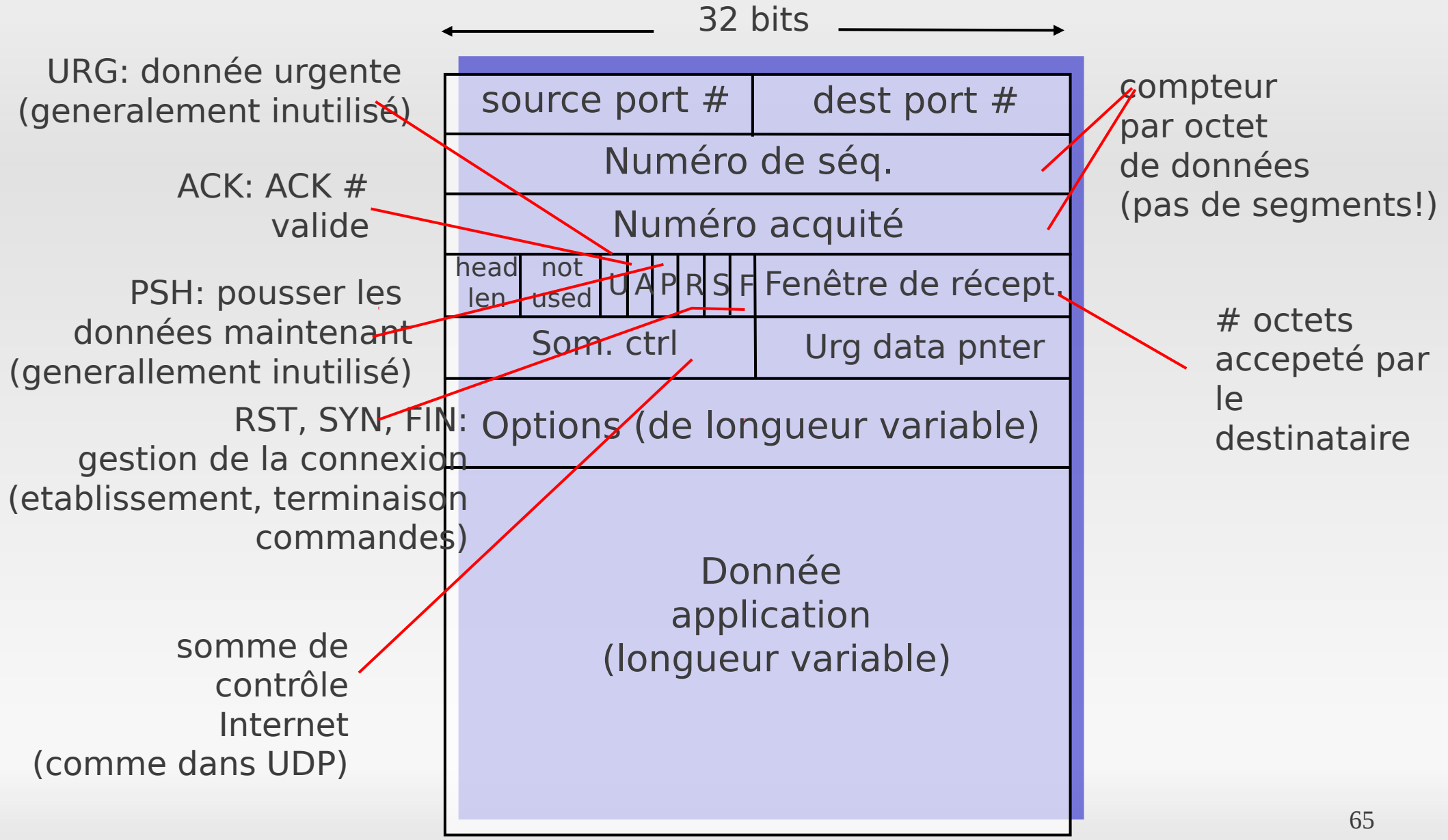
TCP: Vue générale RFCs: 793, 1122, 1323, 2018, 2581

- point-à-point:
  - Un expéditeur, un destinataire
- fiable, en séquence flux d'octet (*byte stream*)*:*
  - Pas de “message boundaries”
- pipeliné:
  - Le contrôle de *congestion et du flux* fixe une taille de fenêtre
- *Tampon d'envoi & réception*



- Donnée en mode duplex:
  - Flux de données bi-directionel dans la même connexion
  - MSS: maximum segment size
- orienté-connexion:
  - Poignée de main (échange de msgs de contrôle) initalise les données d'état expéditeur, destinataire avant l'échange de données
- Contrôle de flux:
  - L'expéditeur ne va pas inonder le destinataire

Structure du segment TCP



Chapitre 3: Plan

- 3.1 Services de la couche transport
- 3.2 Multiplexage and demultiplexage
- 3.3 Transport sans connexion: UDP
- 3.4 Principe du transfert de donnée fiable
- 3.5 Transport orienté connexion: TCP
  - Structure des segments
  - Transfert de données fiable
  - Controle de flux
  - Gestion de la connexion
- 3.6 contrôle de congestion
  - Principe
  - Controle de congestion de TCP
- 3.7 Performance de TCP

Transfert de données fiable de TCP

- TCP crée un service transfert de données fiables (rdt) au dessus du service non fiable d'IP
- Les segments sont pipelinés
- ACKs cumulatif
- TCP utilise une minuterie (timer) pour détecter les pertes

- Les retransmissions sont déclanchés par:
  - timeout
  - ACKs dupliqués
- On considère un expéditeur TCP simplifié:
  - ignore ACKs dupliqués
  - ignore contrôle de flux et contrôle de congestion

Num. de seq. et ACKs TCP

Numéro de séquence:

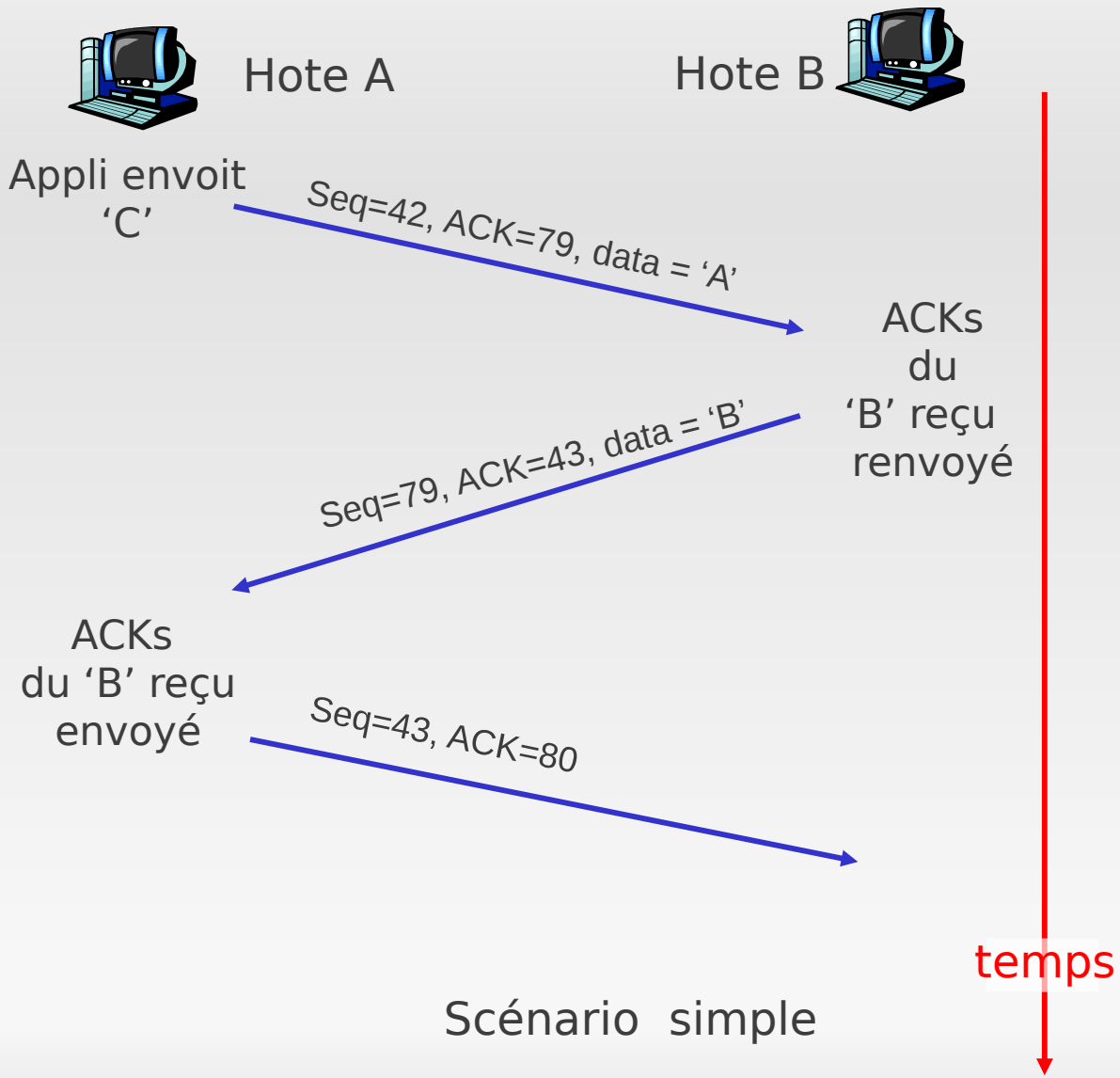
- Fil de “numéro” d'octets du premier octet dans les données du segment

ACKs:

- Num. de seq. du prochain octet **attendu**
- ACK cumulatif (quitte tous octets de numéros de seq inférieurs)

Qu.: Comment le destinataire gère les segments hors-séquence

- Rep: TCP ne le spécifie pas – c'est à la discrétion du développeur



Evènement du côté expéditeur TCP:

Données reçues par l'app:

- Créer un segment avec un N° de seq
- Le N° de seq est le numéro de séquence d'un fil d'octets du premier octet de donnée du segment
- Lancement de la minuterie (timer) si elle n'est pas déjà en train de tourner (penser à la minuterie pour le plus ancien segment non acquité)
- intervalle d'expiration: `TimeOutInterval`

timeout:

- Retransmettre le segment qui a causé le timeout
- Redémarrer la minuterie

ACK reçus:

- S'il acquite des segments précédement non acquité
  - Mettre à jour ce que l'on sait être acquité
  - Démarrer le minuteur s'il reste des segments en suspens

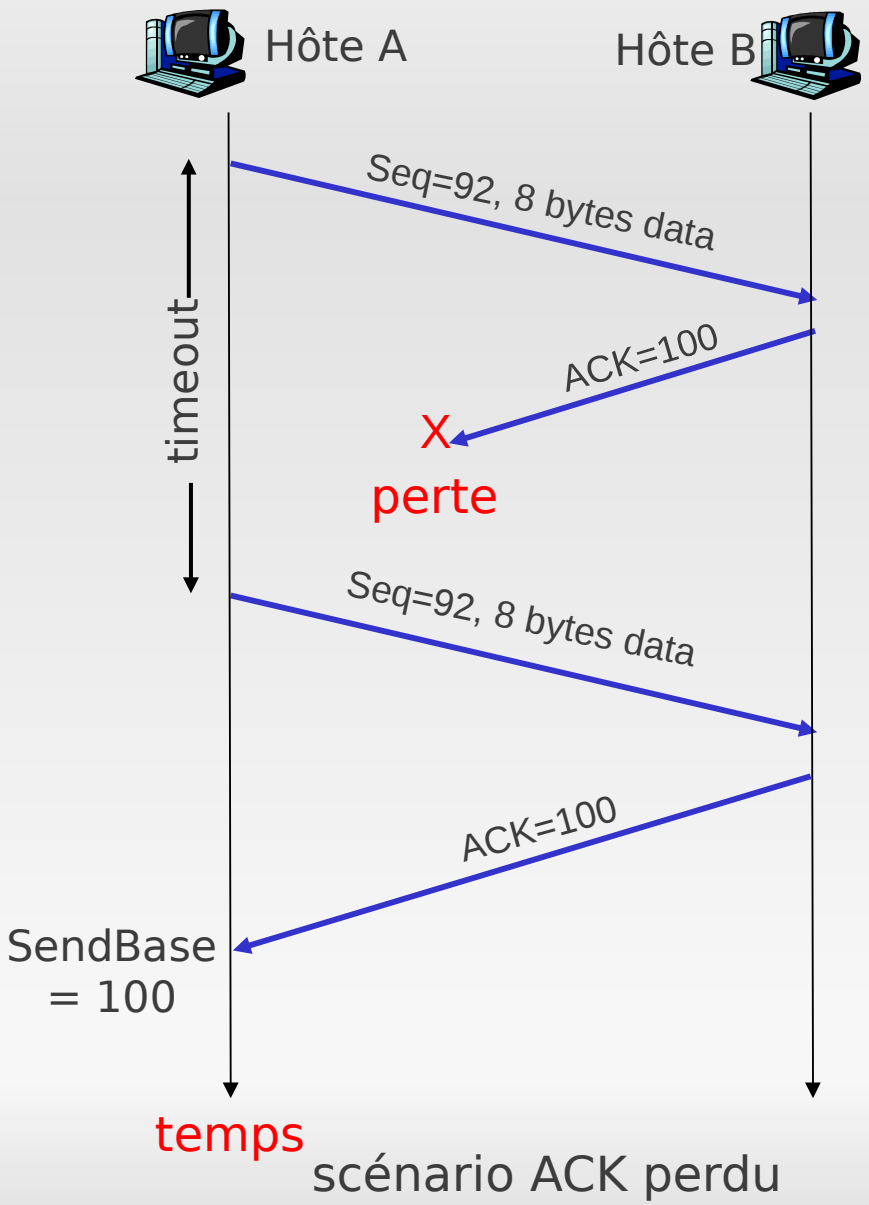
TCP expéditeur(simplifié)

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

loop (forever) {
    switch(event)
    event: data reçu de l'application de la couche supérieure
        cree un segment TCP avec un numéro de séquence NextSeqNum
        if (minuterie arretée)
            demarer la minuterie
        passer le segment à la couche IP
        NextSeqNum = NextSeqNum + length(data)

    event:  timeout de la minuterie
        retransmettre segment pas-encore-acquité  avec
            le plus petit numéro de séquence
        demarrer la minuterie

    event: ACK reçu, avec le valeur du champs ACK de s'y satisfaisant
        if (y > SendBase) {
            SendBase = y
            if (il y a actuellement des  segments pas-encore-acquités)
                Demarrer la minuterie}
} /* fin de boucle  forever */
```

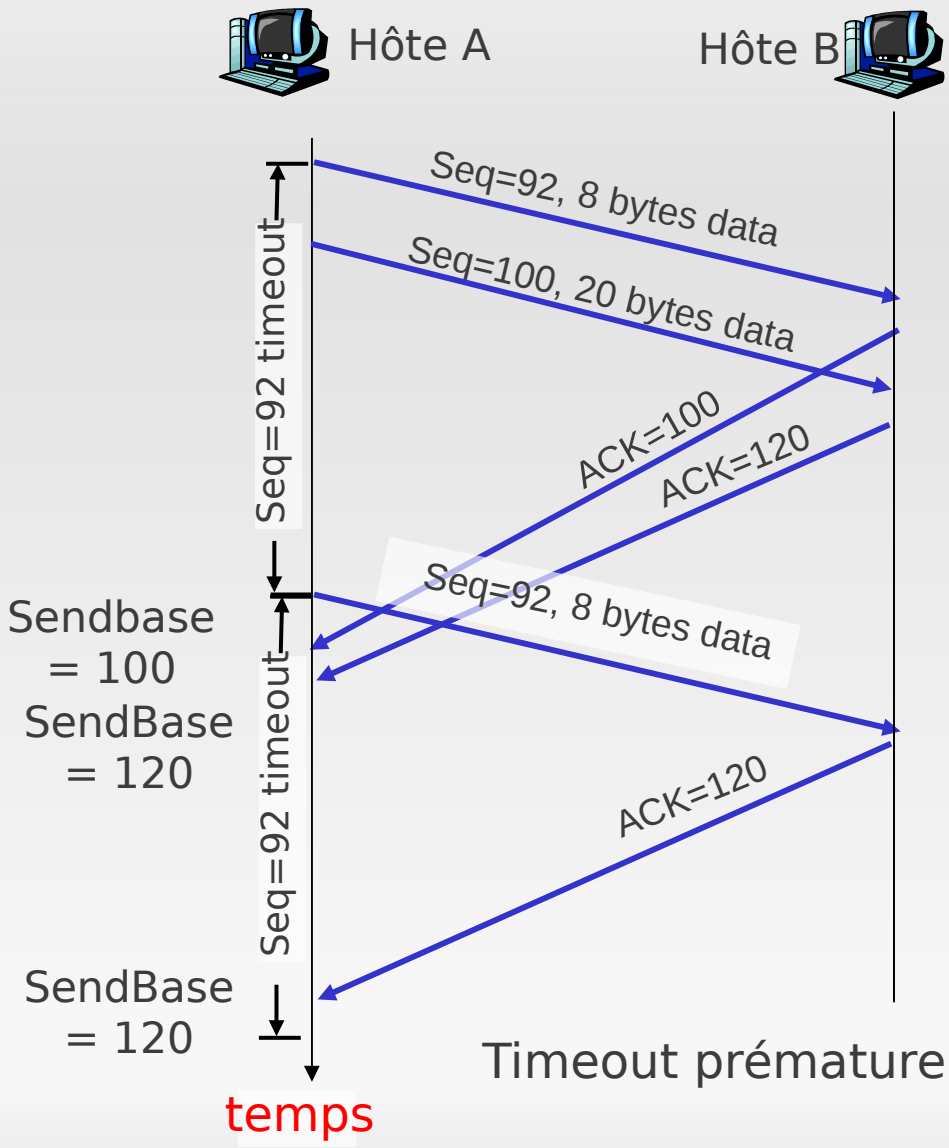
Commentaire:
SendBase-1: dernier octet acquité cumulativement

Exemple:
SendBase-1 = 71;
y= 73, donc le dest. veut 73+ ;
y > SendBase, donc cette nouvelle donnée est acquitée.

TCP: scénarios de retransmission

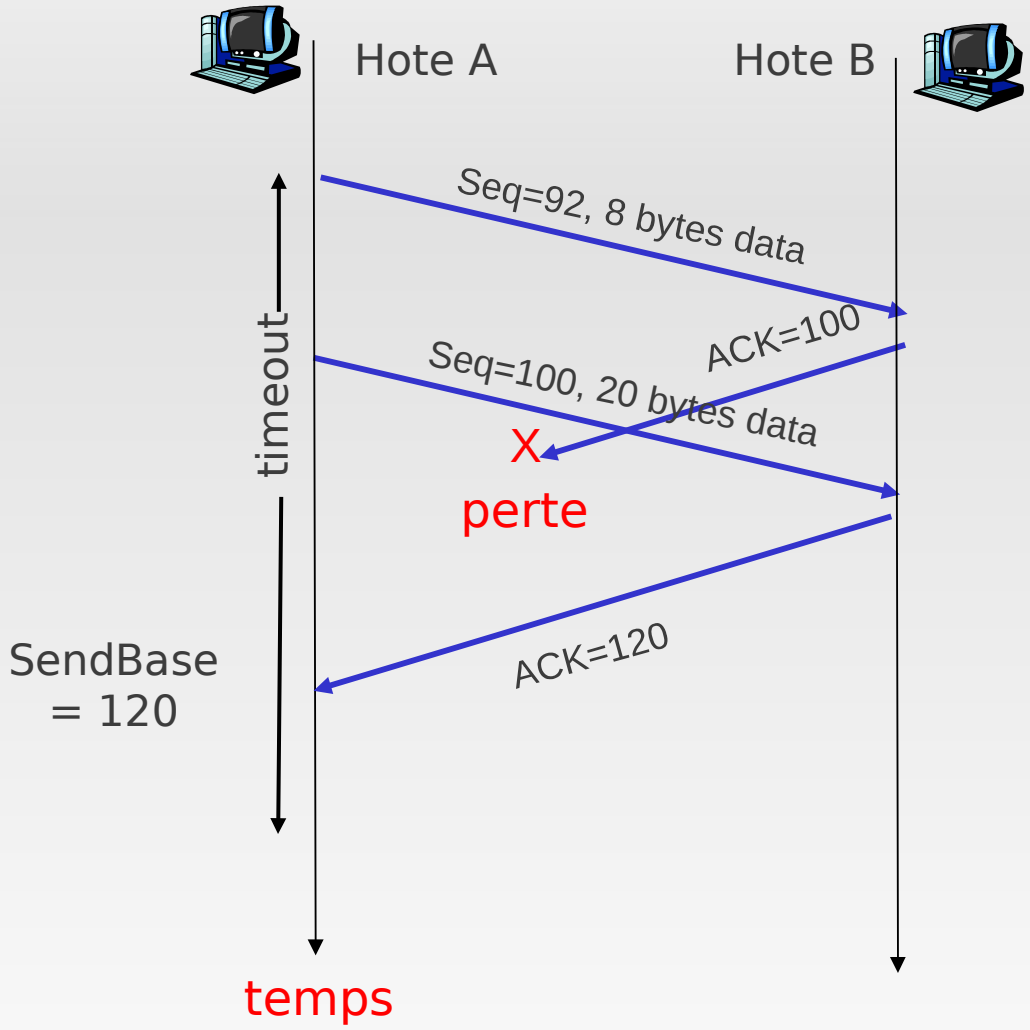


scénario ACK perdu



Timeout prématuré

TCP scénarios de retransmission (suite)



scénario ACK accumulatif

TCP : émission d'un ACK [RFC 1122, RFC 2581]

Evènement chez destin.	Action TCP du dest.
Arrive d'un segment dans l'ordre avec un seq # attendu. Toutes les données jusqu'à ce seq # ont été ACKée.	Génération retardée de l'ACK: attendre jusqu'a 500ms pour envoyer l'ACK. Si aucun segment n'arrive durant ces 500ms alors envoyer l' ACK.
Arrivé d'un seqment dans l'ordre avec le N° de seq attendu. Un autre segment a un ACK non encore envoyé.	Envoyer immédiatemment d'un ACK cumulatif. Accuse réception des deux segments arrivés dans l'ordre (celui dont l'envoie de l'ACK est retardé + celui qui vient d'arriver).
Arrivé d'un segment dans le désordre. D'un plus haut N° de seq que celui attendu. Un écart est détecté.	Envoyer immédiatemment un *ACK dupliqué* indiquant le numéro de seq. du prochain octet attendu
Arrivé d'un segment qui comble partiellement l'écart.	Envoyer immédiatemment un ACK, si le segment augmente la valeur de base window de l'expéditeur

Retransmission rapide

- Hors-délai période souvent relativement longue:
  - Un délai long avant le renvoie d'un paquet perdu
- Segments perdu détectés par réception d'ACKs dupliqués.
  - L'expéditeur envoie souvent beaucoup de segment successivement.
  - Si un segment est perdu, il y aura probablement beaucoup d'ACKs dupliqués.
- Si l'expéditeur reçoit 3 ACKs pour la même donnée, cela suppose que le segment après une donnée aquittée a été perdue:
  - retransmettre: réenvoyer un segment avant que le timeout ne se produise.

Algorithme de retransmission rapide:

```
event: ACK recu, avec un champs ACK egal y
        if (y > SendBase) {
            SendBase = y
            if (il y actuellement des segments pas ecore aquites)
                start timer
          }
        else {
            incrementer un compteur count d'ACK dupliques recu pour y
            if (le compteur d'y = 3) {
                reenvoyer le segment de numero de seq y
             }
```

Un AC dupliqué pour des segments déjà acquitté

retransmission rapide

Aller-retour TCP: temps et Timeout

Q: Comment fixer la valeur du timeout TCP ?

- Plus grand que le RTT
  - mais le RTT varie...
- Trop court: timeout prématuré
  - Retransmissions inutiles
- Trop long: lente réaction à une perte de paquet

Q: Comment estimer le RTT?

- **SampleRTT**: mesure le temps à partir de la transmission d'un segment transmission jusqu'à la réception d'un ACK
  - Ignorer la retransmission
- **SampleRTT** variera, on veut estimer un RTT plus lisse
  - Moyenne pondérée sur plusieurs mesures récentes, pas juste un **SampleRTT** courant

Round Trip Time et Timeout de TCP

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

- Moyenne exponentielle pondérée
- L'influence des échantillons du passé décroit exponentiellement rapidement
- valeur typique: **α = 0.125**

Aller retour TCP: temps et timeout

Calcul du timeout

- **`EstimtedRTT`** plus une “marge de sécurite”
  - Grande variation de **`EstimatedRTT ->`** plus grande marge de sécurité
- D'abord estimer de combien SampleRTT dévie de EstimatedRTT:

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

(valeur typique: β = 0.25)
```

Ensuite fixe l'intervalle de timeout:

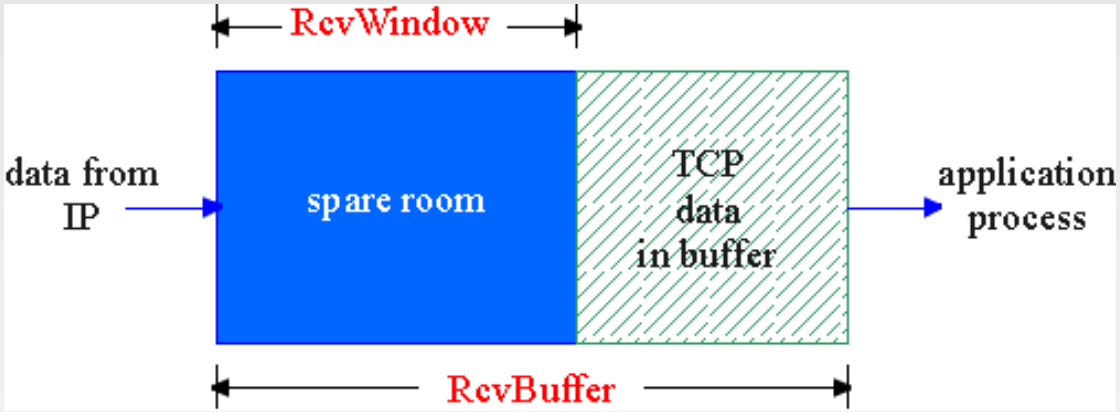
```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

Chapitre 3: Plan

- 3.1 Services de la couche transport
- 3.2 Multiplexage and demultiplexage
- 3.3 Transport sans connexion: UDP
- 3.4 Principe du transfert de donnée fiable
- 3.5 Transport orienté connexion: TCP
  - Structure des segments
  - Transfert de données fiable
  - Controle de flux
  - Gestion de la connexion
- 3.6 contrôle de congestion
  - Principe
  - Controle de congestion de TCP
- 3.7 Performance de TCP

Contrôle de flux de TCP

- Le coté destinateur de la connexion TCP à un tampon mémoire de:



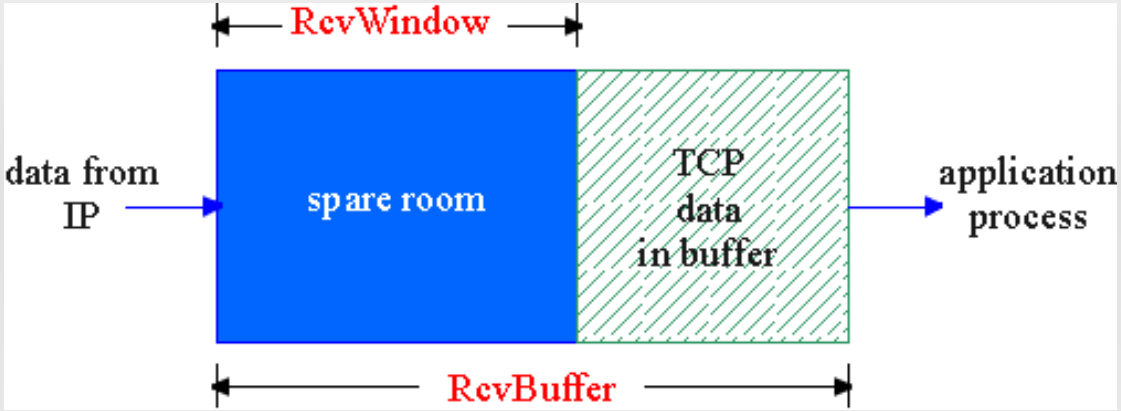
☒ Le processus peut lire lentement les données dans la lecture des données sur le tampon

Control de flux

L'expéditeur ne va pas surcharger le tampon du destinataire en transmettant trop de données et trop vite

- Service d “adaptation de la vitesse de trans.”: sert à adapter la vitesse de transmission à la rapidité de lecture du processus.

Contrôle de flux de TCP: suite



(Suppose que le destinataire TCP rejette les segments hors-séquence)

- De l'espace libre dans le tampon

  = RcvWindow

  = RcvBuffer-[LastByteRcvd - LastByteRead]

- Le destinataire avertit la taille de l'espace libre en includant la valeur de **RcvWindow** dans les segments
- L'expéditeur limite le nombre de donnée non acquitée à **RcvWindow**
  - Guarantie que le tampon de réception ne déborde pas

Chapitre 3: Plan

- 3.1 Services de la couche transport
- 3.2 Multiplexage and demultiplexage
- 3.3 transport sans connexion: UDP
- 3.4 Principe du transfert de donnée fiable
- 3.5 Transport orienté connexion: TCP
  - Structure des segments
  - Transfert de données fiable
  - Controle de flux
  - Gestion de la connexion
- 3.6 contrôle de congestion
  - Principe
  - Controle de congestion de TCP
- 3.7 Performance de TCP

Gestion de la connexion TCP

Rappel: L'expéditeur TCP,

établit une “connexion” avant l'échange de données

- Initialisation de variables TCP:
  - seq. #
  - tanpons, info. de contrôle de flux (e.g. **`RcvWindow`**)
- *Client:* initiateur de connexion

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

- *Serveur:* contacté par le client

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

Poignée de main en trois étapes:

Etape 1: le client envoie un segment TCP SYN au serveur

- spécifie le seq # initial du client
- Ne contient pas de données

Etape 2: le serveur reçoit SYN, répond avec un segment SYNACK

- Serveur alloué des tampons de mémoire
- spécifie le seq. # initial du serveur

Etape 3: le client reçoit SYNACK, répond avec un segment ACK segment, qui peut contenir des données.

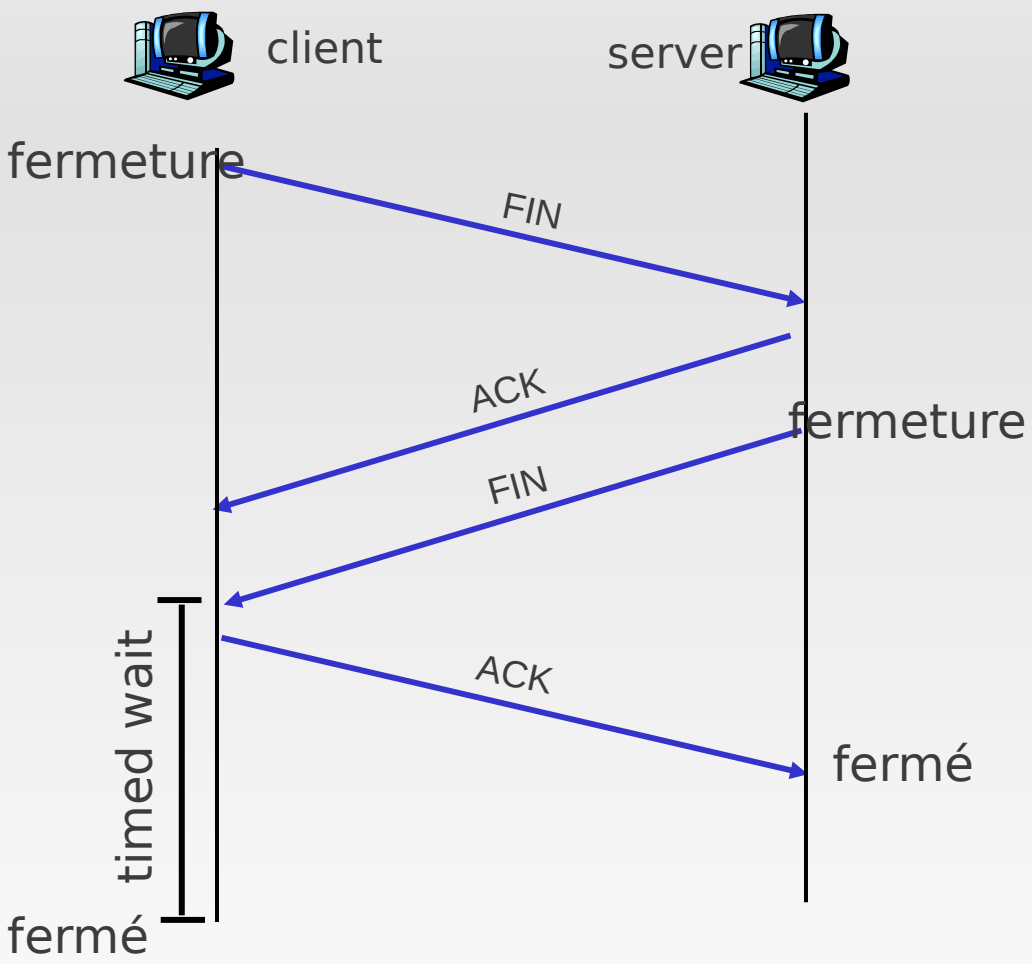
Gestion de la connexion TCP (suite)

Etape 3: le client reçoit FIN, répond avec un ACK.

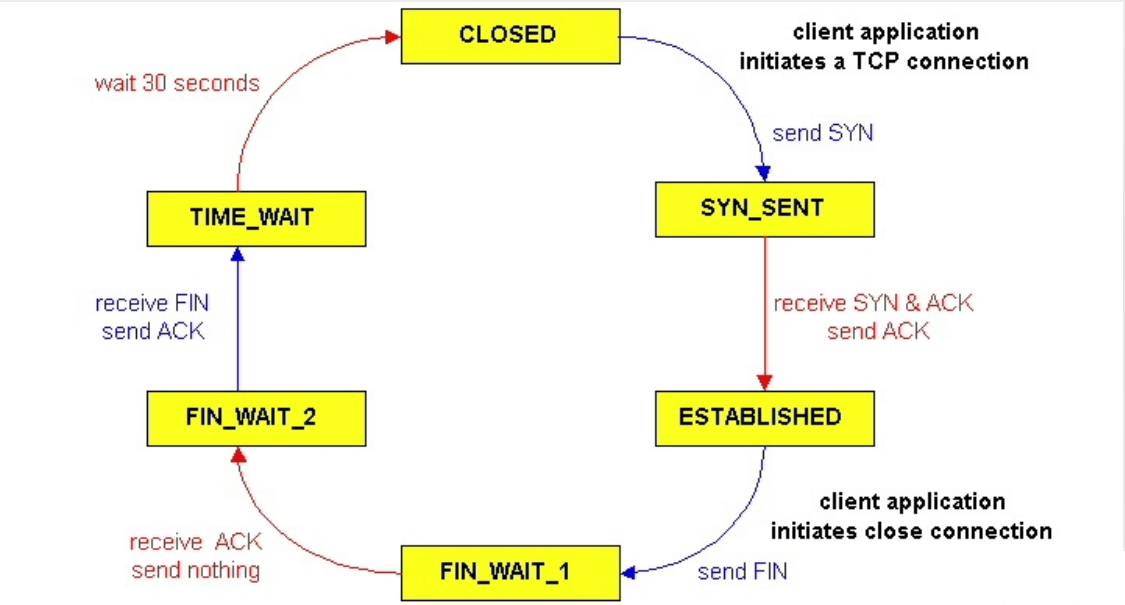
- Entre dans une phase “timed wait” - répondra avec un ACK pour des FINs reçus

Etape 4: le serveur reçoit un ACK. connexion fermée.

Note: avec de petites modifications, peut gérer des FINs simultanés.

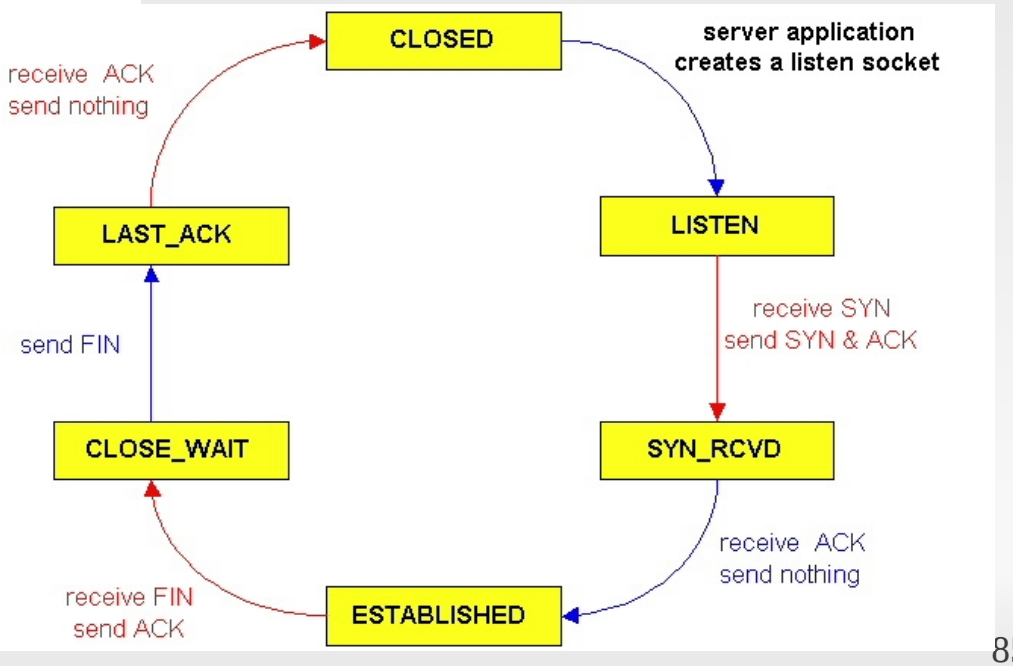


Gestion de la connexion TCP (suite)



Le cycle de vie
du client TCP

Le cycle de vie
du seveur TCP



Chapitre 3: Plan

- 3.1 Services de la couche transport
- 3.2 Multiplexage and demultiplexage
- 3.3 transport sans connexion: UDP
- 3.4 Principe du transfert de donnée fiable
- 3.5 Transport orienté connexion: TCP
  - Structure des segments
  - Transfert de données fiable
  - Controle de flux
  - Gestion de la connexion
- 3.6 contrôle de congestion
  - Principe
  - Controle de congestion de TCP
- 3.7 Performance de TCP

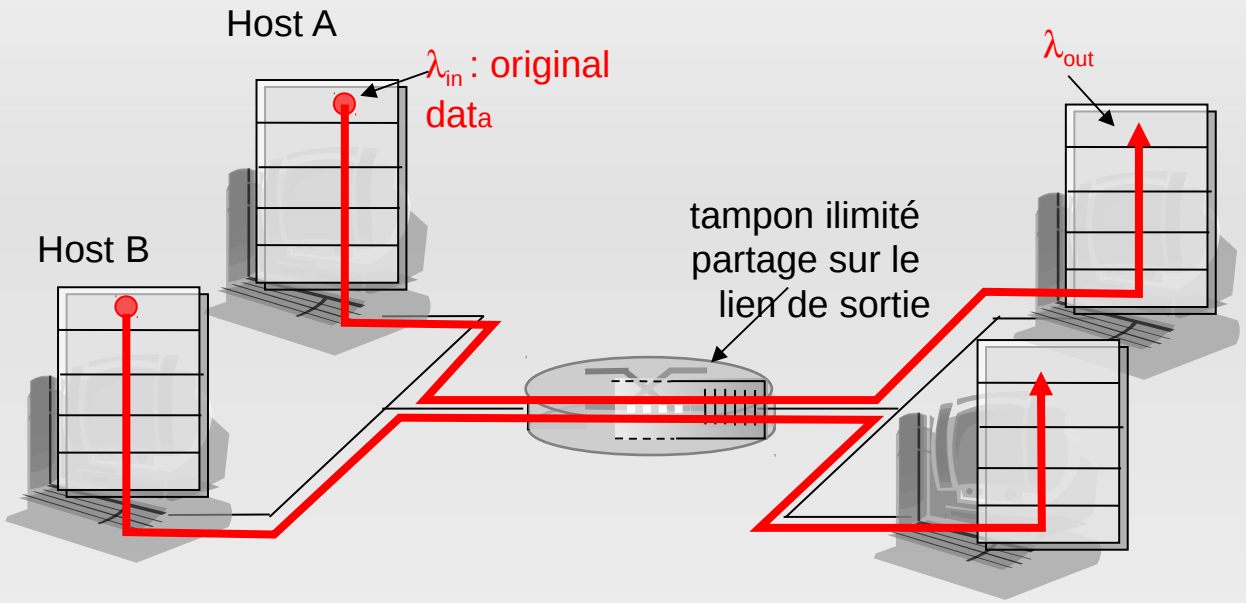
Principe du contrôle de congestion

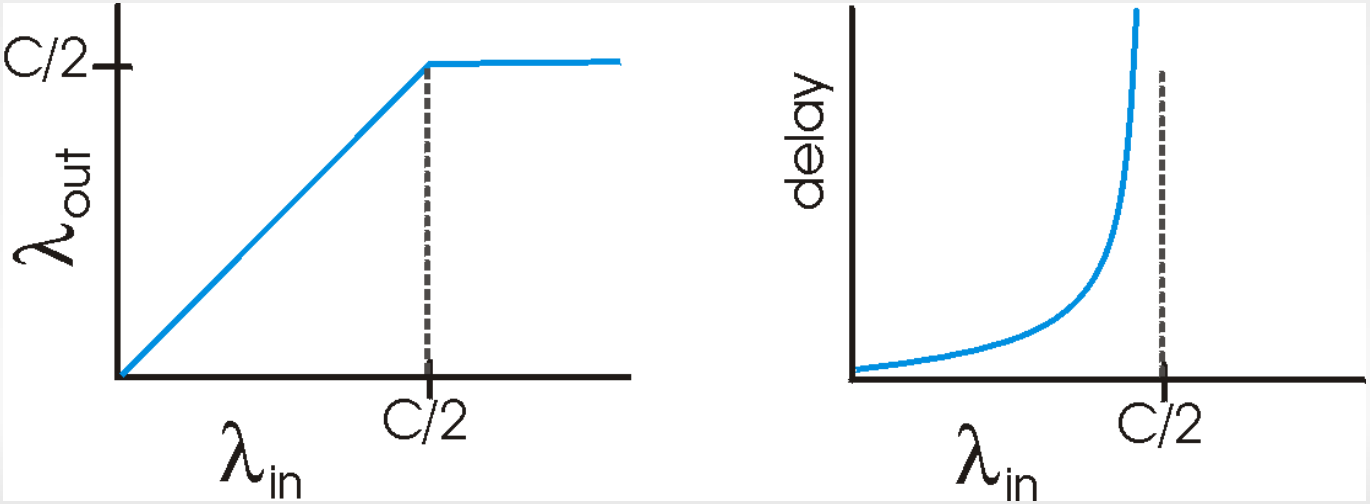
Congestion:

- informellement: “trop de sources qui envoient trop de données trop vite pour que le réseau puisse les traiter ”
- différent du controle de flux!
- manifestations:
  - Perte de paquets (débordement de tampon au niveau des routeurs)
  - Délais longs (les queues s'allongent dans les tampons des routeurs)

Causes/coûts de la congestion: scénario 1

- deux expéditeur, deux destinataires
- un routeur, des tampons infinis
- Pas de retransmission

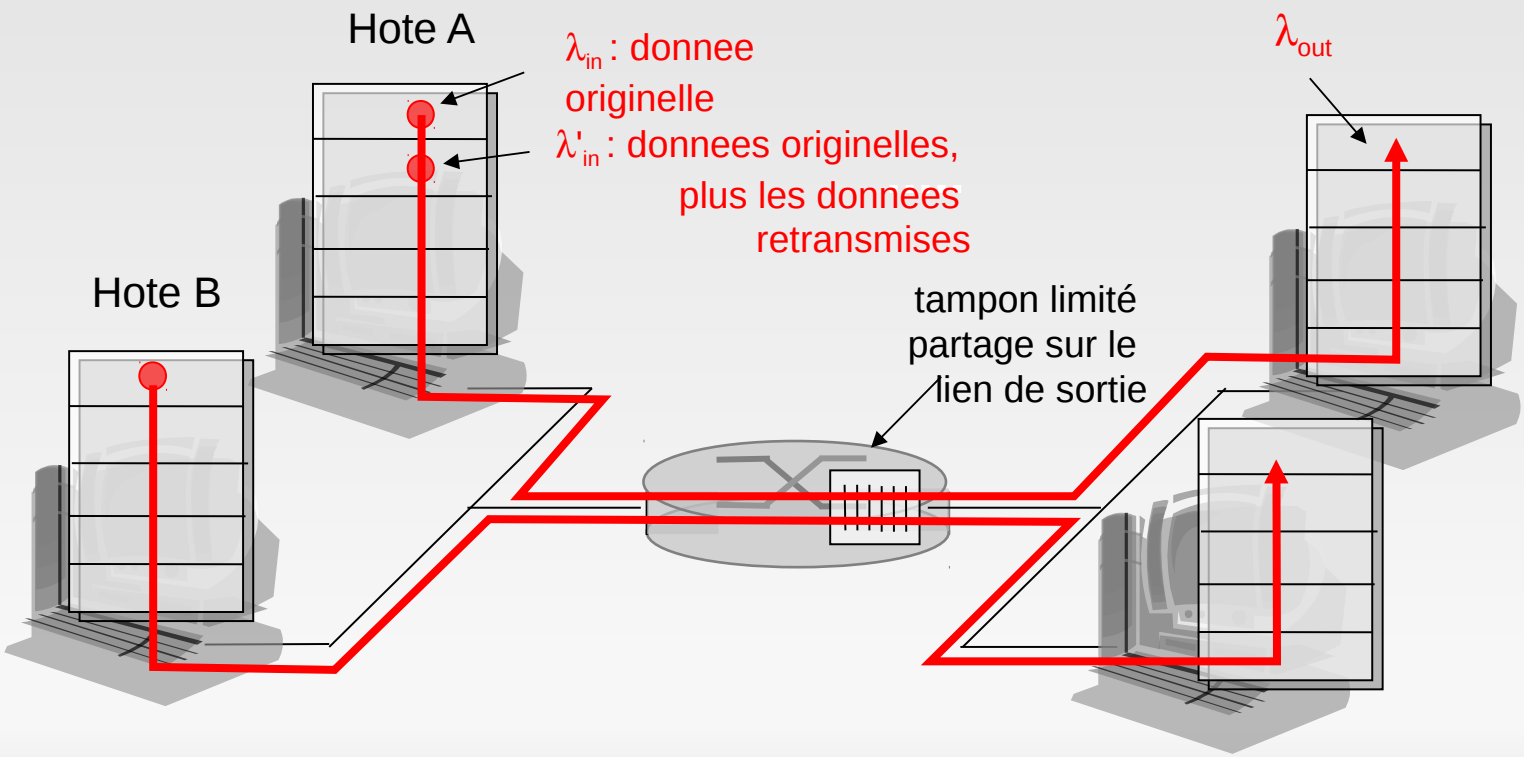




- De très long délai lors d'une congestion
- Transmission maximum faisable

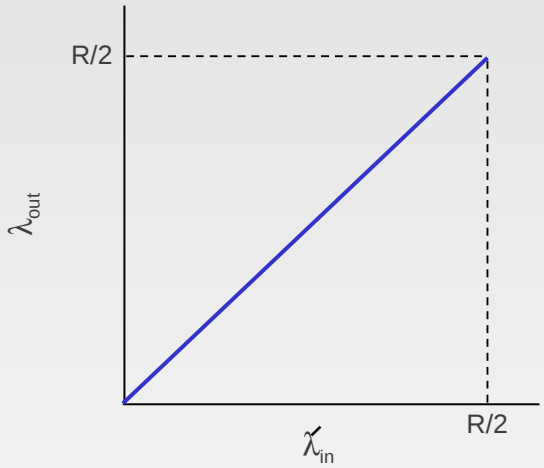
Causes/coûts de congestion: scénario 2

- Un routeur, des tampons *finis*
- L'expéditeur retransmet les paquets perdus

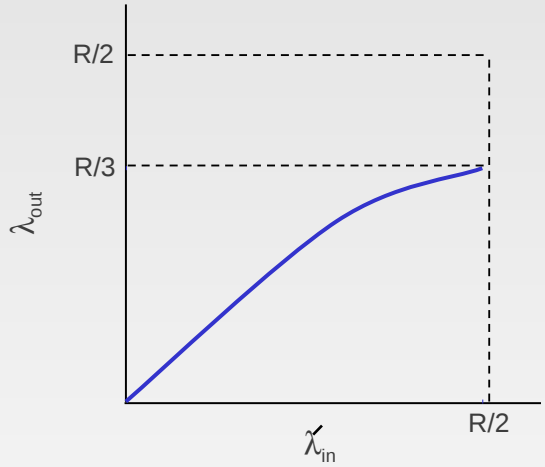


Causes/coûts de congestion: scénario 2

- toujours: $\lambda_{in} = \lambda_{out}$ (meilleurs cas)
- Retransmission “parfaite” seulement lorsqu'il y a des pertes: $\lambda'_{in} > \lambda_{out}$
- retransmission de paquets retardés (mais pas perdus) rend plus grande λ'_{in} (que dans le cas parfait) pour la même λ_{out}



a.



b.

“Coûts” de la congestion:

- ☒ Plus de travail retransmission
- ☐ des retransmissions non nécessaires: le lien transporte de nombreuses copies des paquets.

Méthodes de contrôle de la congestion

Deux approches répandues sont:

Control de congestion de bout en bout:

- Pas de retour direct du réseau
- congestion gérée par les hôtes observant des pertes, des retards
- approche utilisée dans TCP

Contrôle de congestion assisté par le réseau:

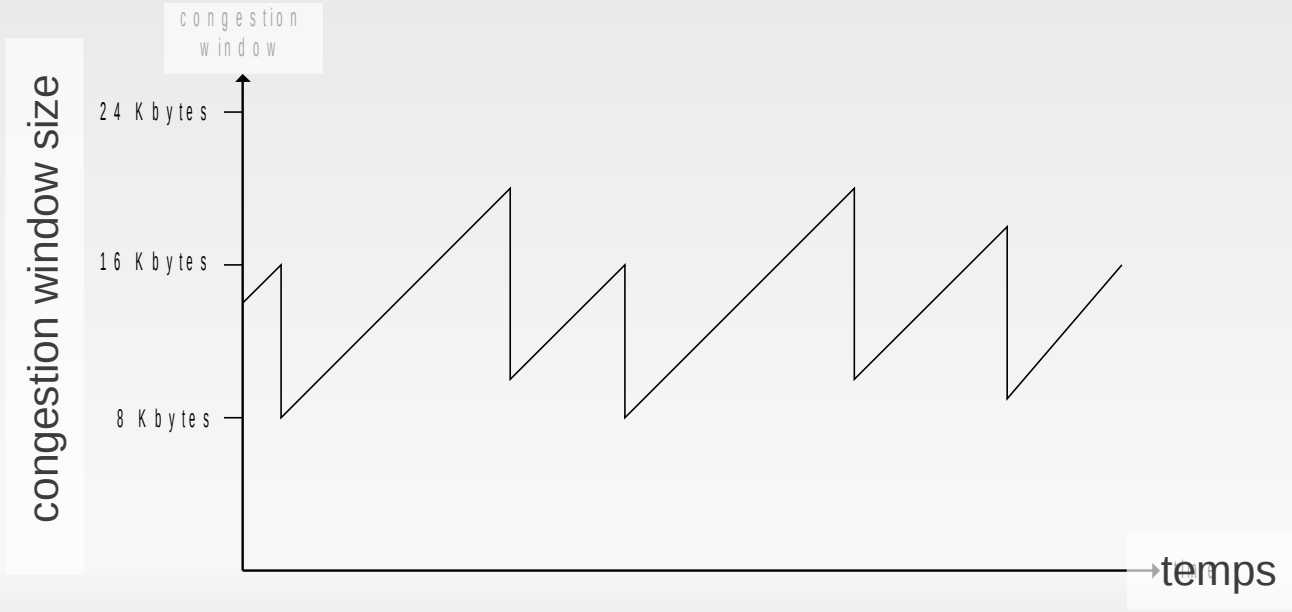
- Les routeurs fournissent des informations aux hôtes
  - simple bit indiquant la congestion (SNA, DECbit, TCP/IP ECN, ATM)
  - Taux explicite auquel l'expéditeur doit envoyer ses données
- Approche utilisée dans ATM

Contrôle de congestion de TCP:

accroissement additive, décroissance multiplicative

- *Approche:* augmenter le taux de transmission (taille de la fenêtre), sondant la bande passante utilisable, jusqu'à ce qu'une perte de paquet se produise
  - *Augmentation additive:* acroît **CongWin** d'1 MSS après chaque RTT jusqu'à détecter une perte
  - *Reduction multiplicative*: réduit la **CongWin** de moitier après une perte

Comportement en Dent de scie



Contrôle de congestion TCP : détails

- L'expéditeur limite la transmission:

  `LastByteSent-LastByteAcked`

  `≤ CongWin`

- Plus grossièrememt,

$$\text{rate} = \frac{\text{CongWin}}{\text{RTT}} \text{ Bytes/sec}$$

- **`CongWin`** est dynamique, dépent de la congestion perçuue du réseau

Comment l'expéditeur perçoit la congestion ?

- Une perte = timeout *ou* 3 ACKs dupliqués
- Expéditeur TCP réduit son taux (**`CongWin`**) après la détection d'une perte

Trois mécanismes:

- AIMD
- Départ lent
- Conservatif après un timeout

Départ lent de TCP

Lorsque une connexion commence, **`CongWin`** = 1 MSS

- Exemple: MSS = 500 octets & RTT = 200 msec
- Taux initial = 20 kbps

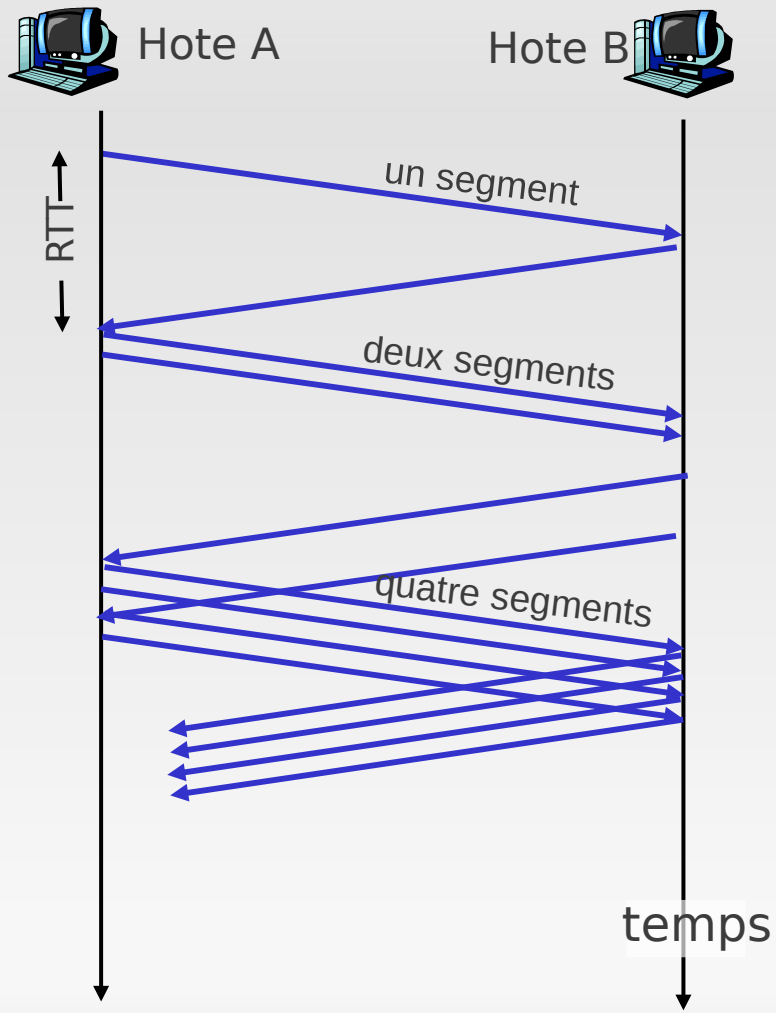
- Bande passante disponible peut être >> MSS/RTT
  - Il est nécessaire d'accéder rapidement à un taux de transmission respectable

☒ Lorsque la connexion commence, accroissement exponentiel du taux jusqu'à la détection d'une perte

Départ lent de TCP (suite)

- Lorsque la connexion commence, accroissement exponentielle du taux

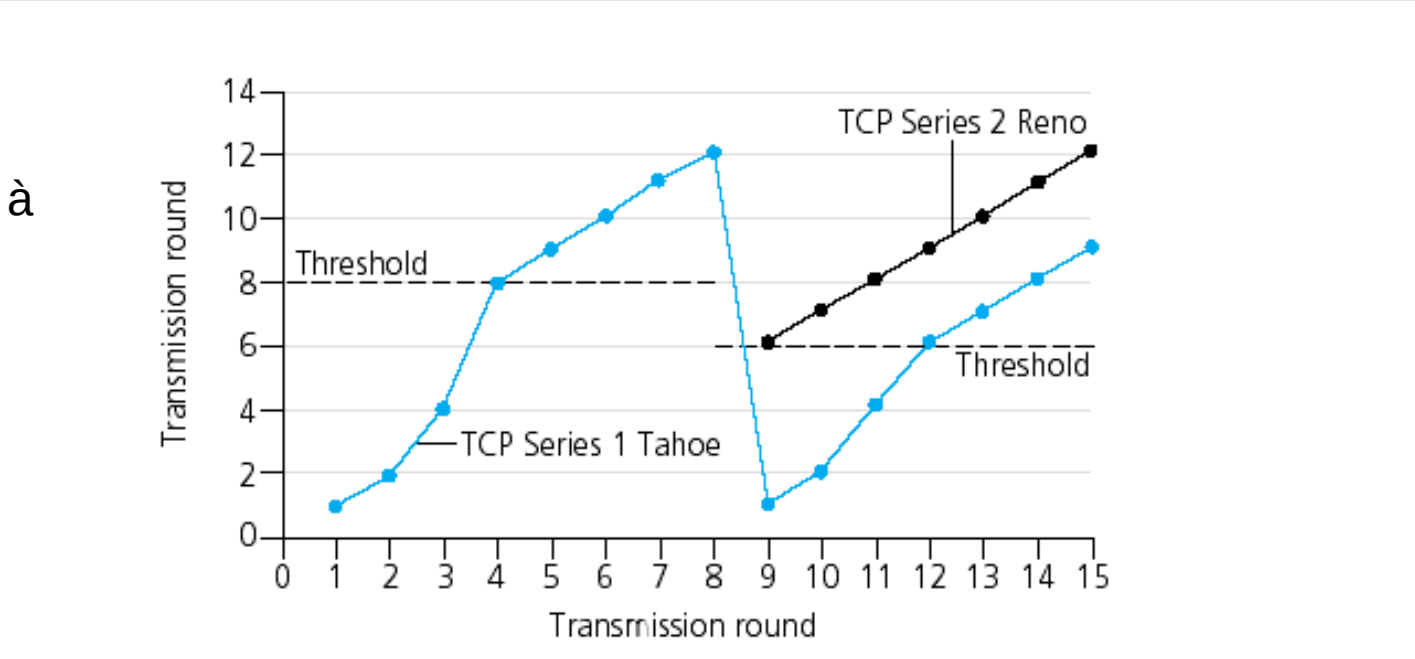
  jusqu'à la détection d'une perte:
  - double **`CongWin`** à chaque RTT
  - Réalise en incrémentant **`CongWin`** pour chaque ACK reçu
- Resumé: taux initial est faible mais augmente de façon exponentielle



Raffinement

Q: A partir de quand la phase exponentielle doit être remplacée par une phase linéaire?

A: Lorsque **`CongWin`** arrive à la moitier de sa valeur avant le timeout.



Implémentation:

- Variable Threshold (seuil)
- Lorsqu'une perte se produit, Threshold est fixé à la moitier de lavaleur de CongWin avant la détection de la perte

Raffinement: détection des pertes

- Après 3 ACKs dupliqués:
  - **`CongWin`** est coupé en deux
  - La fenêtre augmente alors linéairement
- Mais après un timeout:
  - **`CongWin`** est fixé à 1 MSS;
  - Le fenêtre augmente alors exponentiellement
  - Jusqu'au seuil **Threshold**, ensuite grossit linéairement

Philosophie:

- 3 ACKs dupliqués indiquent que le reseau est capable de transmettre certains segments
- timeout indique une congestion “plus alarmante”

Résumé: contrôle de congestion de TCP

- Lorsque **`CongWin`** est en dessous de **`Threshold`**, l'expéditeur est en phase de départ-lent, la fenêtre augmente exponentiellement.
- Lorsque **`CongWin`** est au dessus de **`Threshold`**, l'expéditeur est en phase d'évitement de congestion augmente linéairement.
- Lorsqu'un triple ACK dupliqué se produit, **`Threshold`** est fixé a **`CongWin/2`** et **`CongWin`** est fixé à **`Threshold`**.
- Lorsqu'un timeout se produit, **`Threshold`** est fixé à **`CongWin/2`** et **`CongWin`** est fixé à 1 MSS.

Chapitre 3: Plan

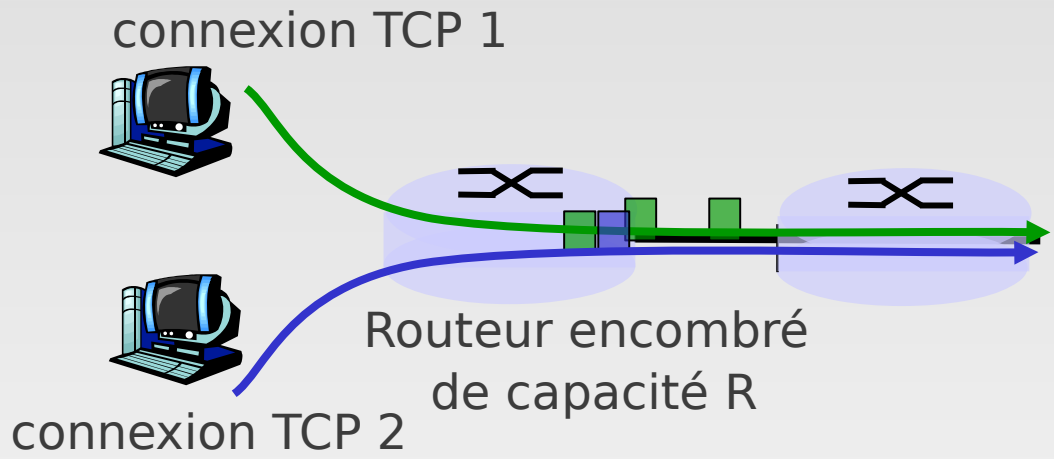
- 3.1 Services de la couche transport
- 3.2 Multiplexage and demultiplexage
- 3.3 transport sans connexion: UDP
- 3.4 Principe du transfert de donnée fiable
- 3.5 Transport orienté connexion: TCP
  - Structure des segments
  - Transfert de données fiable
  - Controle de flux
  - Gestion de la connexion
- 3.6 contrôle de congestion
  - Principe
  - Controle de congestion de TCP
- 3.7 Performance de TCP

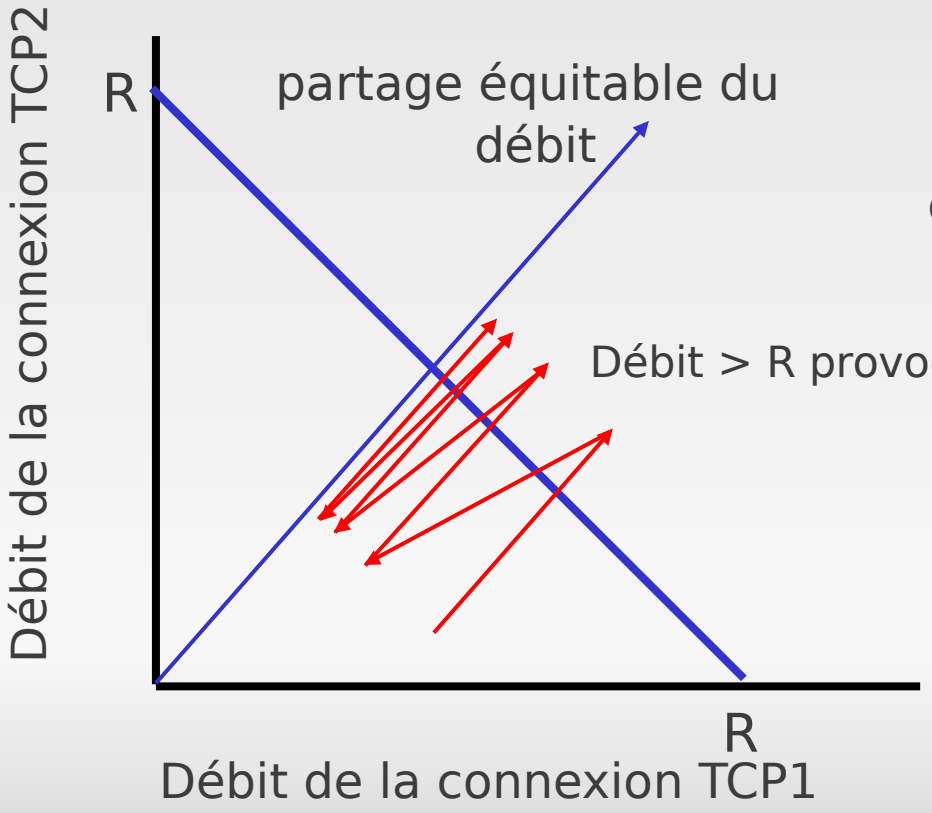
Taux de transmission TCP

- Quel est le taux de transmission moyen de TCP en fonction de la taille de la fenêtre et d'un RTT?
  - On ignore le “départ-lent”
- Soit W la taille de la fenêtre lorsqu'une perte se produit (si le réseau est stable cette valeur est toujours de même grandeur).
- Lorsque la fenêtre vaut W, le taux de transmission est W/RTT
- Juste après une perte, la taille de la fenêtre descend a W/2, et le taux de transmission à W/2RTT.
- Taux de transmission moyen est de: .75 W/RTT

Équité

- K connexions TCP partagent le même lien, chacune devraient avoir 1/K de ce lien
- addition d'1 Unité
  multiplication par 0.5





Débit > R provoque une perte: divise la fenêtre par 2

- Faussée car par un navigateur ouvrant plusieurs connexion TCP pour une même page web