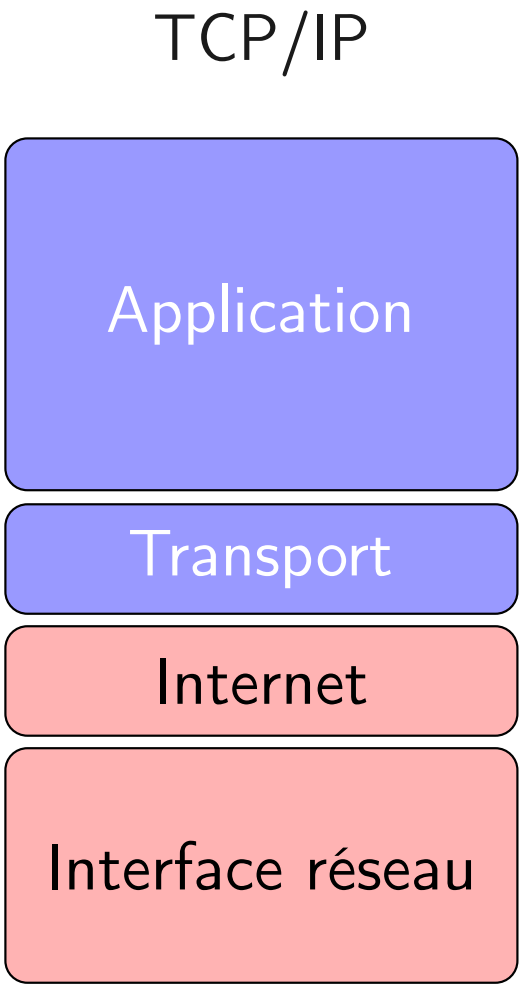
# Couche Transport

*Sophie Robert, Université d'Orléans*

# La couche transport




- La couche transport
  - A partir des caractéristiques de la couche Réseau
  - La communication logique entre les processus
  - Les services rendus
  - Les différents protocoles
  - TCP - UDP

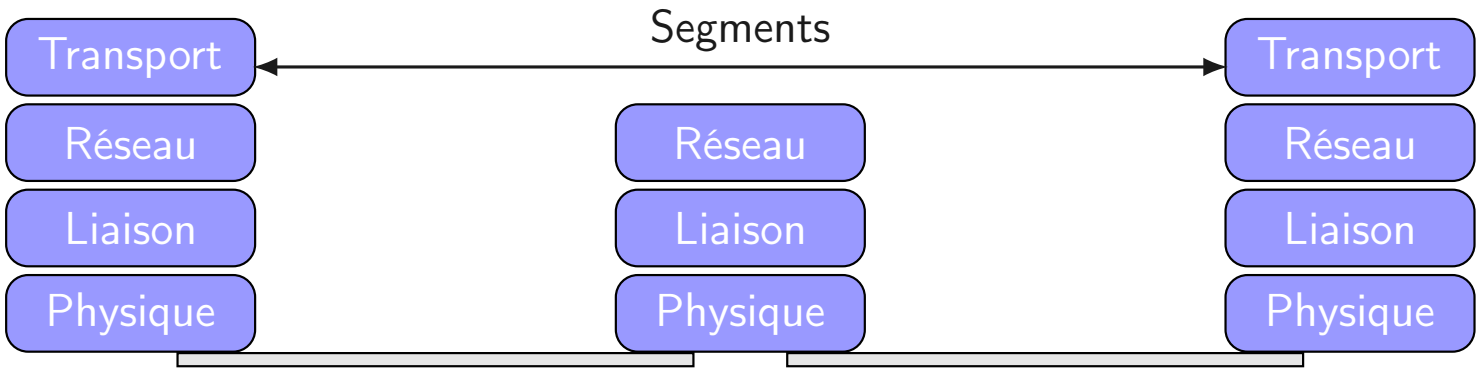
# Couche réseau (Network)

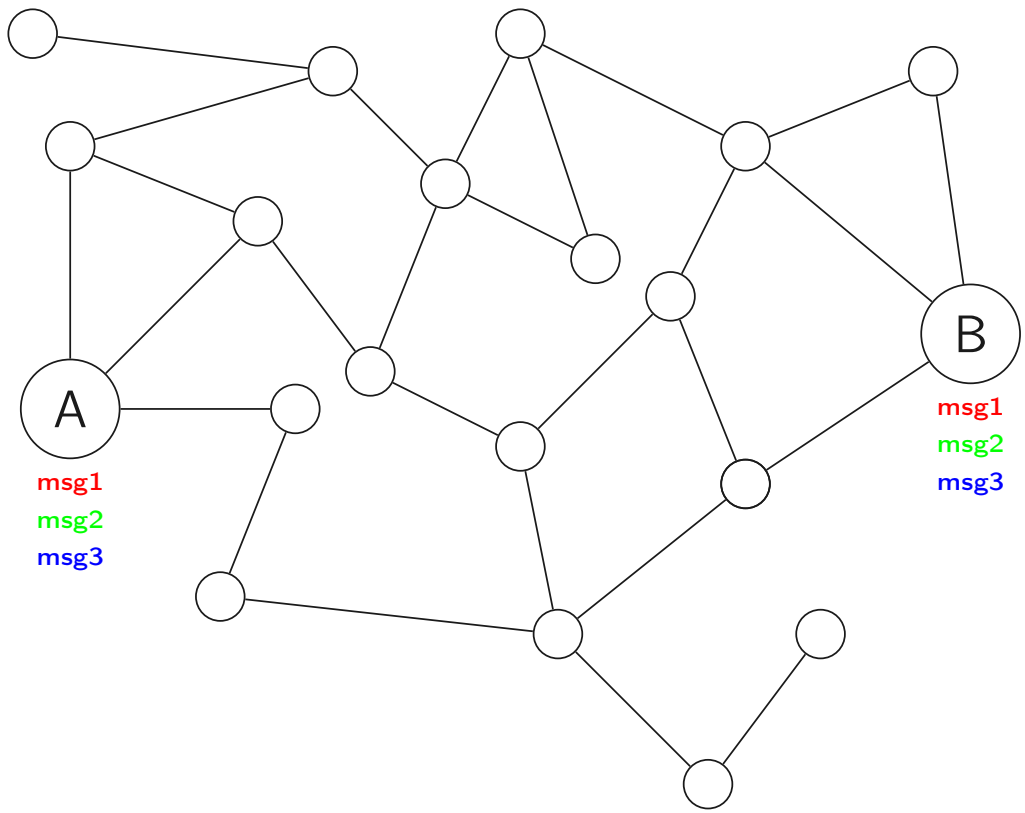
- Lien entre des points distants avec intermédiaires.
- Spécification des adresses pour le routage.
- Pertes, corruptions ou inversions potentielles de parties de messages.

# Couche transport (Transport)

- De nouveau, liaisons entre des points distants, mais sans intermédiaire.
- Retour d'une communication fiable, sans erreur (**TCP**).
- Mode connecté (segments) ou non (datagrammes).

# Couche transport (Transport)



# Problématique

La couche réseau permet la transmission de paquets entre deux machines distantes.

Deux **processus** différents sur la machine $A$ veulent parler à deux **processus** sur la machine $B$.

La couche réseau ne connaît que les **interfaces réseau**...
Comment distinguer les **processus** d'une même machine qui partagent une **même interface réseau** ?

# Solution

La couche transport ajoute une notion de **ports**.

Pour **UDP** et **TCP** un entier codé sur 16 bits (ports réservés 0-1023, enregistrés 1024-49141, dynamiques 49152-65535).

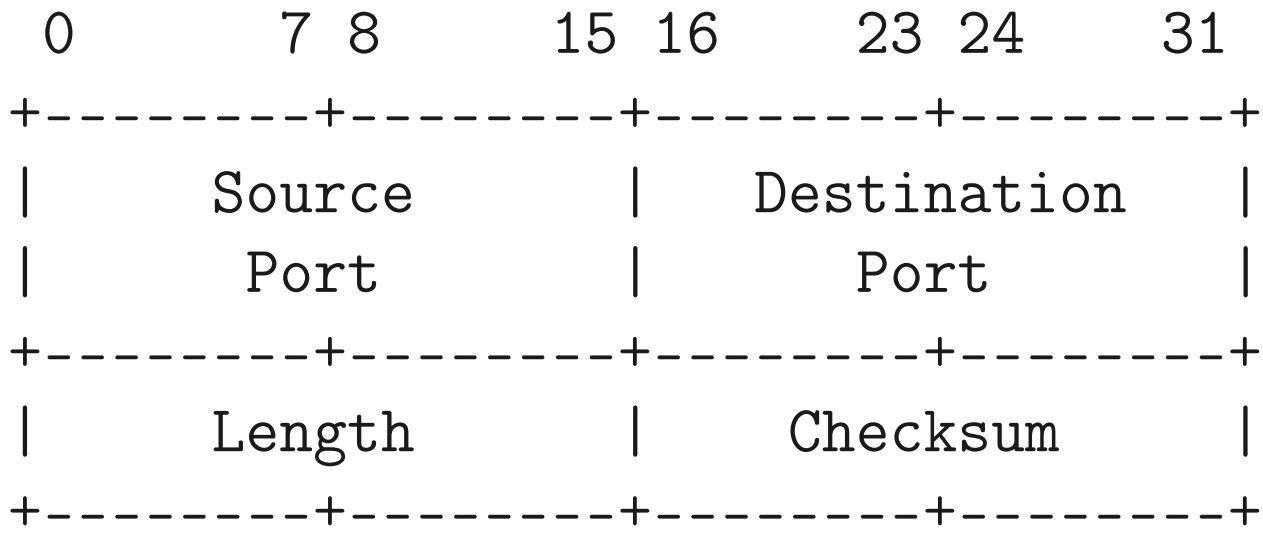
Communication identifiée par un **quadruplet**

(Adresse Réseau A, Port A, Adresse Réseau B, Port B)

# UDP

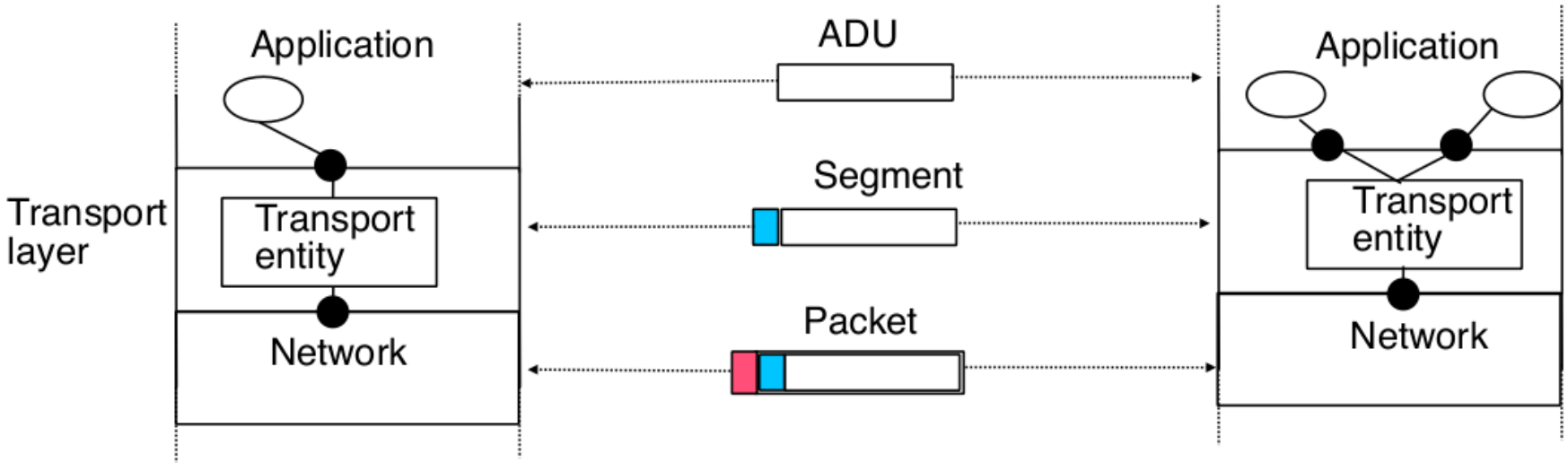
Ajoute les **ports** à IP, et un code détecteur d'erreurs.

Encapsulé dans IP.



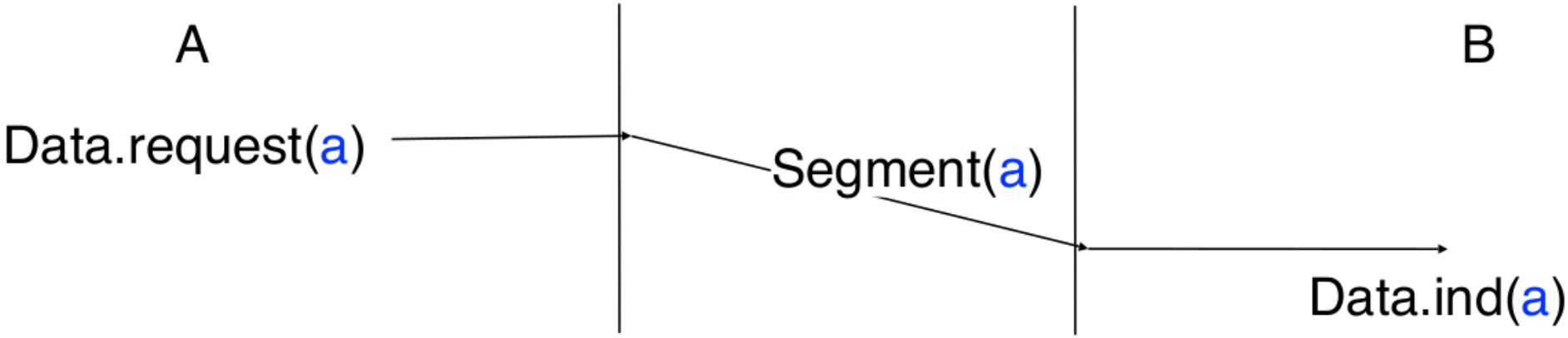

Checksum : complément à 1 de la somme des compléments à 1 des segments de 16 bits de (UDP + pseudo-header IP)

# Les protocoles de la couche transport




- SDU Service Data Unit
- Ajout d'un entête

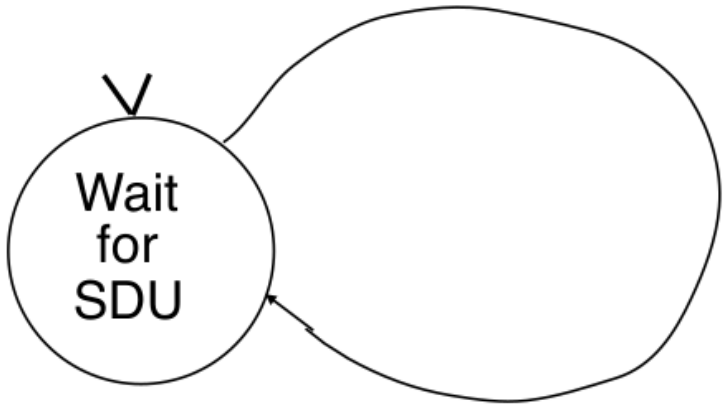
# Construction du protocole : les définitions




- *data.req/data.ind* primitives couche application
- *send/recvd* primitives couche réseau

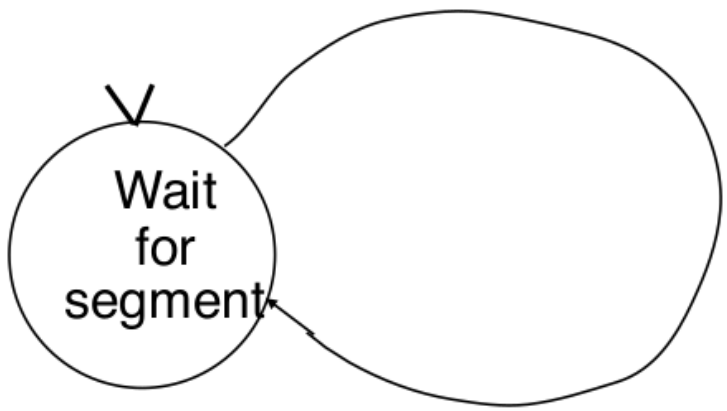
# Description par une machine à états finis

## Sender




`Data.req(SDU)`
`send(SDU)`

## Receiver




`recvd(SDU)`
`Data.ind(SDU)`

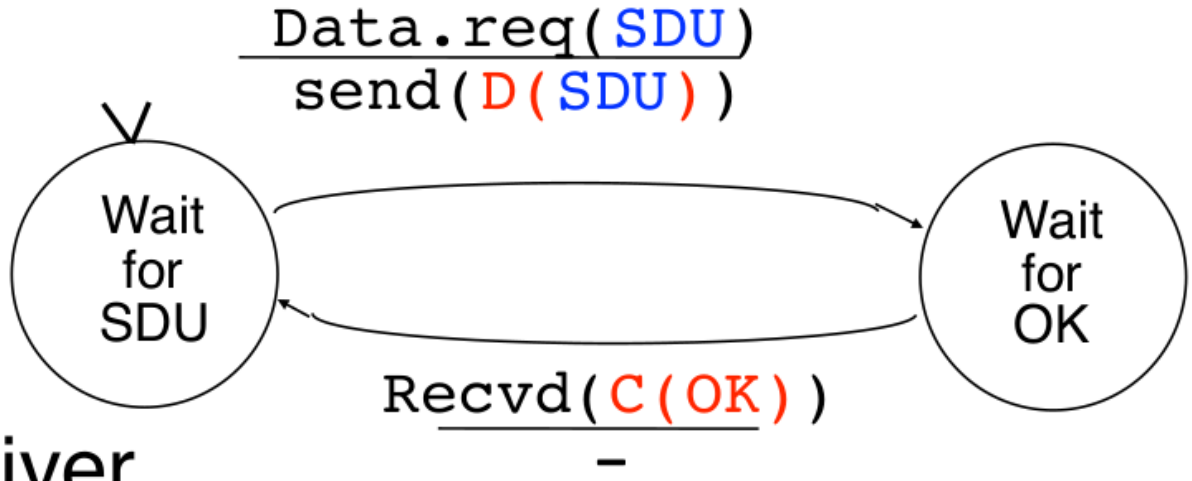
# Protocole : le contrôle

Que se passe-t-il si le *receiver* est 10 fois plus lent que le *sender* ?

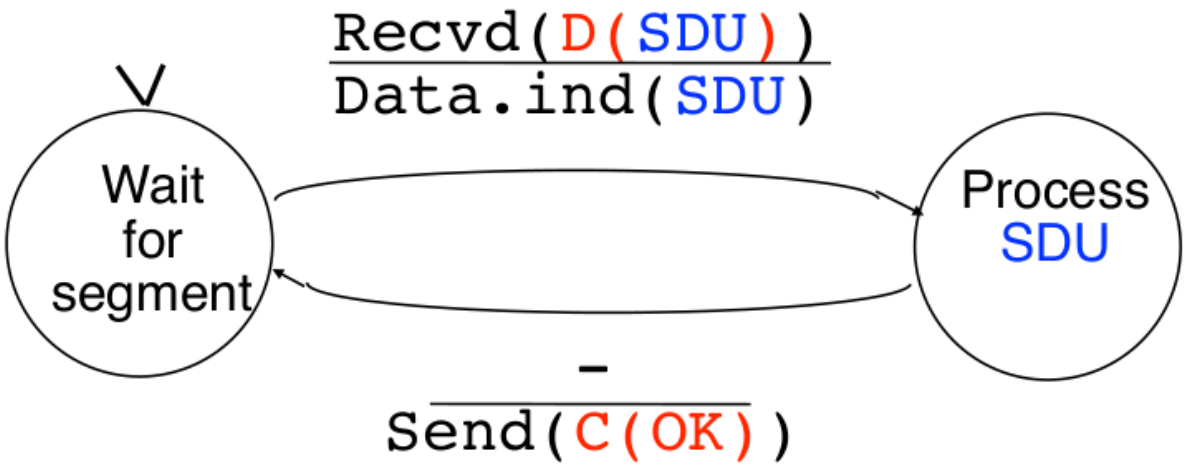
- Aux segments de données `Data(SDU)` on va rajouter des segments de contrôle `C(OK)`
- Dans l'entête au moins un bit indique quel est le type de segment Data ou Contrôle
- Boucle d'échanges entre le *sender* et le *receiver*

# Protocole : le contrôle

## Sender

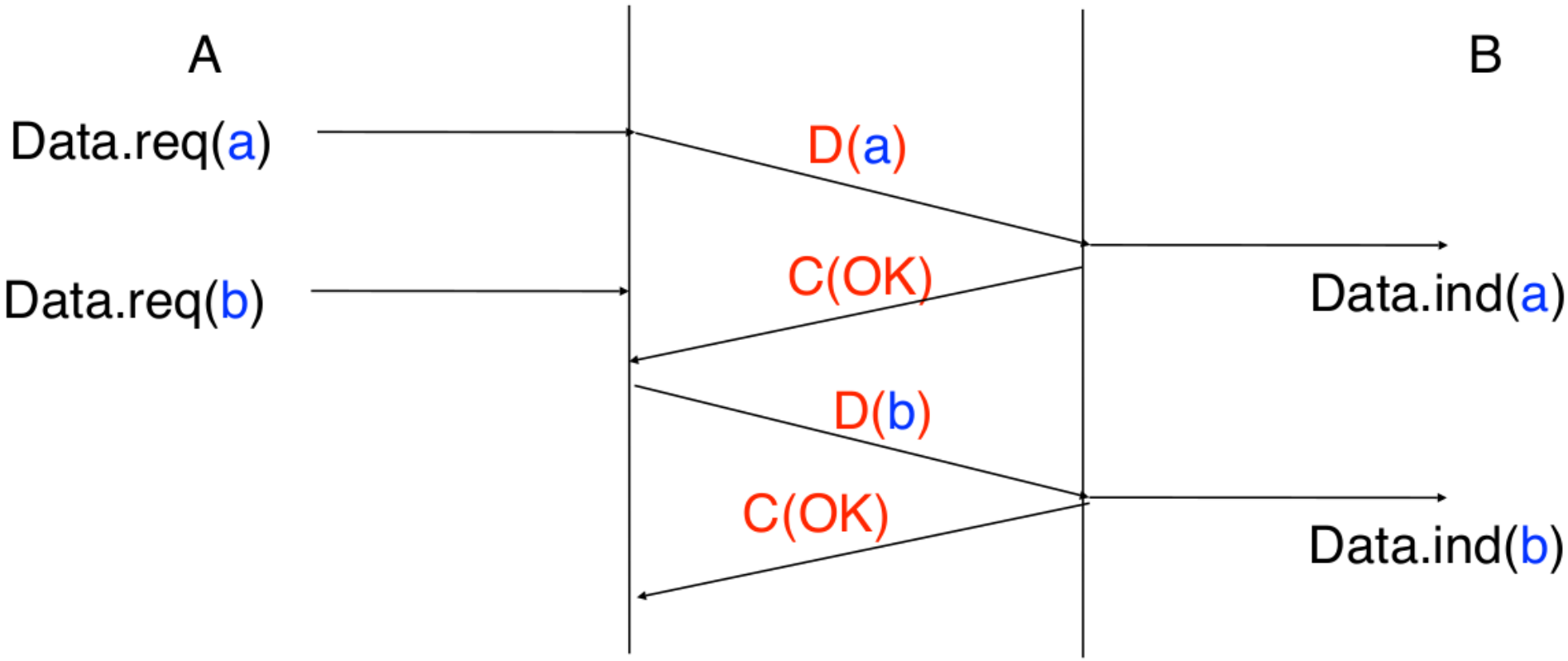



## Receiver



# Protocole : le contrôle

le *sender* transmet des données lorsqu'il a reçu l'autorisation du *receiver*




Chronogramme du *Stop and Wait*

# Protocole : pour un service fiable

Hypothèses

- Les SDU sont petits
- La qualité de service de la couche réseau
  1. les erreurs de transmission sont possibles
  2. il n'y a pas de perte
  3. il n'y a pas de duplication
- la transmission est unidirectionnelle

# Les erreurs de transmission

- Au niveau de la transmission sur le support (la couche physique)
  - Une erreur aléatoire isolée (un bit est inversé)
  - Un groupe de $n$ bits erronés (burst errors)

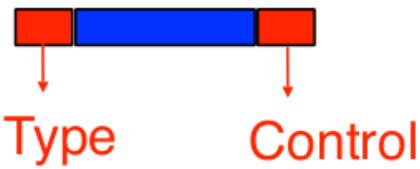
  * Les autres sources de corruption de données (attaque,...) sont traitées en dehors de la couche transport.

# Comment détecter ces erreurs ?

- Le *sender* ajoute des informations de contrôle au segment
- Le *receiver* vérifie ces informations

$\rightarrow$ Ces informations sont calculées à partir du segment construit

$\rightarrow$ Le *receiver* doit pouvoir les recalculer

# Le bit de parité

- Très simple (serial lines)
- Pour un groupe de $n$ bits rajouter un bit afin d'avoir un nombre pair de 1 dans le groupe de $n + 1$ bits

- On ne peut que détecter

0011010 1

1101100 0

# Internet Checksum

- Le segment avec entête est organisé en mots de 16 bits
- On construit la somme en complément à 1 de ces mots
- Le complément à 1 du résultat (16 bits) est ajouté au segment

- Le *receiver* effectue la même opération et doit trouver $0xFFFF$

# CRC Cyclical Redundancy Check

- Utilisation d'un code polynomial $g$ de degré $r$
- Génération de $r$ bits de redondance à partir du segment de longueur $d$ bits
- Le *sender* envoie $d + r$ bits (segment + redondance)

- Le *receiver* calcule la division modulo 2 par $g$
- Il doit trouver 0

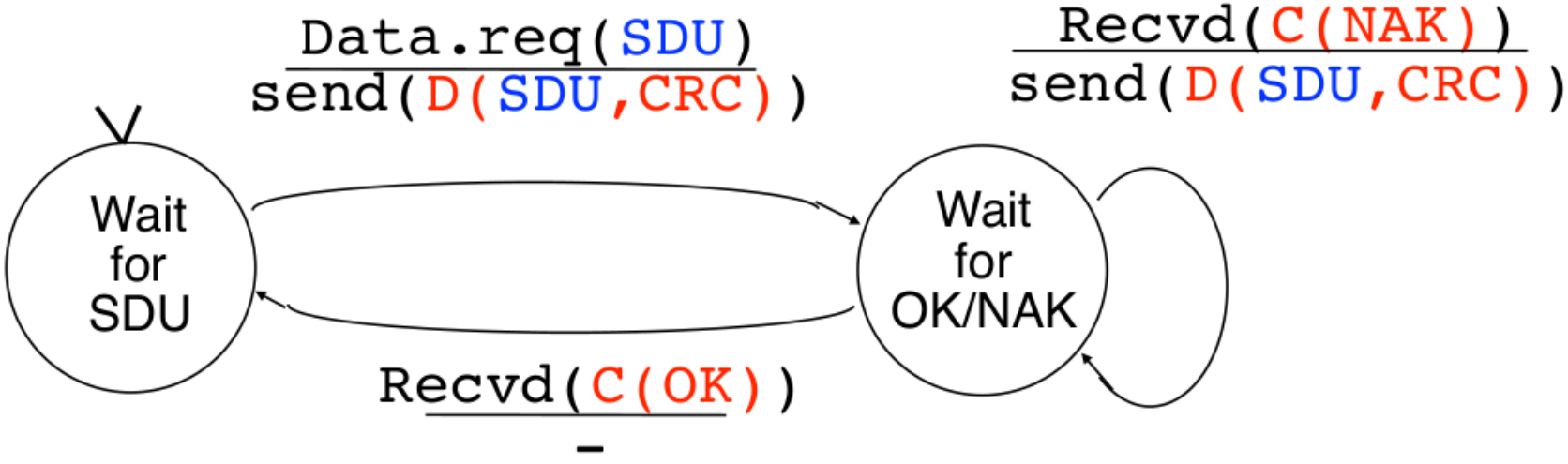
# Protocole : pour un service fiable

- *receiver* $\rightarrow$ Après vérification du *Checksum*
  1. S'il est correct le *receiver* envoie C(OK) pour confirmer la réception et autoriser le *sender* à poursuivre
  2. S'il est incorrect le *receiver* le rejette (pas de data.ind) et envoie un nouveau message de contrôle C(NAK)

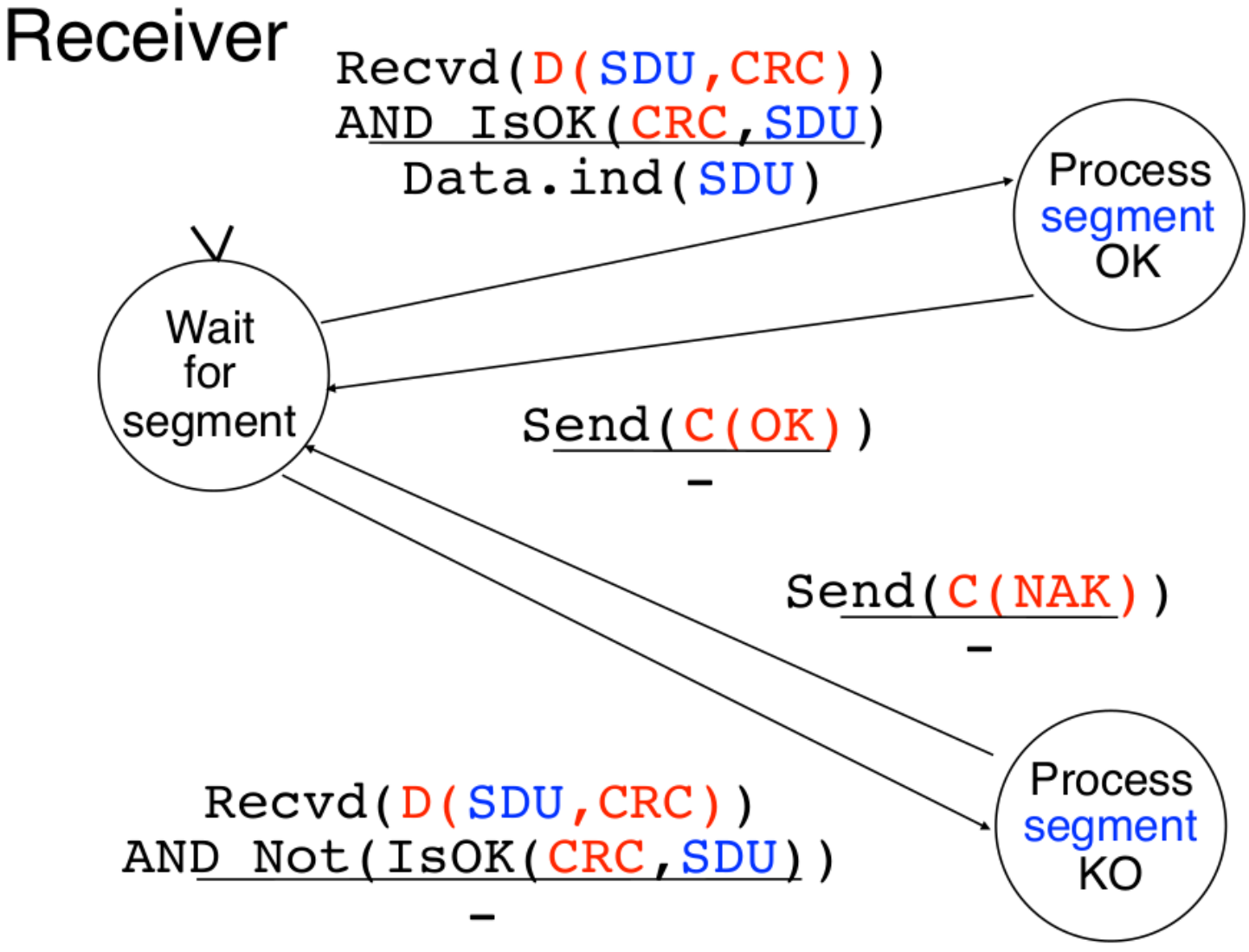
- *sender* $\rightarrow$ Si réception d'un C(NAK) il retransmet le segment.

# Protocole : C(OK) et C(NAK)

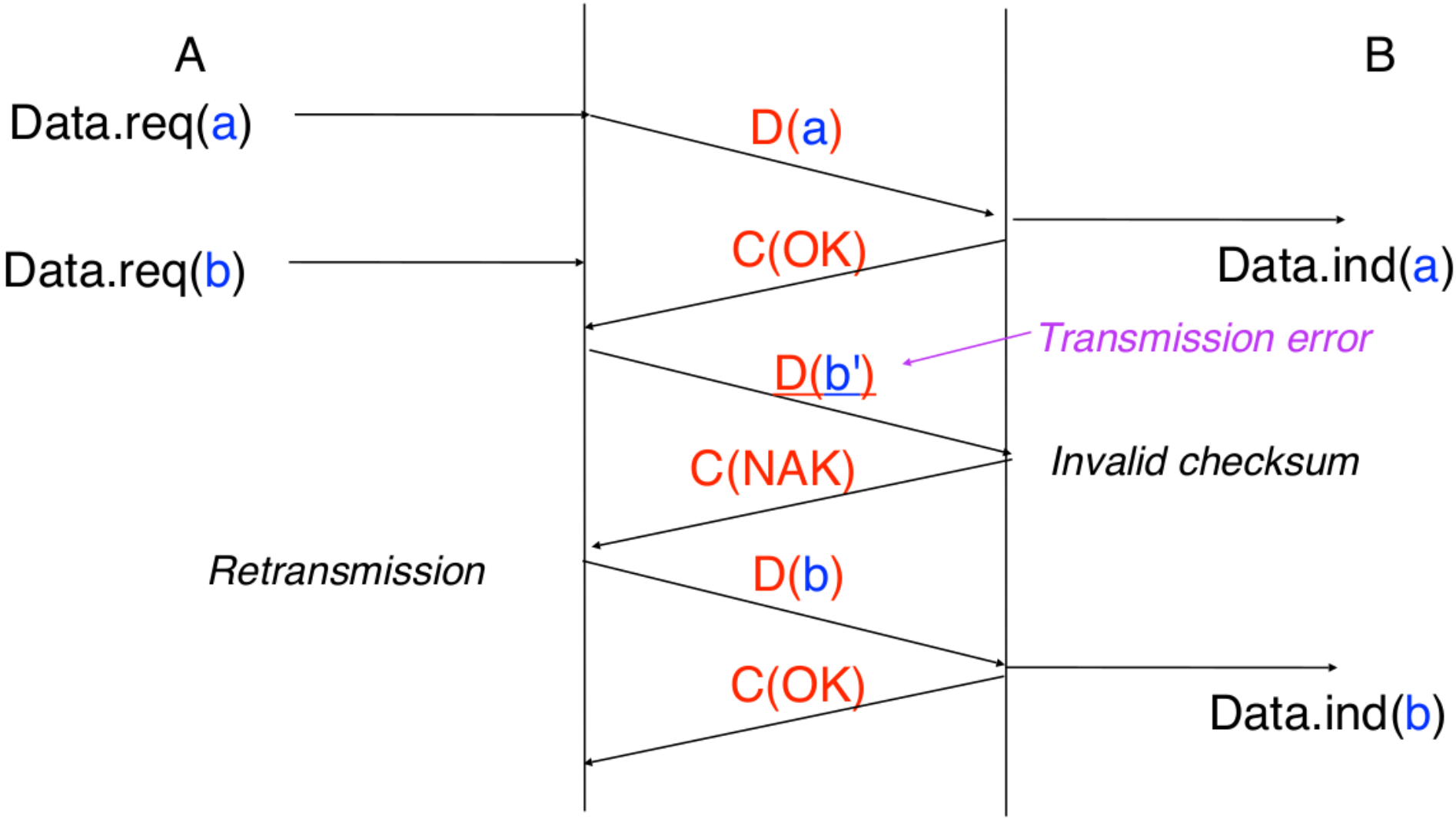
## Sender



# Protocole : C(OK) et C(NAK)



# Protocole : C(OK) et C(NAK)

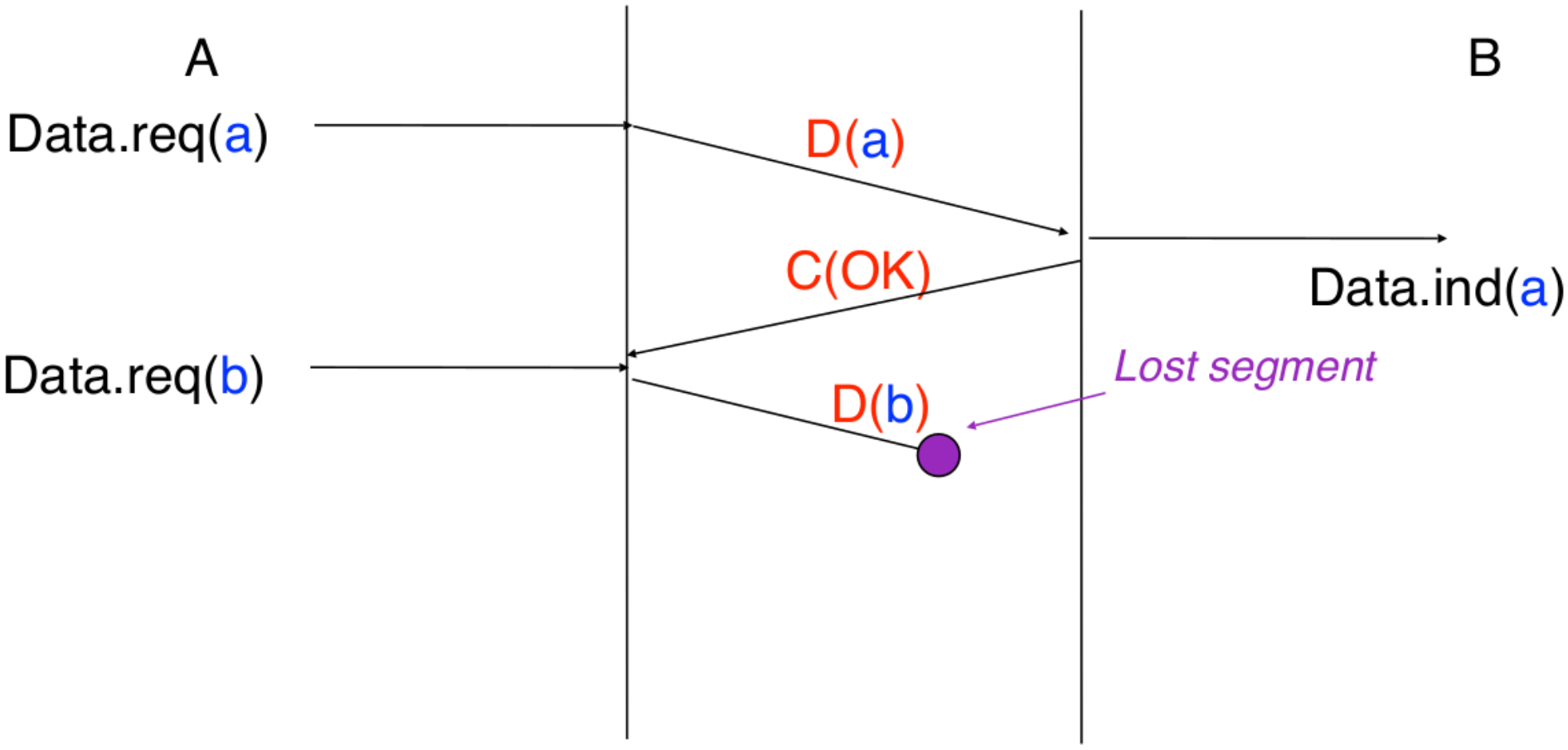


# Protocole : pour un service fiable

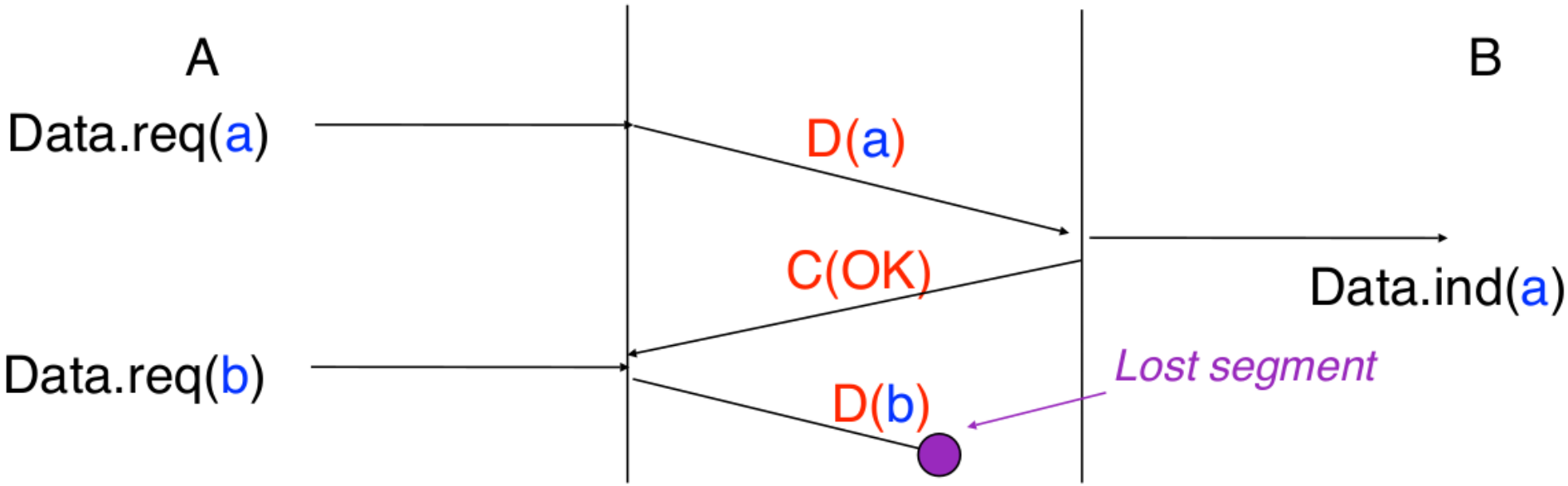
Hypothèses

- Les SDU sont petits
- La qualité de service de la couche réseau
  1. les erreurs de transmission sont possibles
  2. les pertes de paquet sont possibles
  3. il n'y a pas de duplication
- la transmission est unidirectionnelle

# Protocole : C(OK) et C(NAK)



# Protocole : C(OK) et C(NAK)

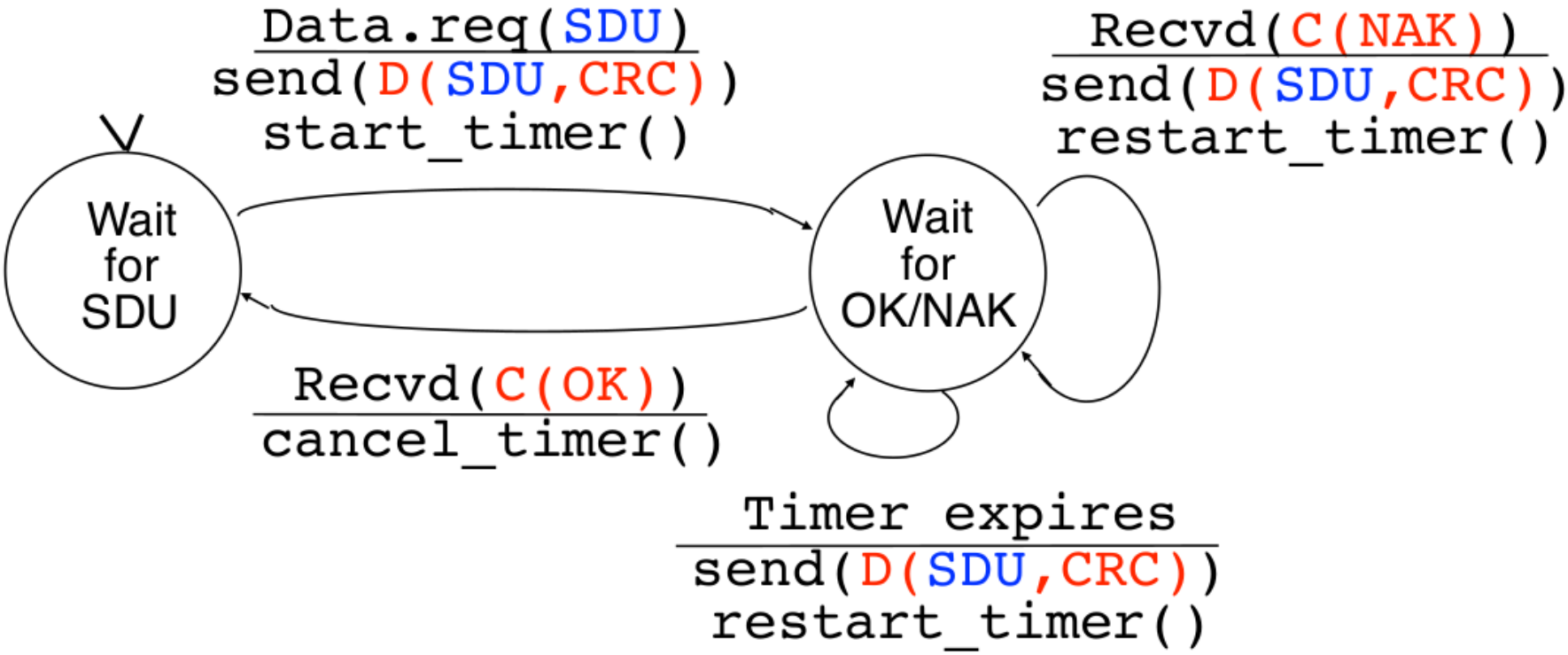



DEADLOCK

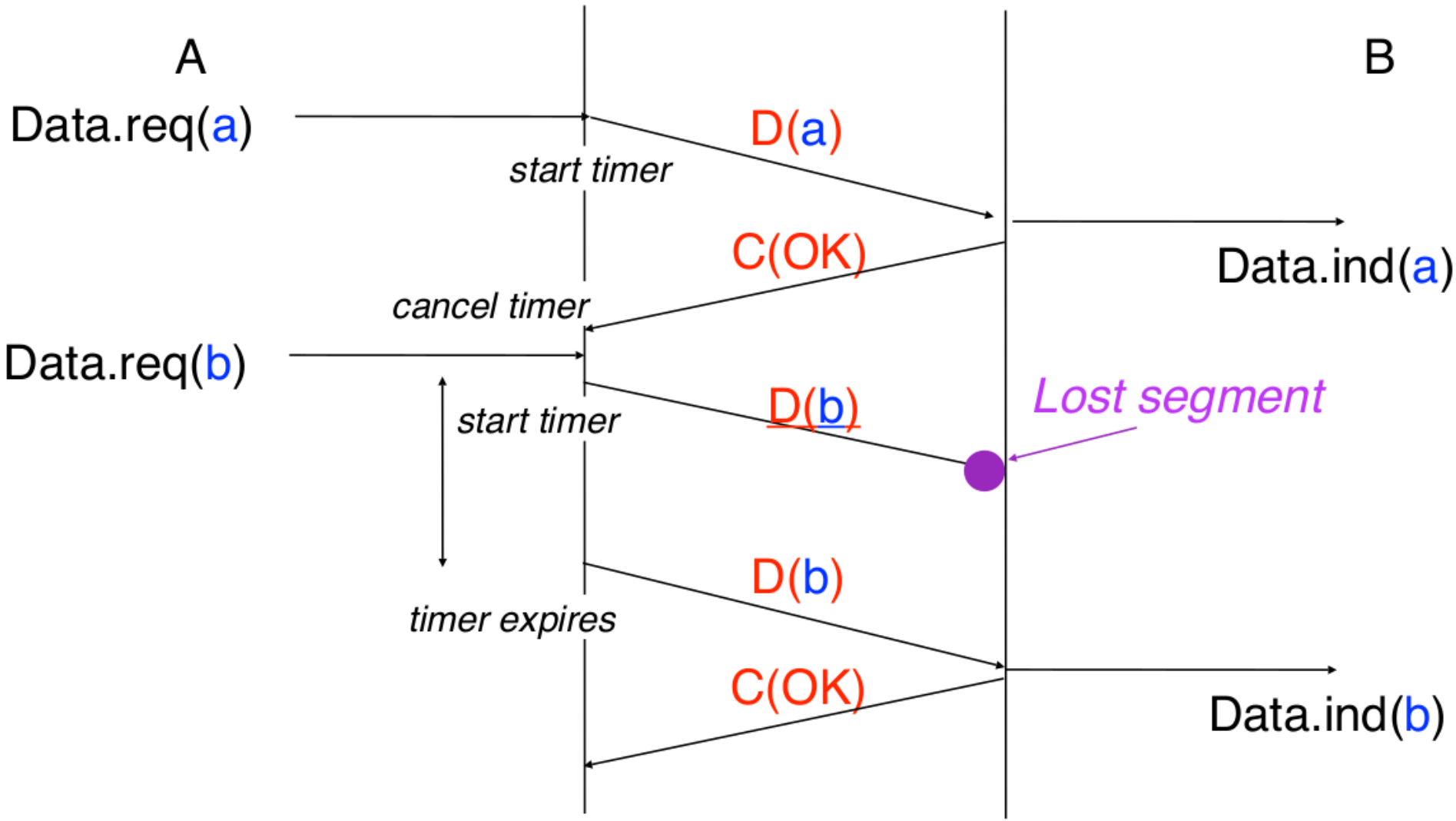
- A est en attente d'un segment de contrôle
- B est en attente de segment de données

# Protocole : avec timer

Modification du *sender* uniquement

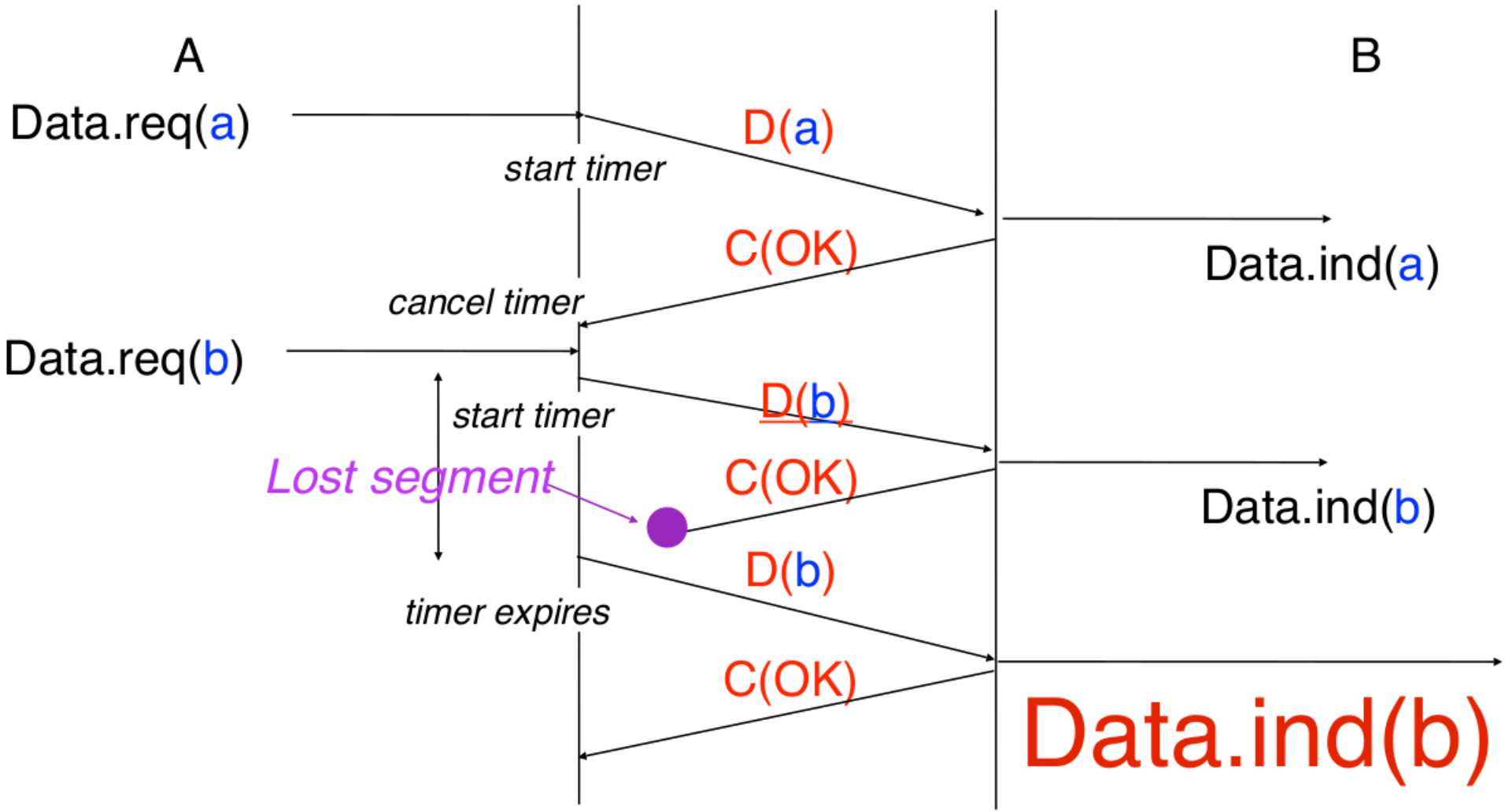


# Protocole : avec timer



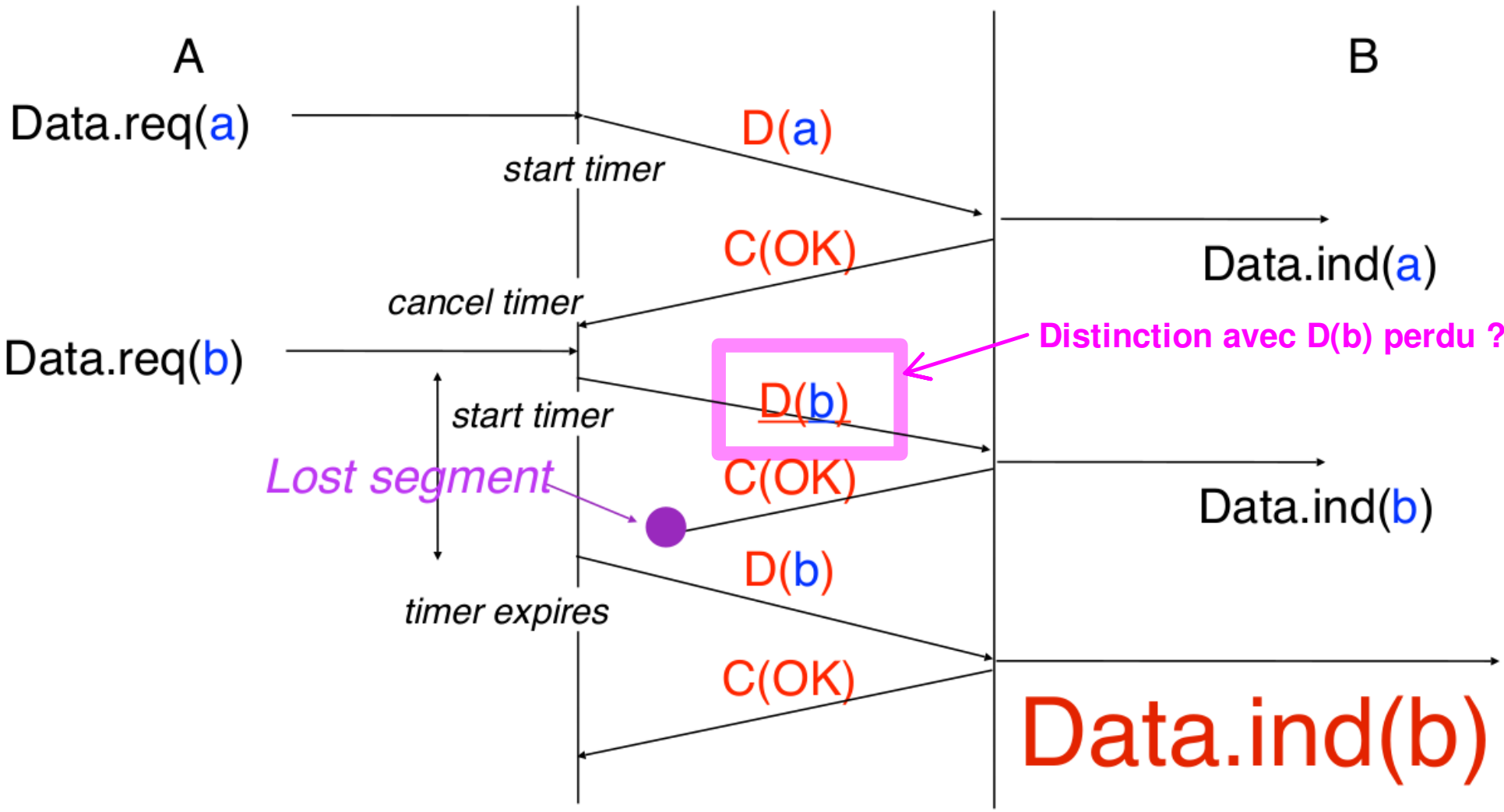
# Protocole : avec timer

**MAIS**



# Protocole : avec timer

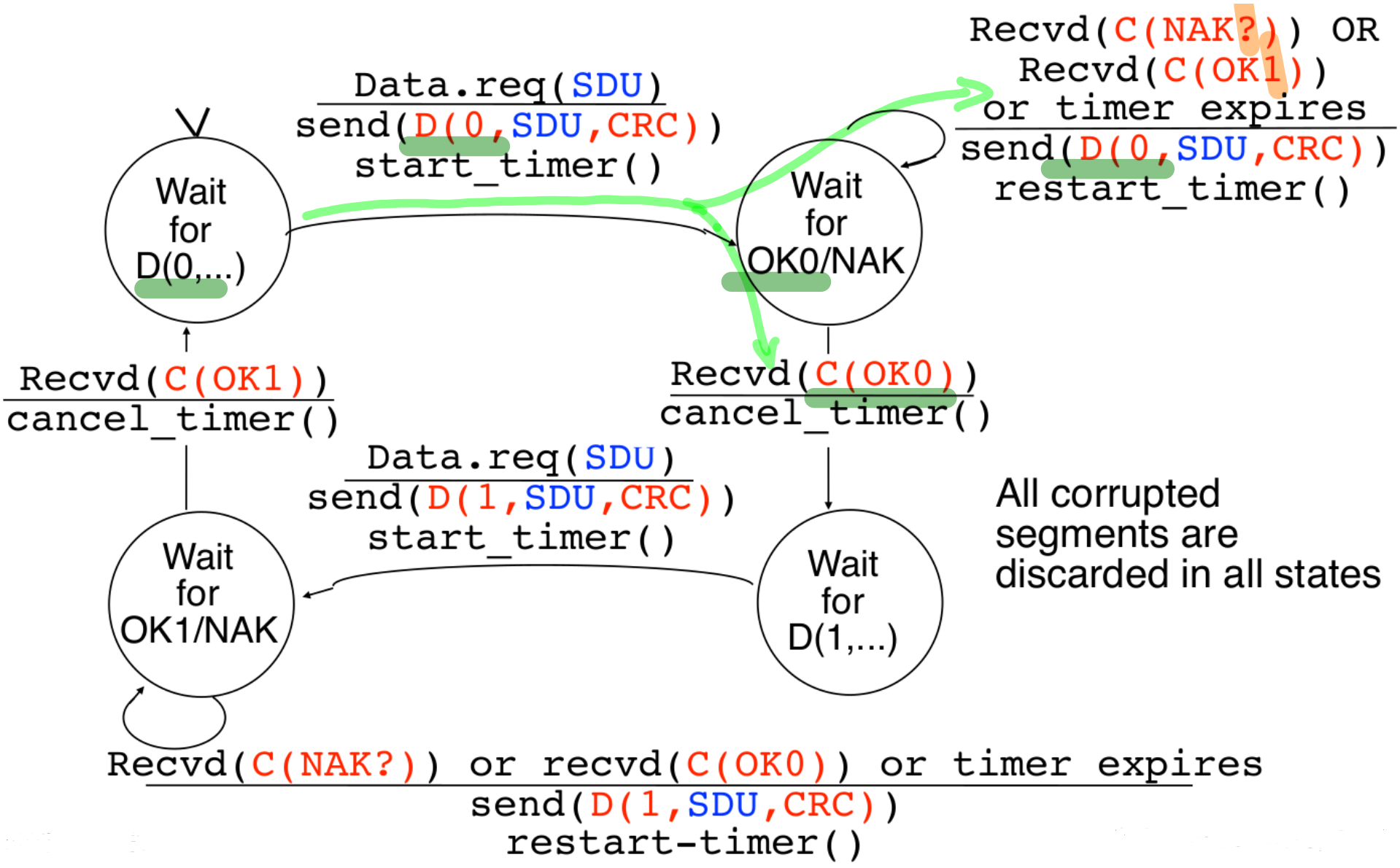
**MAIS**



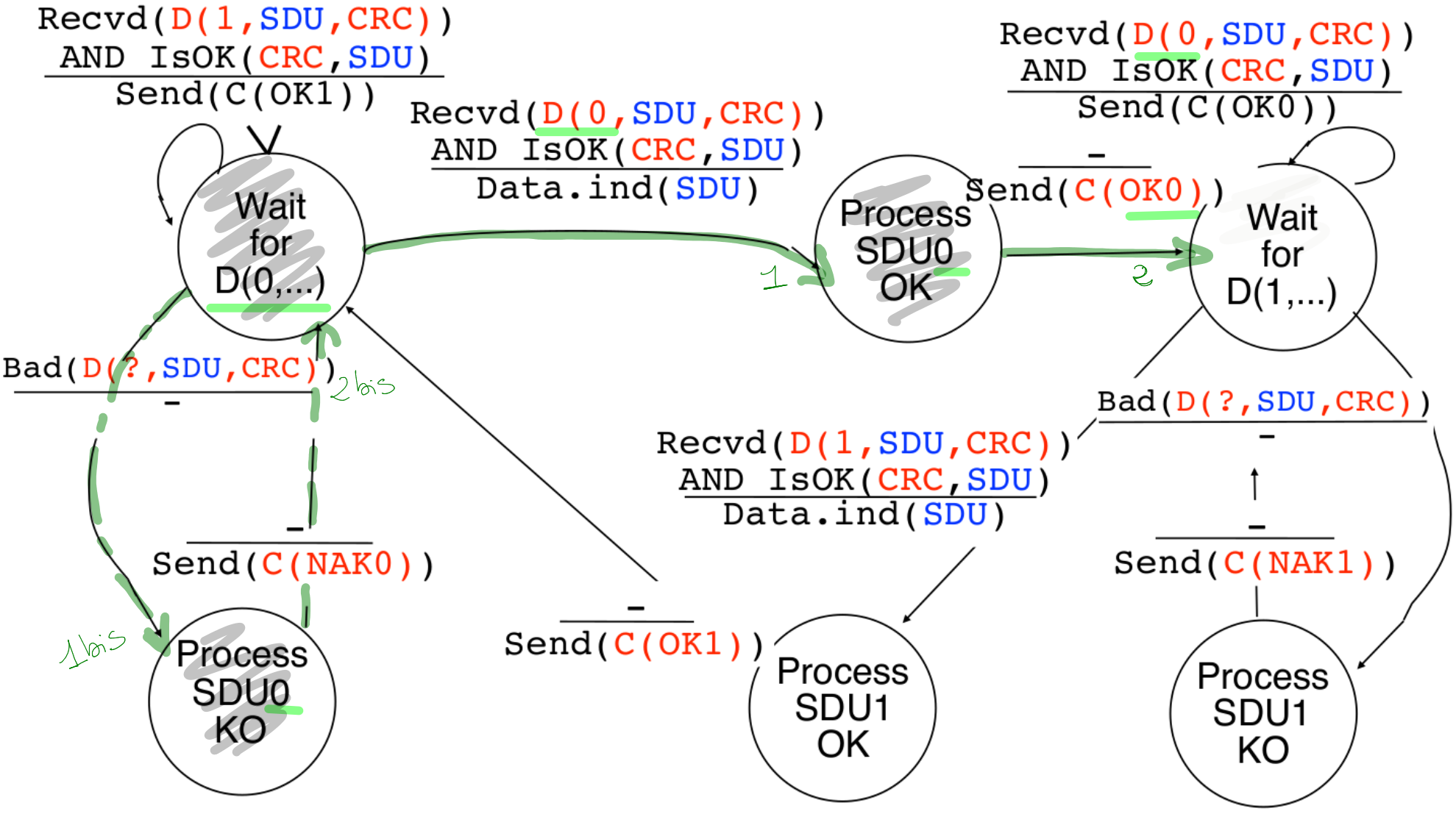
# Protocole : Vers le bit alterné

- Le principe : numéroter les segments
- Le *receiver* vérifie le numéro attendu
  - C(OK) si le numéro est correct
  - C(NAK) sinon
- Un bit est suffisant : le bit alterné

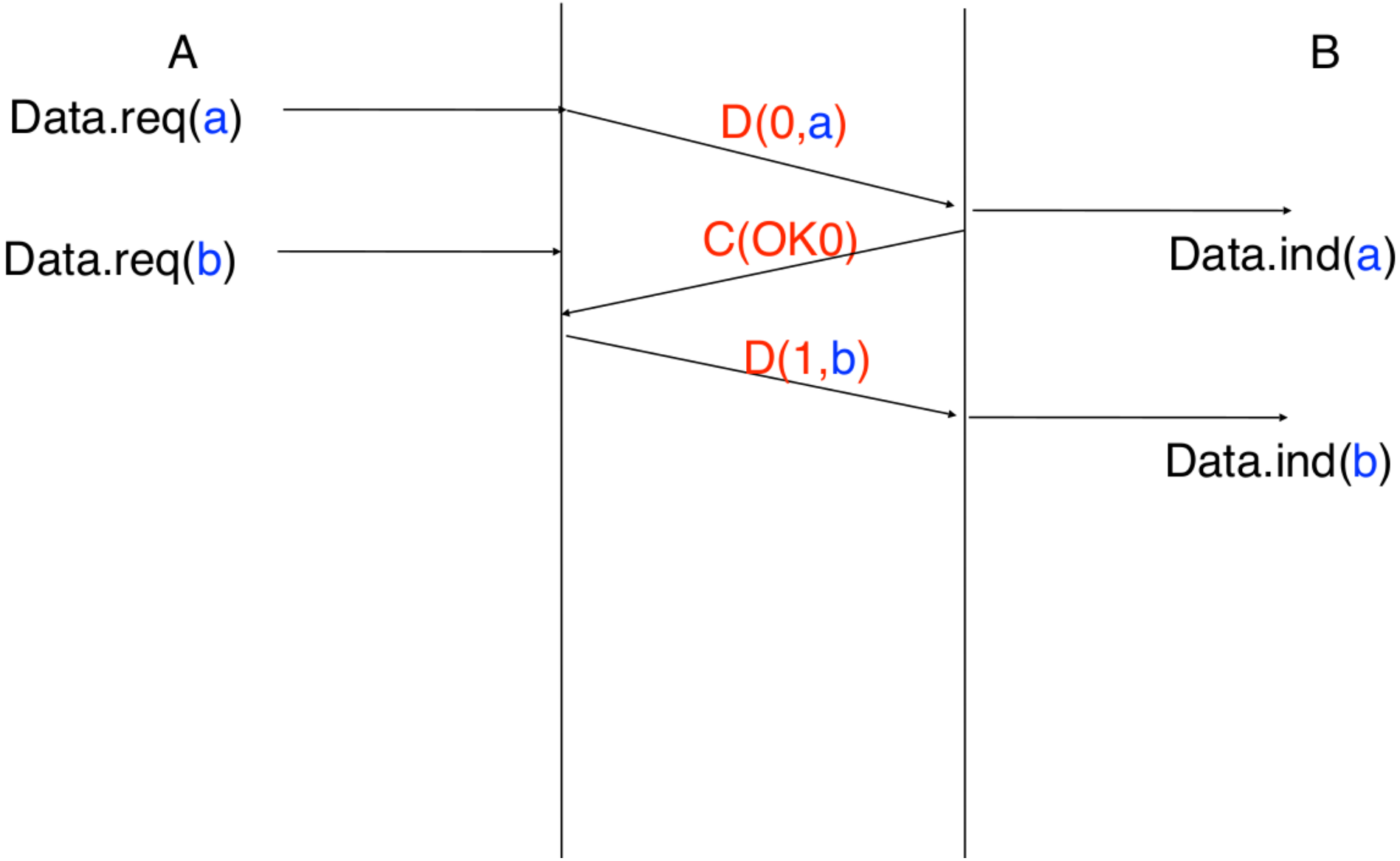
# Protocole du bit alterné : *sender*



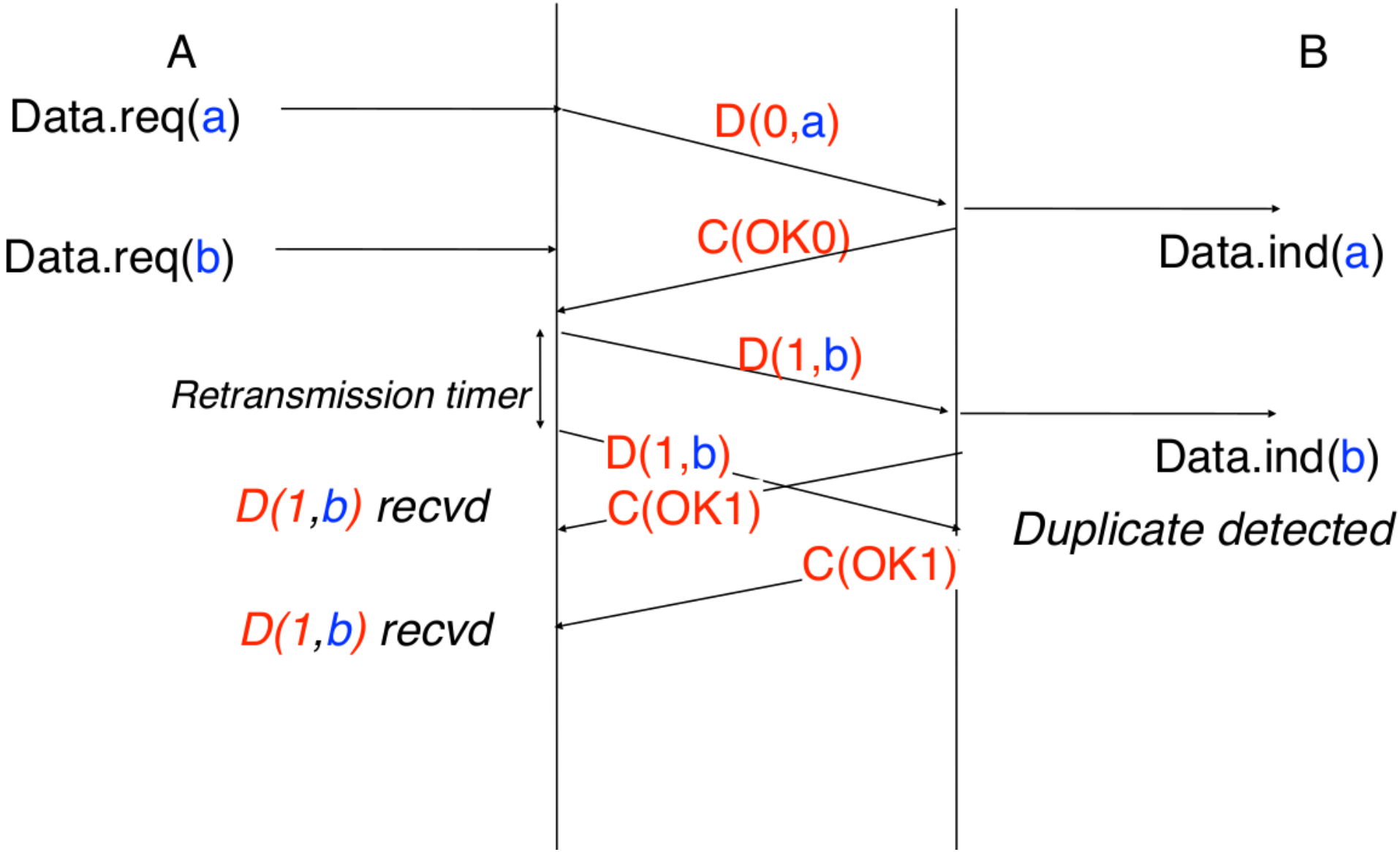
# Protocole du bit alterné : *receiver*



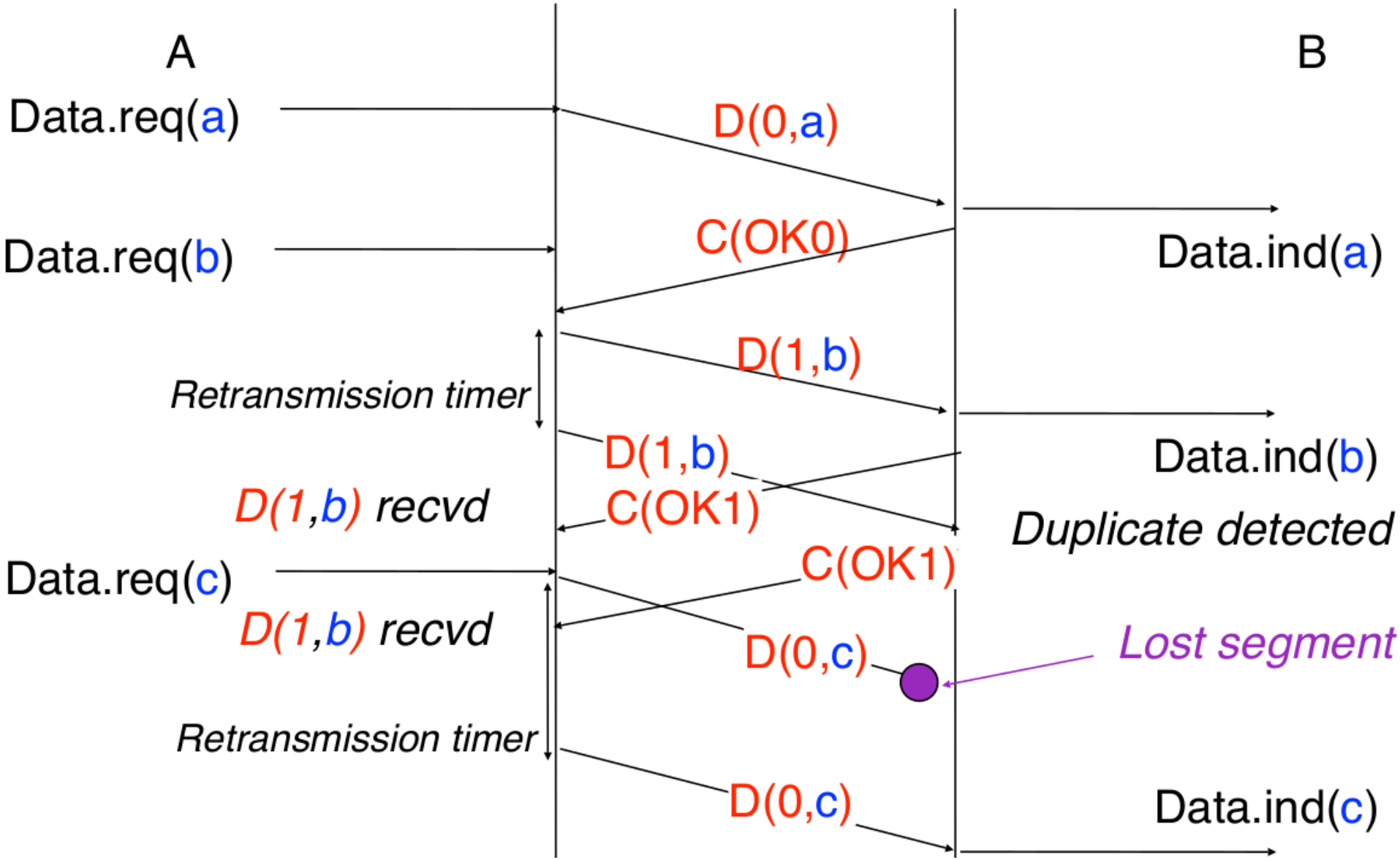
# Protocole du bit alterné : un scénario



# Protocole du bit alterné : un scénario

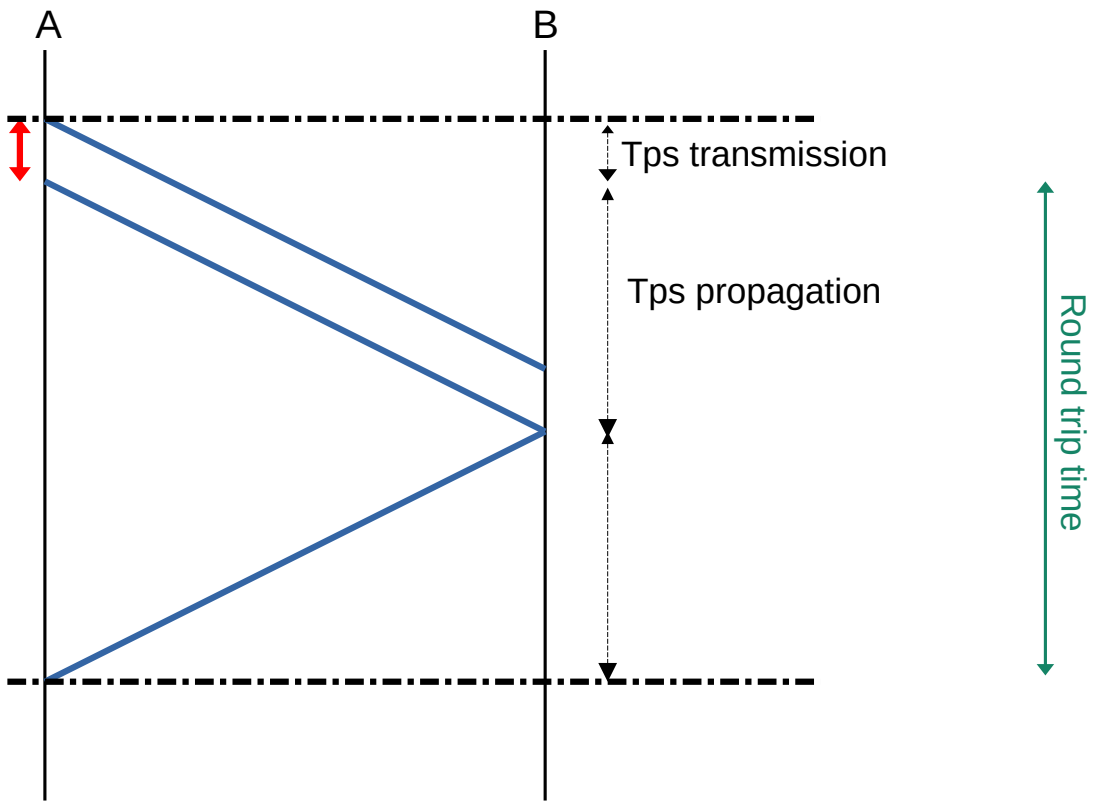


# Protocole du bit alterné : un scénario



# Protocole du bit alterné : la performance

- Le temps de propagation : 250ms
- Le temps de transmission : 50kbps (lié à la couche physique)
- La taille d'un segment : 1000 bits
- $\frac{1000}{50000} = 20ms$



# Protocole du bit alterné : la performance

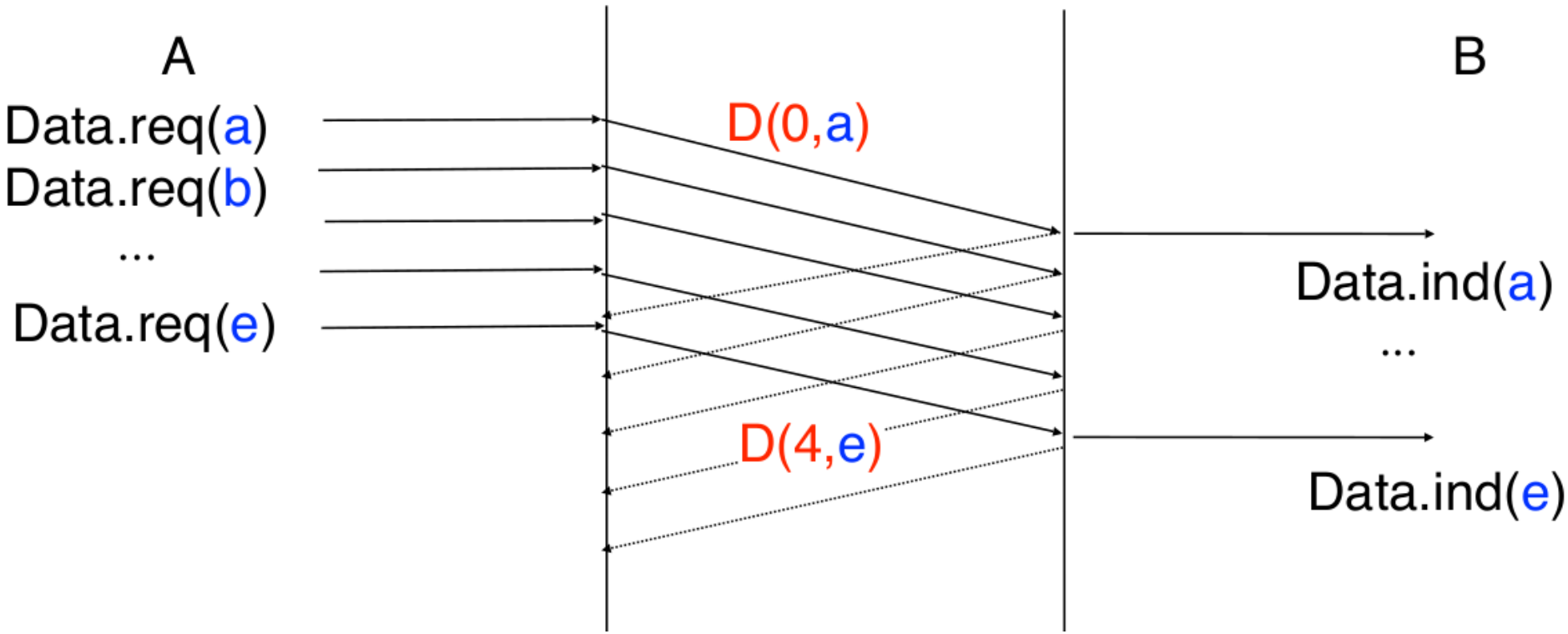
- Le temps de propagation : 250ms
- Le temps de transmission : 50kbps (lié à la couche physique)
- La taille d'un segment : 1000 bits
- $\frac{1000}{50000} = 20ms$
- La taille d'un segment de contrôle est négligeable

La performance est dépendante du Round-Trip-Time et de la bande passante

# Comment améliorer le protocole bit alterné

- Autoriser le *sender* à transmettre des données sans attendre chaque acquittement
- Utiliser des fenêtres d'émission et/ou de réception

-> Améliore l'efficacité et limite les attentes



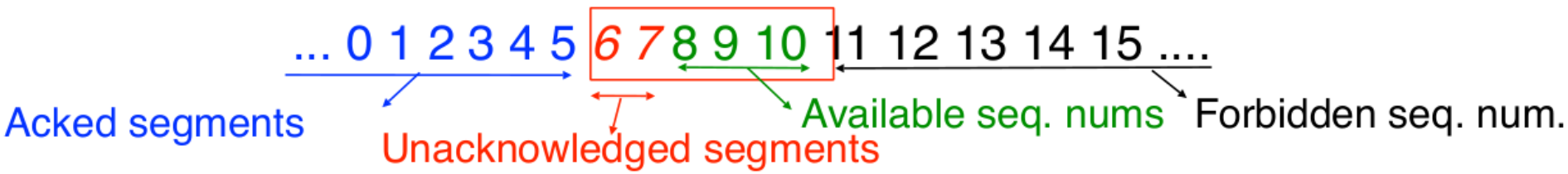
# Les modifications sur le protocole

- **Numéros de séquence**
  - Chaque segment de données contient son propre numéro de séquence.
  - Chaque segment de contrôle indique le numéro du segment de données concerné (OK/NAK).
- ***sender***
  - Doit stocker les segments non encore acquittés pour une retransmission éventuelle.
- ***receiver***
  - Doit éventuellement stocker les segments reçus hors séquence.

Et les potentiels débordements des buffers d'émission et de réception ?

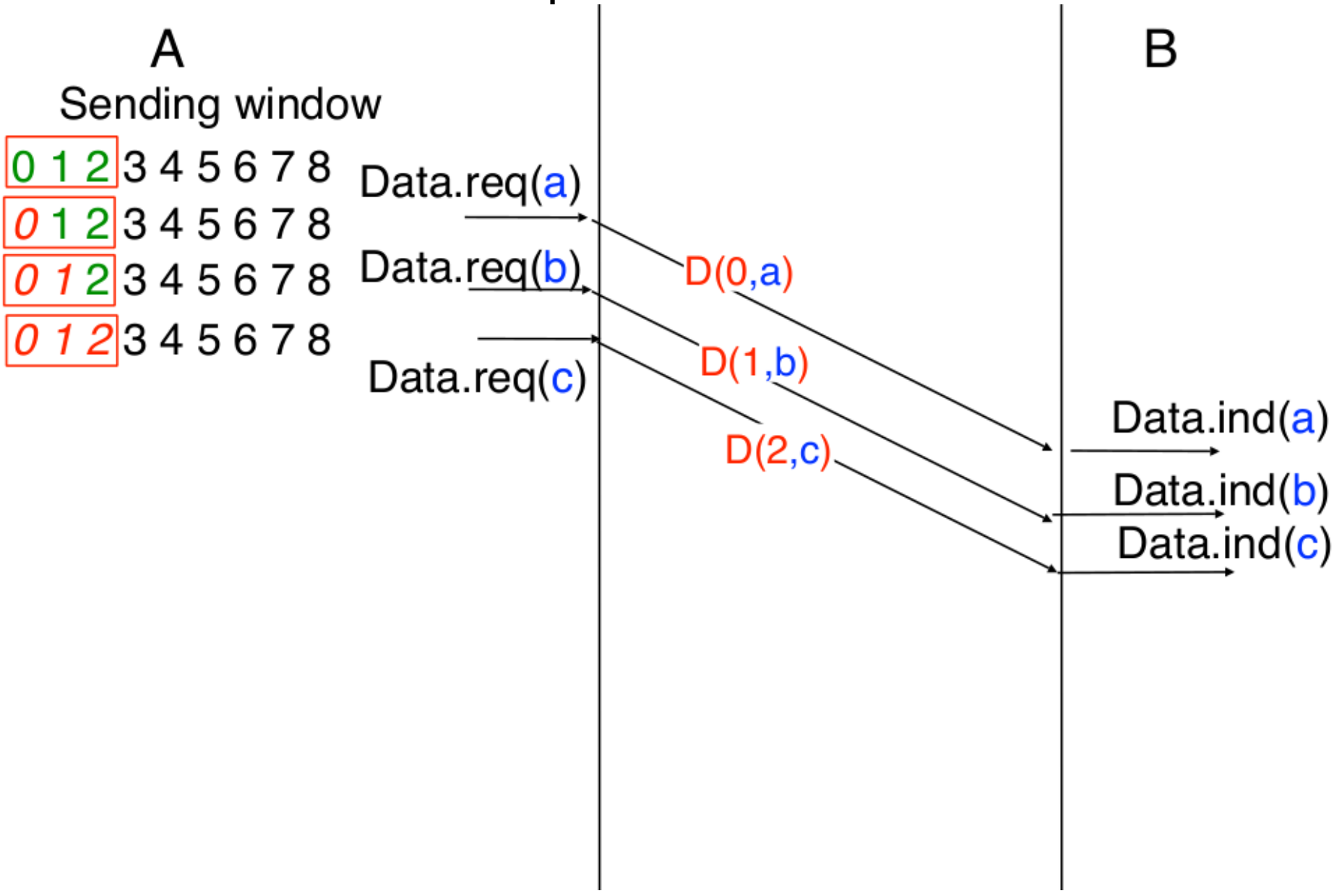
# Les modifications sur le protocole

- Le *sender* doit utiliser une fenêtre glissante avec l'état des segments.

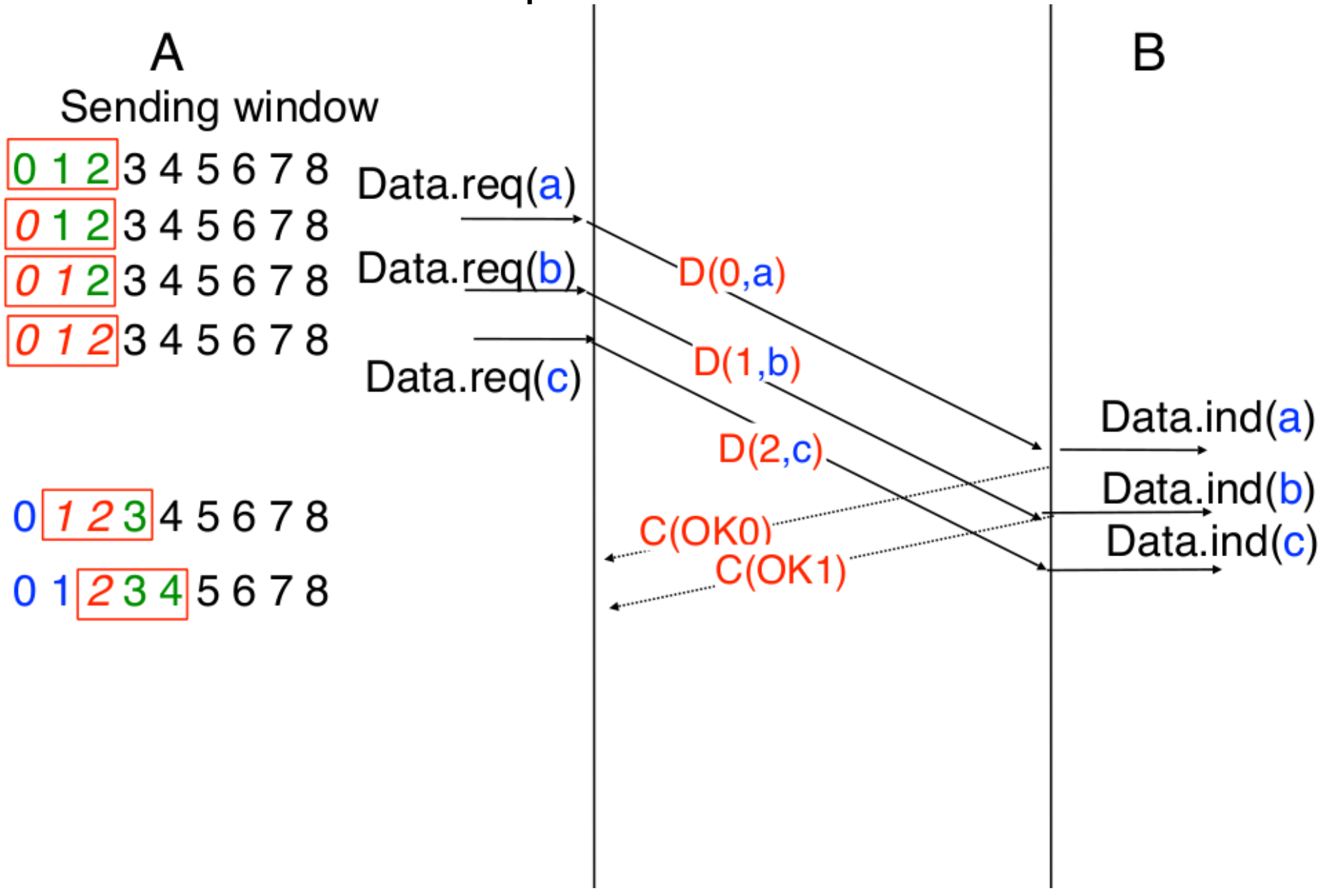



- Le *receiver* doit également maintenir une fenêtre des segments qu'il peut recevoir.
- Les tailles des 2 fenêtres doivent être compatibles (côté *receiver* au moins aussi grande que côté *sender*)

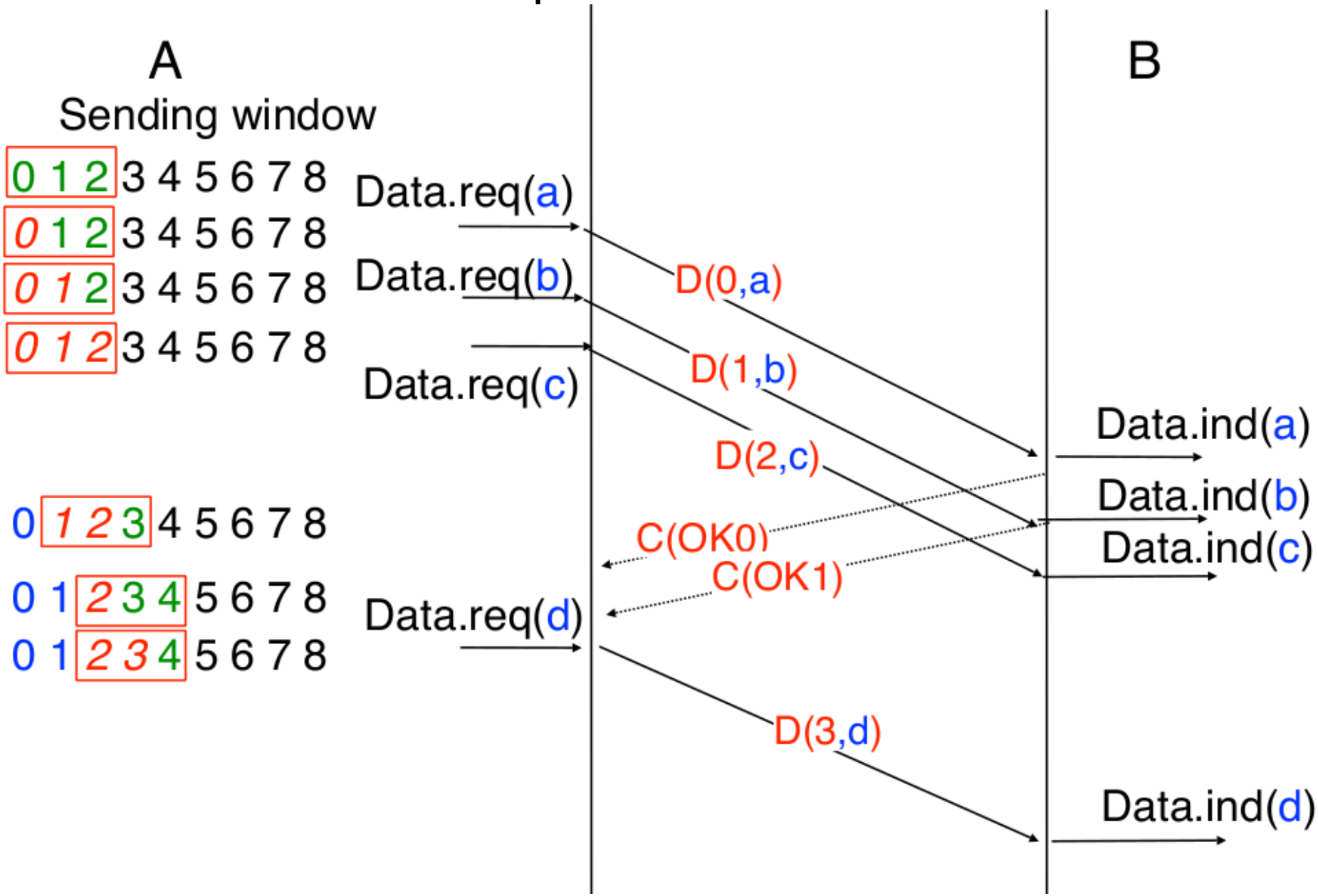
# Un scénario exemple



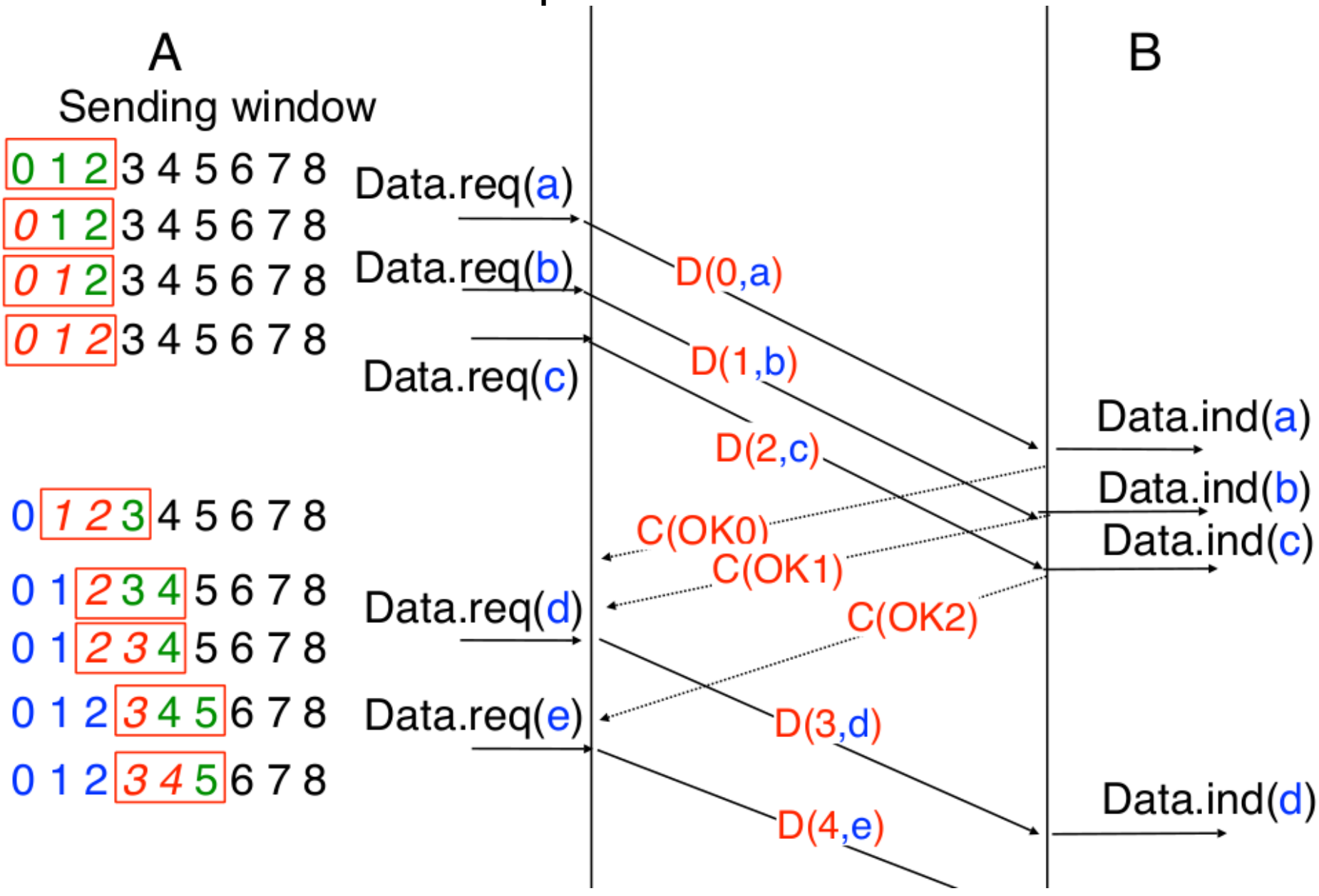
# Un scénario exemple



# Un scénario exemple



# Un scénario exemple

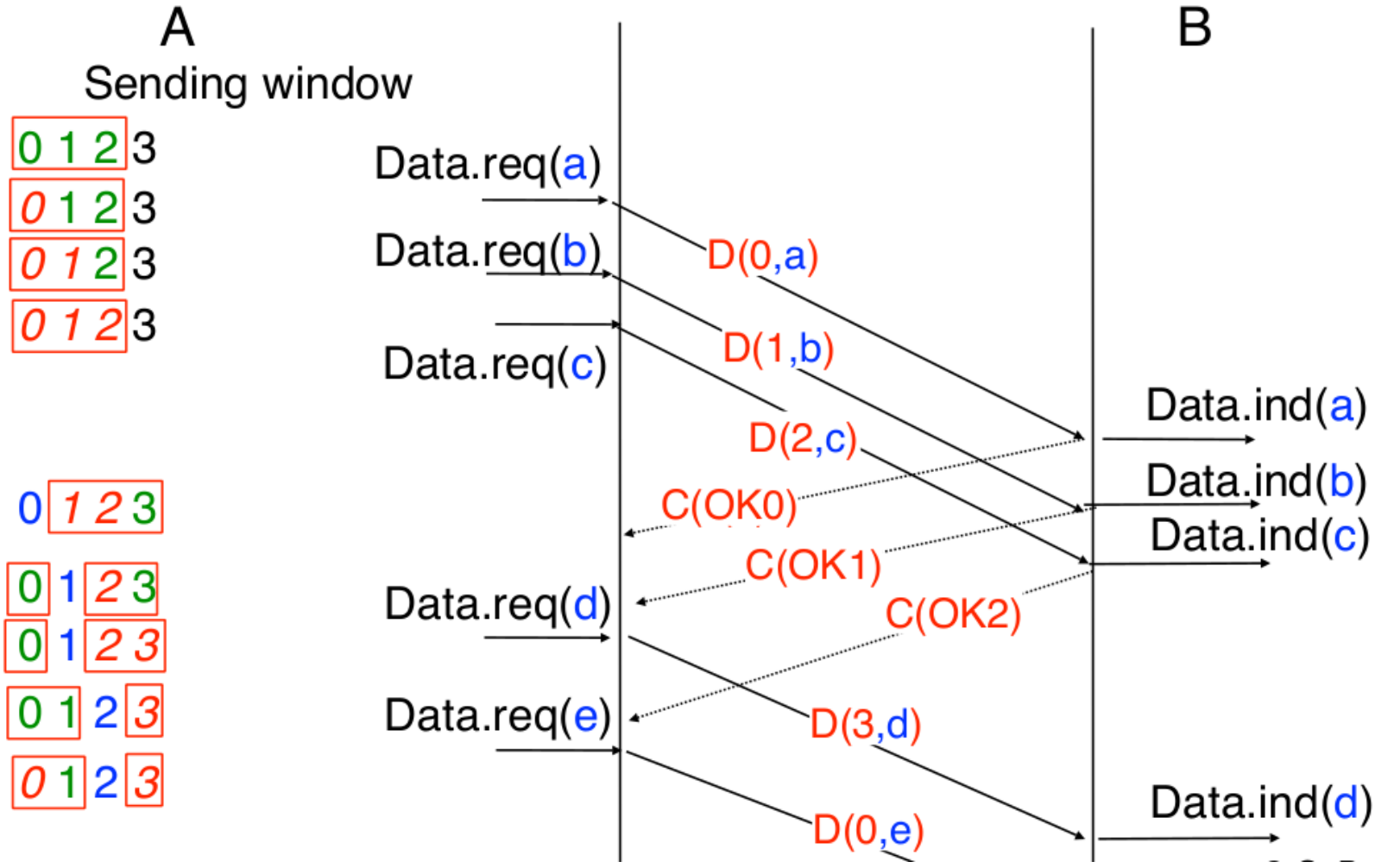


# Attention à la numérotation

- **Taille**
  - Combien de bits dans l'en-tête pour encoder le numéro de séquence ?
  - Avec $N$ bits, on obtient $2^N$ numéros de séquence possibles.
- **Règle**
  - Chaque segment transmis contient son numéro de séquence modulo $2^N$.
  - Un même numéro peut être réutilisé pour plusieurs segments.
  - Attention : cela peut poser des problèmes...
- **Fenêtre glissante**
  - Liste des numéros de séquence consécutifs (modulo $2^N$) que l'émetteur peut envoyer.

# Scénario exemple

Une fenêtre glissante de taille 3 et un encodage sur 2 bits



# Les protocoles à fenêtre(s)

## GoBackN

- Fenêtre d'émission uniquement
- Comment gérer les pertes (arrivée dans le désordre)

## Répétition sélective

- Fenêtre d'émission et Fenêtre de réception
- plus difficile à implémenter

# GoBackN

Côté *receiver*

- Il doit être le plus simple possible
- Il n'accepte que les segments dans le bon ordre
- OK(X) acquitte tous les segments jusqu'à X compris
- NAK(X) le segment numéro X est erroné ou perdu

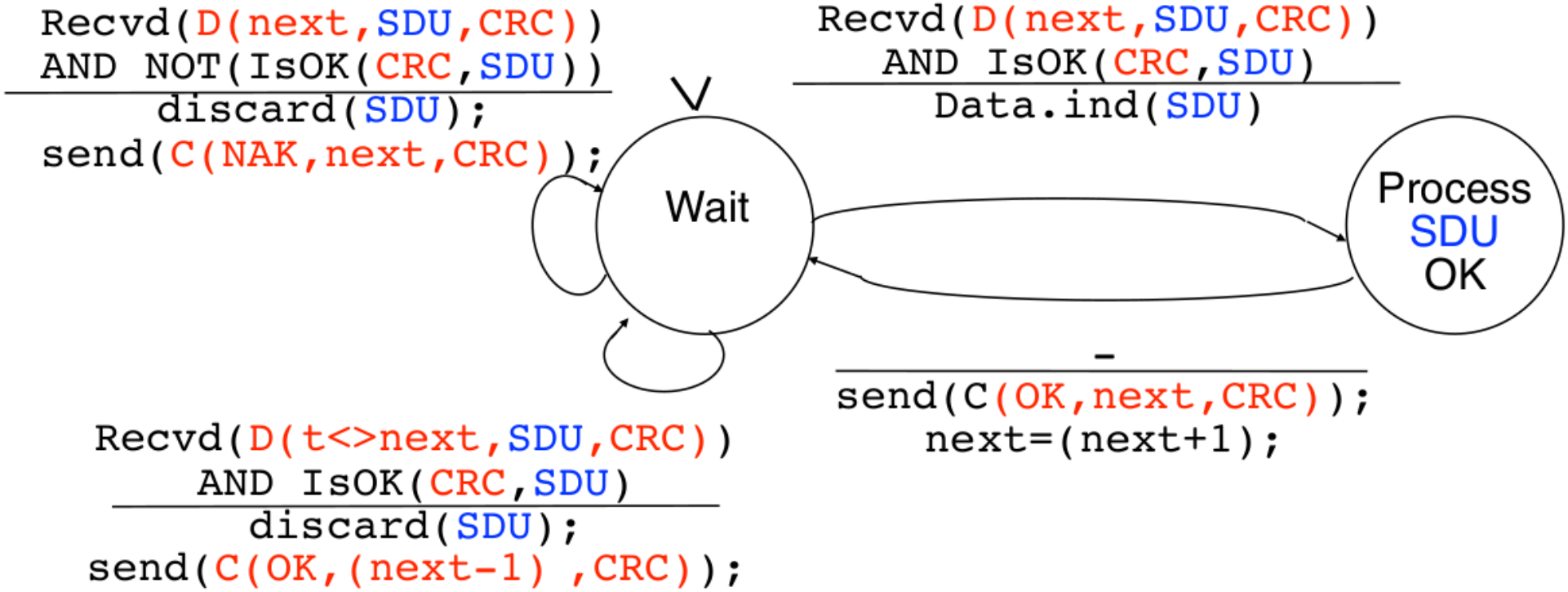
Côté *sender*

- Le timer permet de détecter les pertes
- A l'expiration de ce timer ou à la réception d'un NAK( ?)
  -> tous les segments non acquittés de la fenêtre sont retransmis

# GoBackN

Côté *receiver*

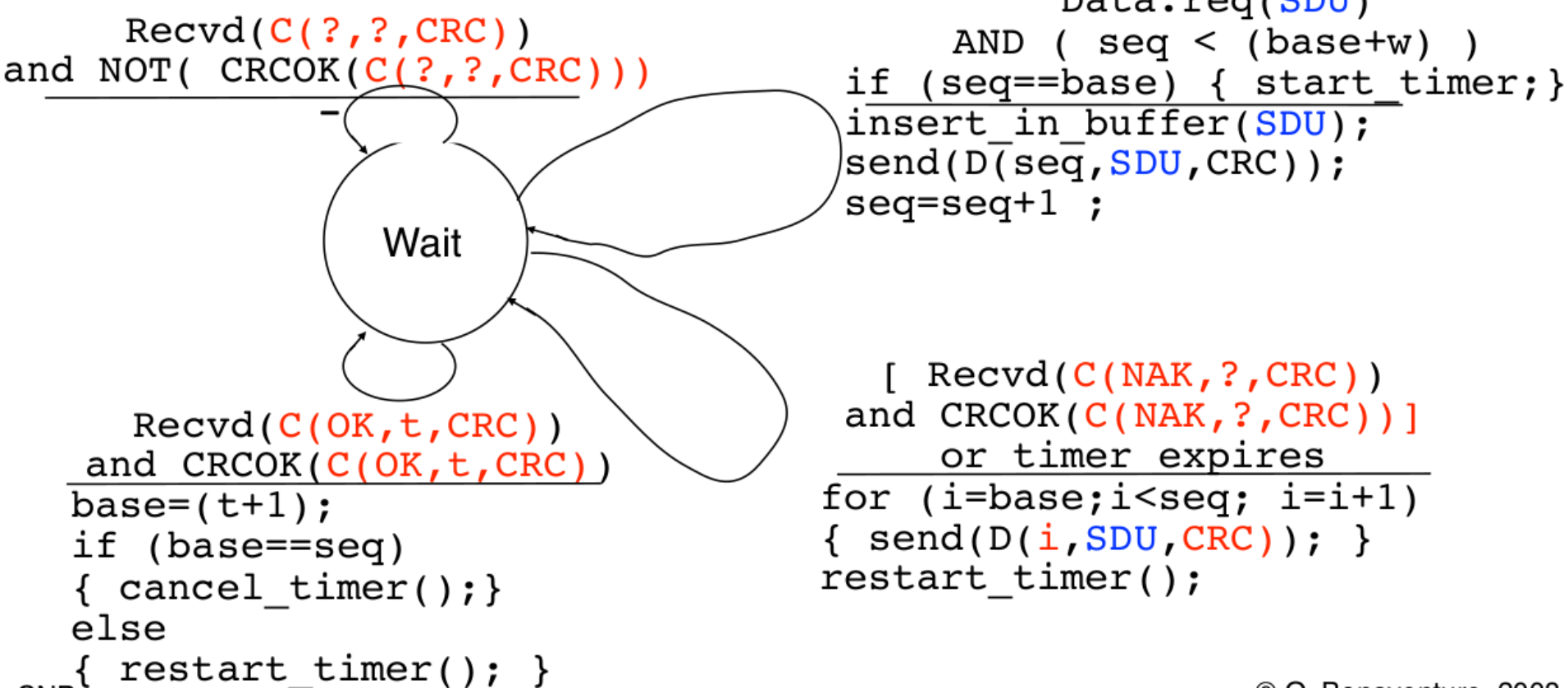
`next` : sequence number of expected data segment



# GoBackN

Côté *sender*

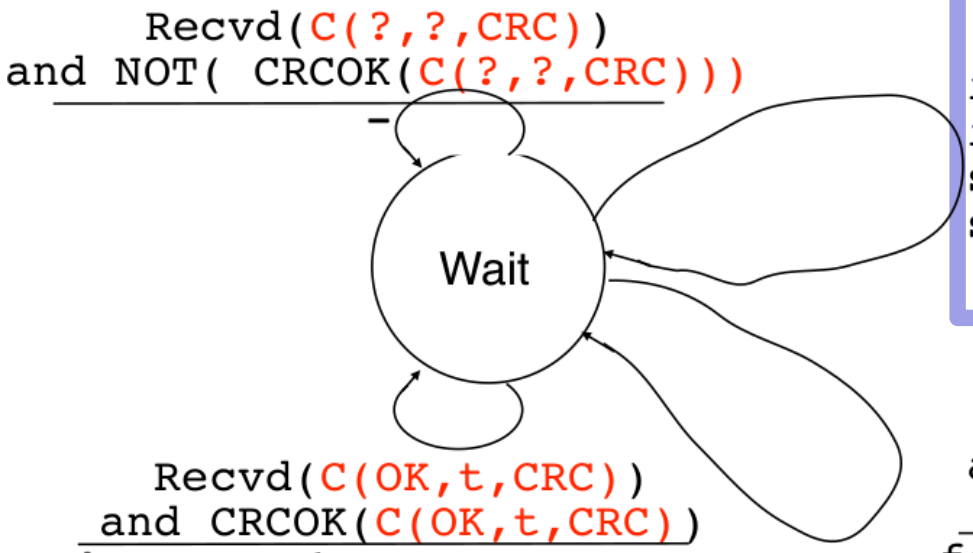
`base` : sequence number of oldest data segment
`seq` : first available sequence number
`W` : size of sending window



# GoBackN

Côté *sender* → envoi de la fenêtre

base : sequence number of oldest data segment
seq : first available sequence number
W : size of sending window



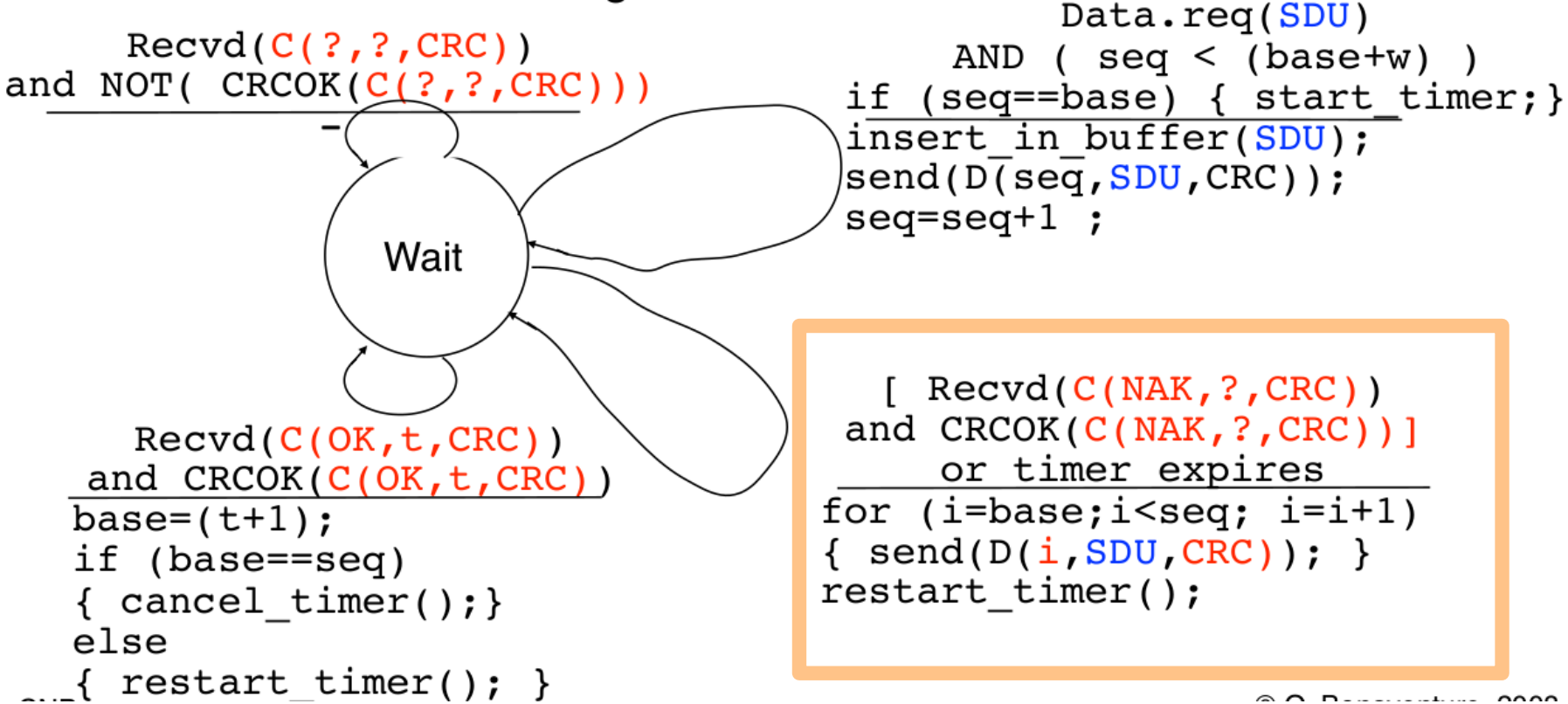

```
Data.req(SDU)
    AND ( seq < (base+w) )
if (seq==base) { start_timer;}
insert_in_buffer(SDU);
send(D(seq,SDU,CRC));
seq=seq+1 ;
```

```
[ Recvd(C(NAK,?,CRC))
and CRCOK(C(NAK,?,CRC))]
    or timer expires
for (i=base;i<seq; i=i+1)
{ send(D(i,SDU,CRC)); }
restart_timer();
```

# GoBackN

Côté *sender* → si problème

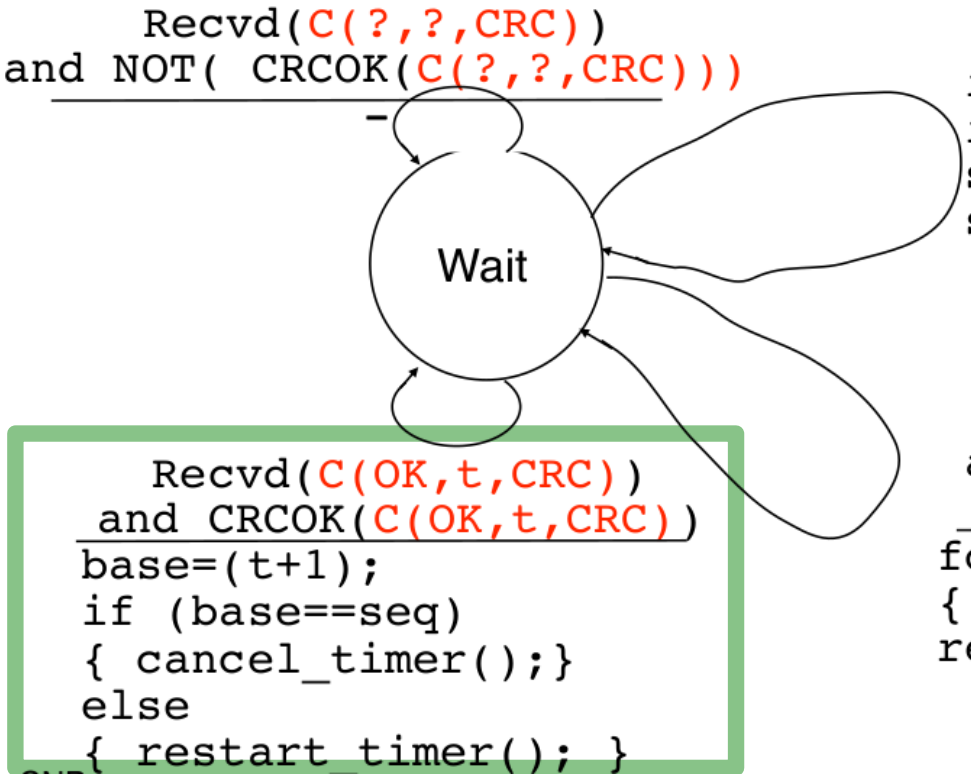
`base` : sequence number of oldest data segment
`seq` : first available sequence number
`W` : size of sending window



# GoBackN

Côté *sender* → si contrôle ok

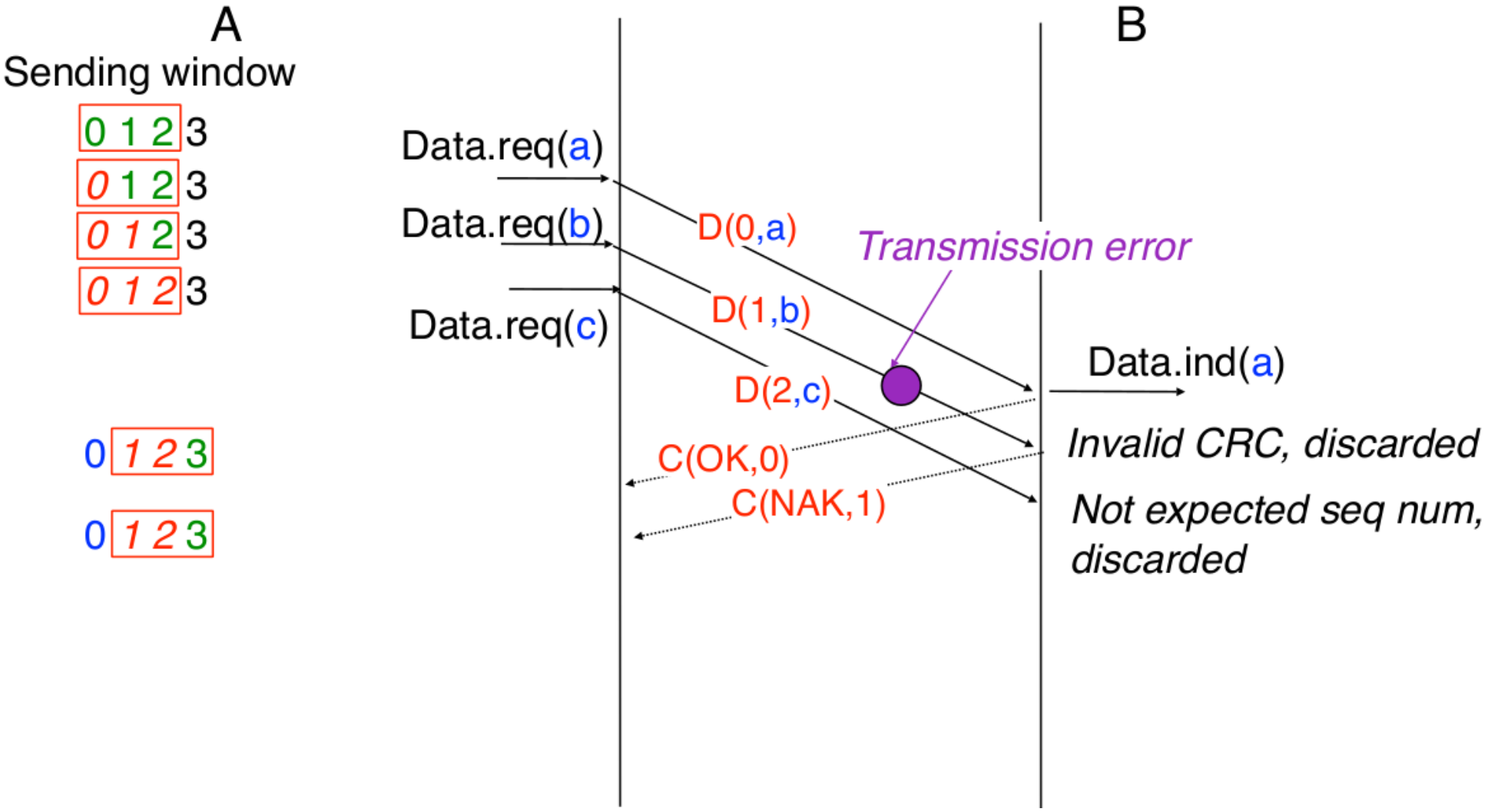
`base` : sequence number of oldest data segment
`seq` : first available sequence number
`W` : size of sending window



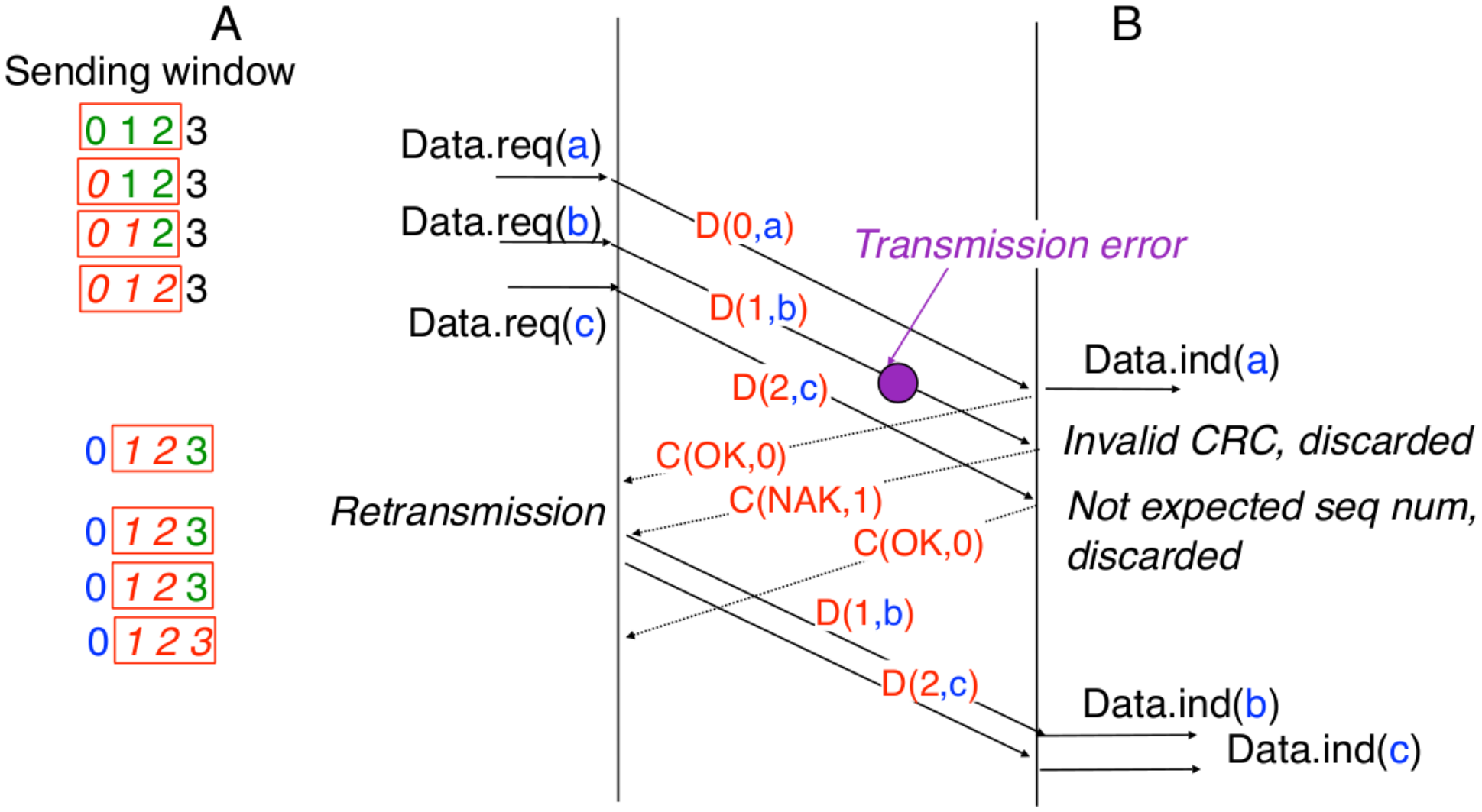

```
Data.req(SDU)
    AND ( seq < (base+w) )
if (seq==base) { start_timer;}
insert_in_buffer(SDU);
send(D(seq,SDU,CRC));
seq=seq+1 ;
```

```
[ Recvd(C(NAK,?,CRC))
and CRCOK(C(NAK,?,CRC))]
    or timer expires
for (i=base;i<seq; i=i+1)
{ send(D(i,SDU,CRC)); }
restart_timer();
```

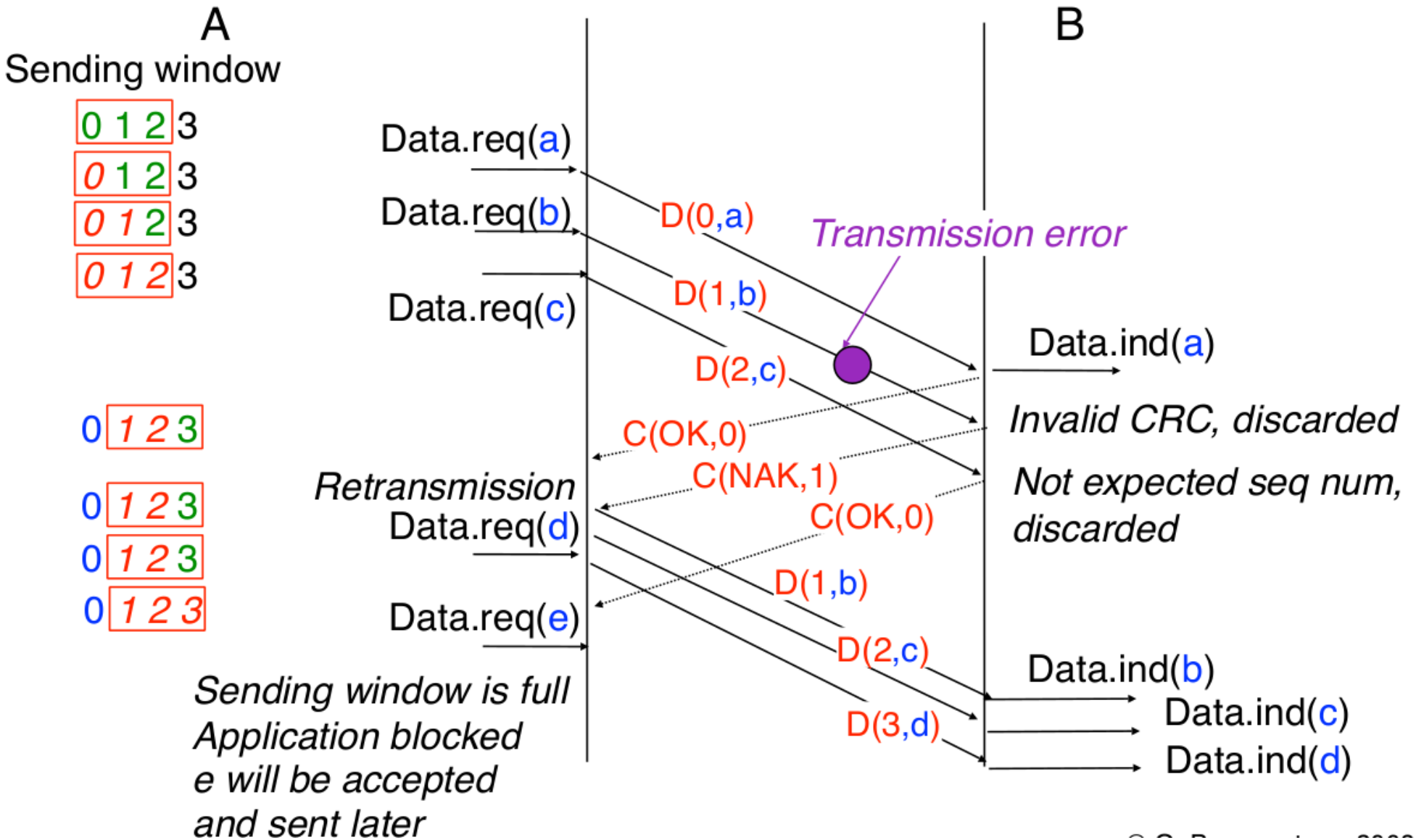
# GoBackN : exemple de scénario 1



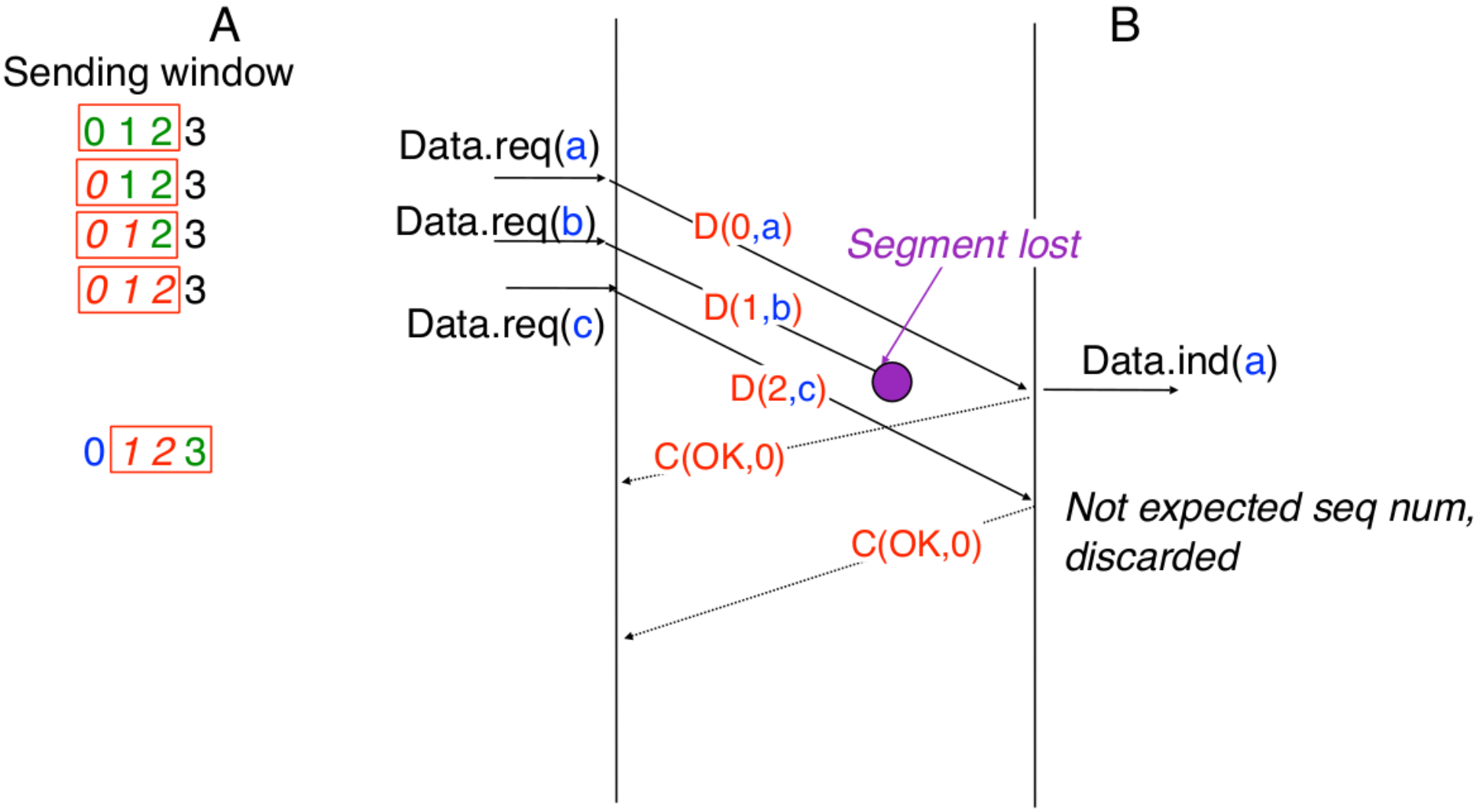
# GoBackN : exemple de scénario 1



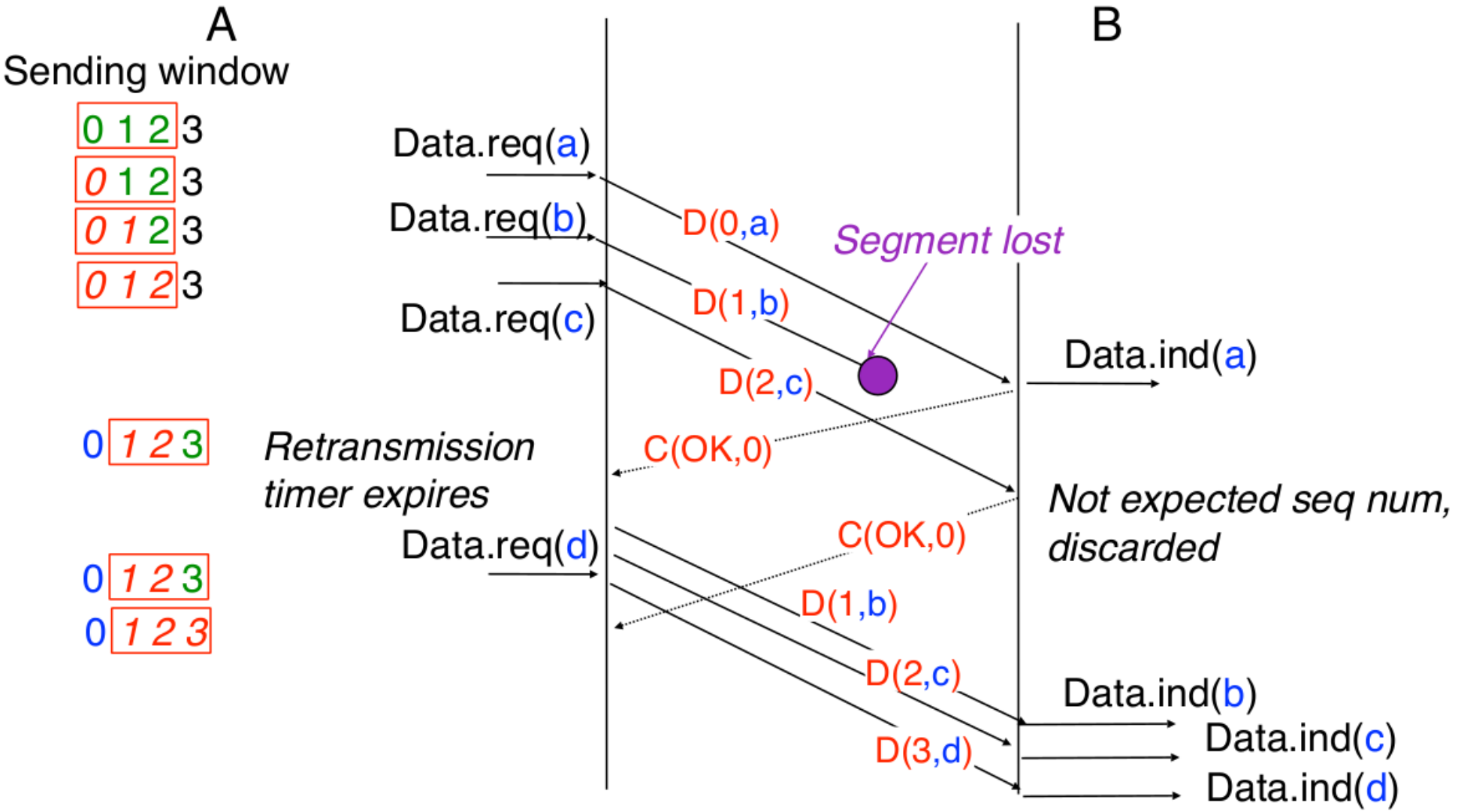
# GoBackN : exemple de scénario 1



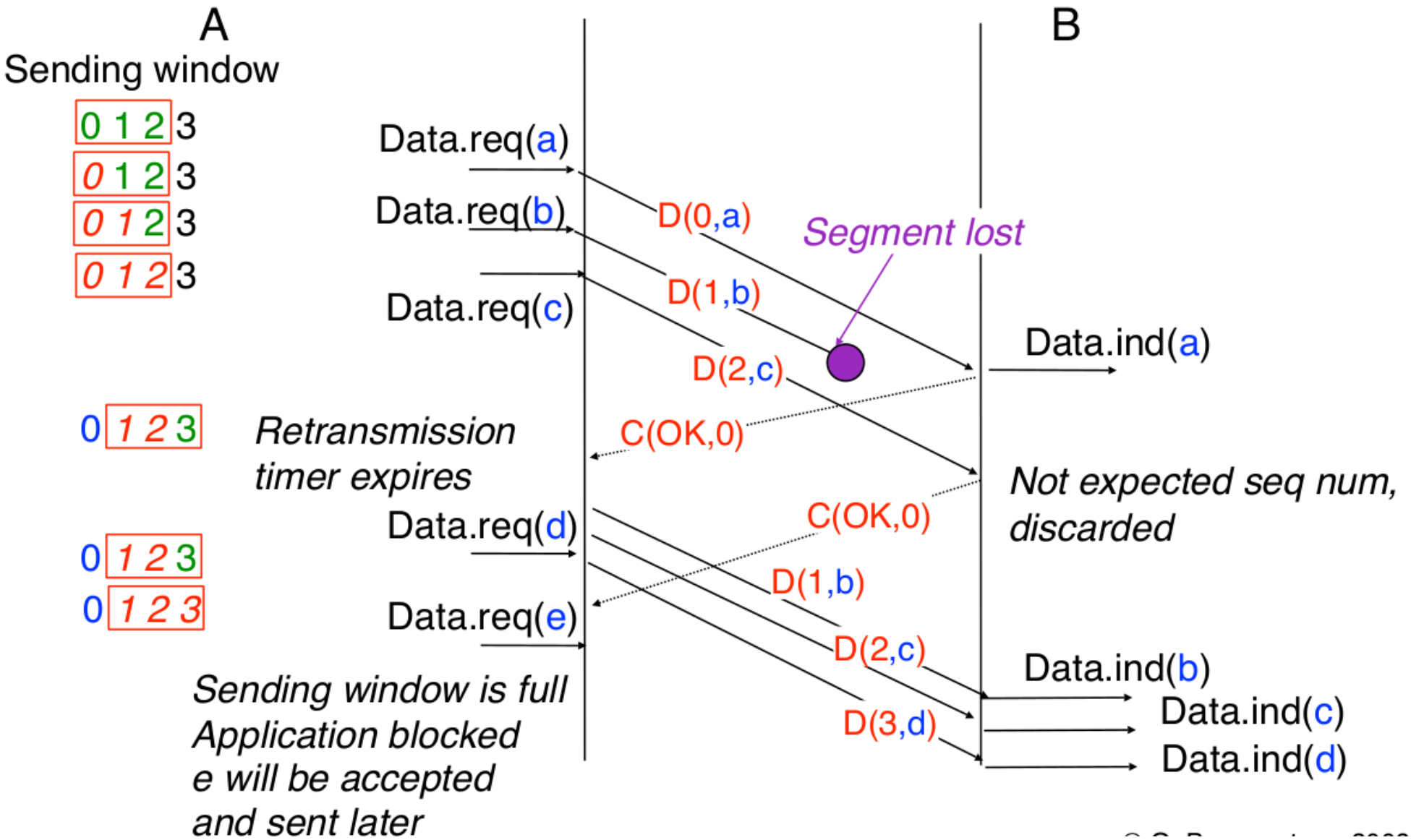
# GoBackN : exemple de scénario 2



# GoBackN : exemple de scénario 2



# GoBackN : exemple de scénario 2



# GoBackN : Taille de la fenêtre

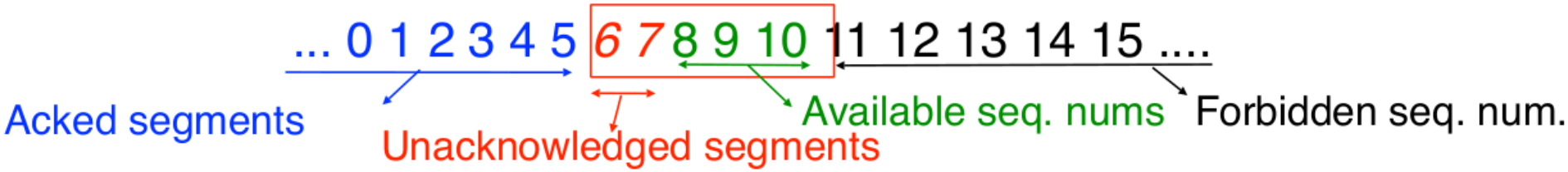
$$W \leq 2^k - 1$$

- $W$ taille de la fenêtre
- $k$ nombre de bits pour la numérotation des segments

* Pour $k = 3$ la numérotation est $0, 1, \ldots, 7$ il faut absolument que 0 soit validé avant de transmettre 7.

# Répétition sélective

une fenêtre également à la réception

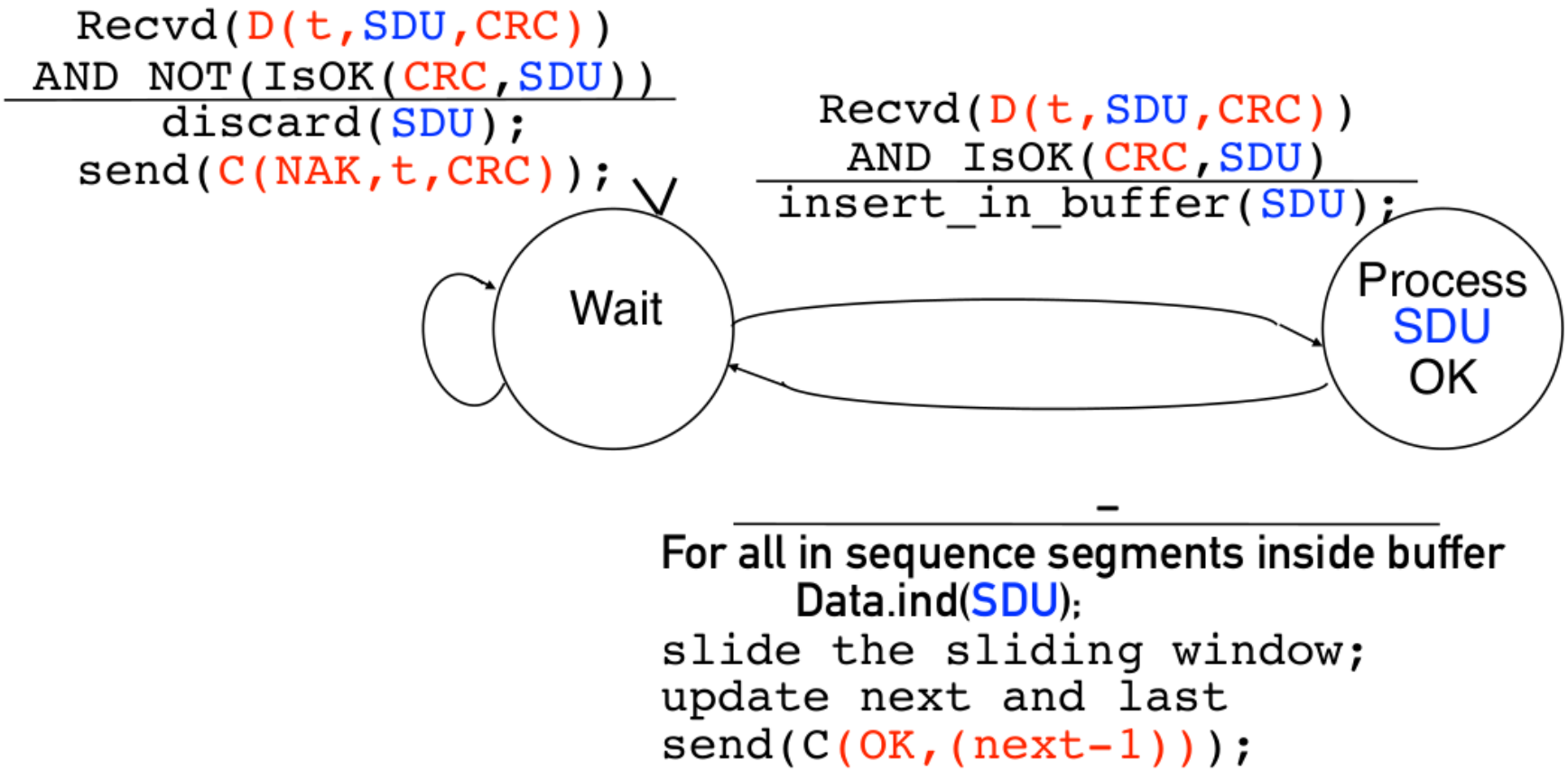



- OK(X) acquitte tous les segments jusqu'à X compris
- NAK(X) signale que le segment X est en erreur
- Le *sender* en cas de perte ou d'erreur d'un segment ne retransmet que ce segment
- ça nécessite un timer par segment

# Répétition sélective

côté *receiver*

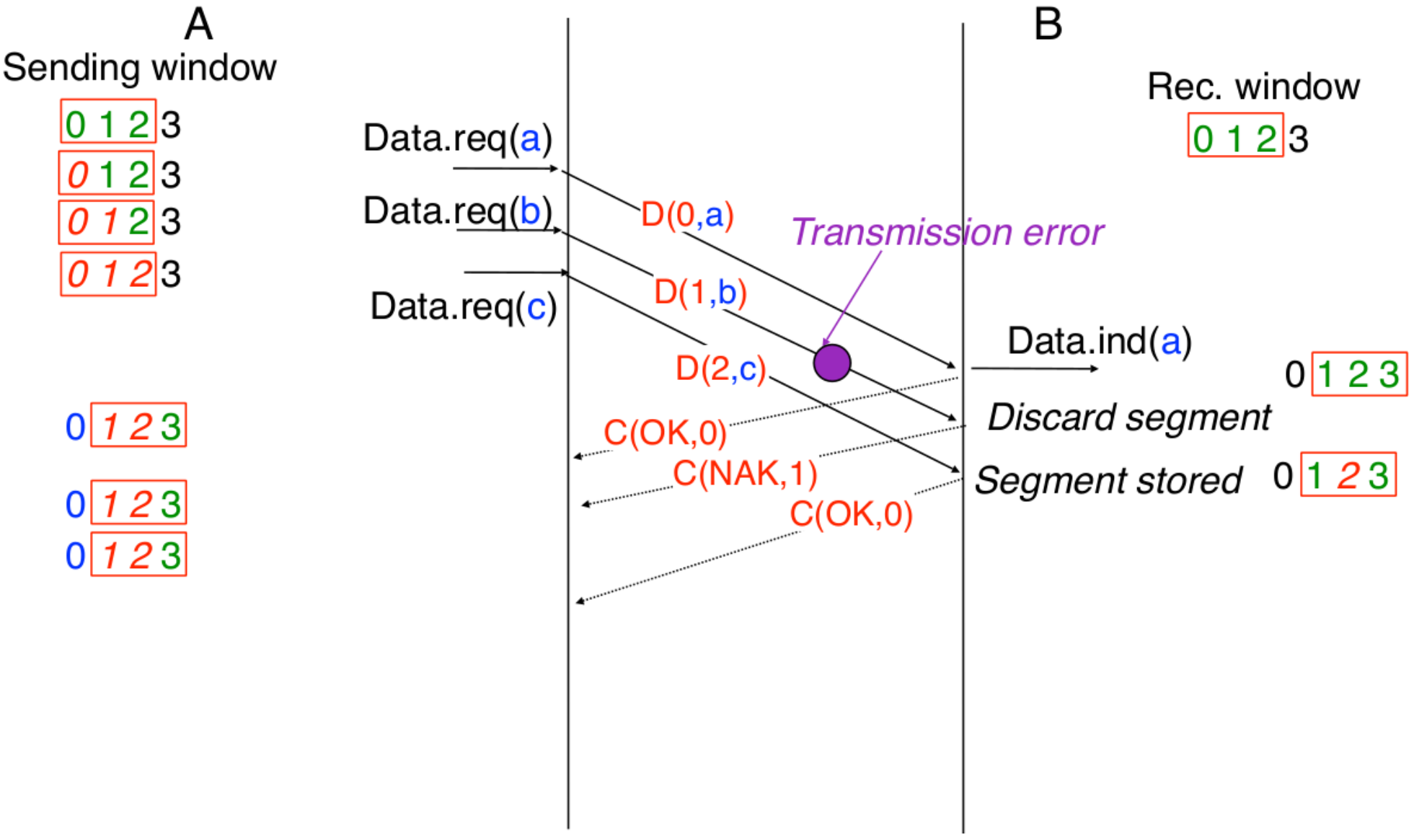
`next` : sequence number of expected data segment
`Last` : last received in-sequence segment



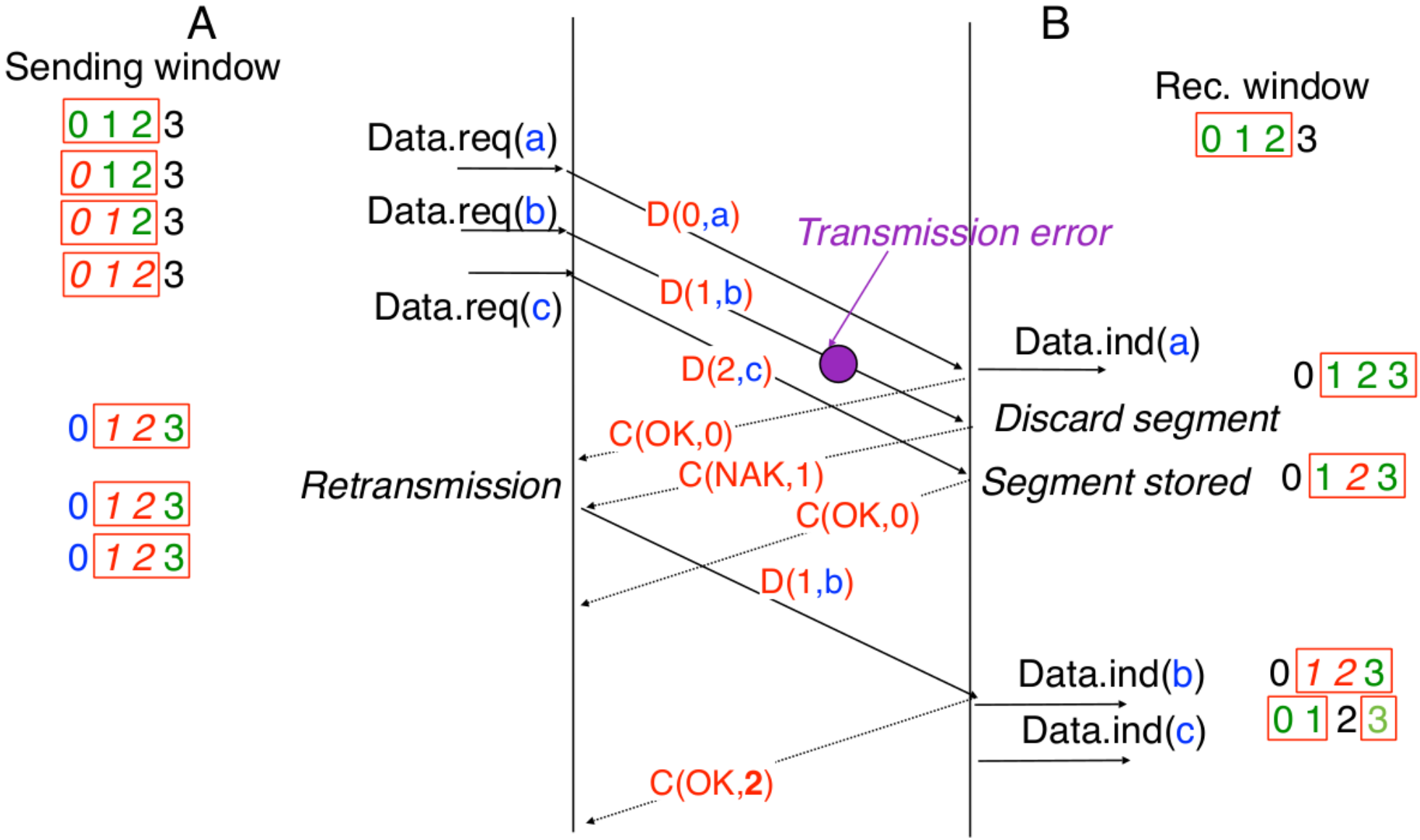
# Répétition sélective

côté *sender*

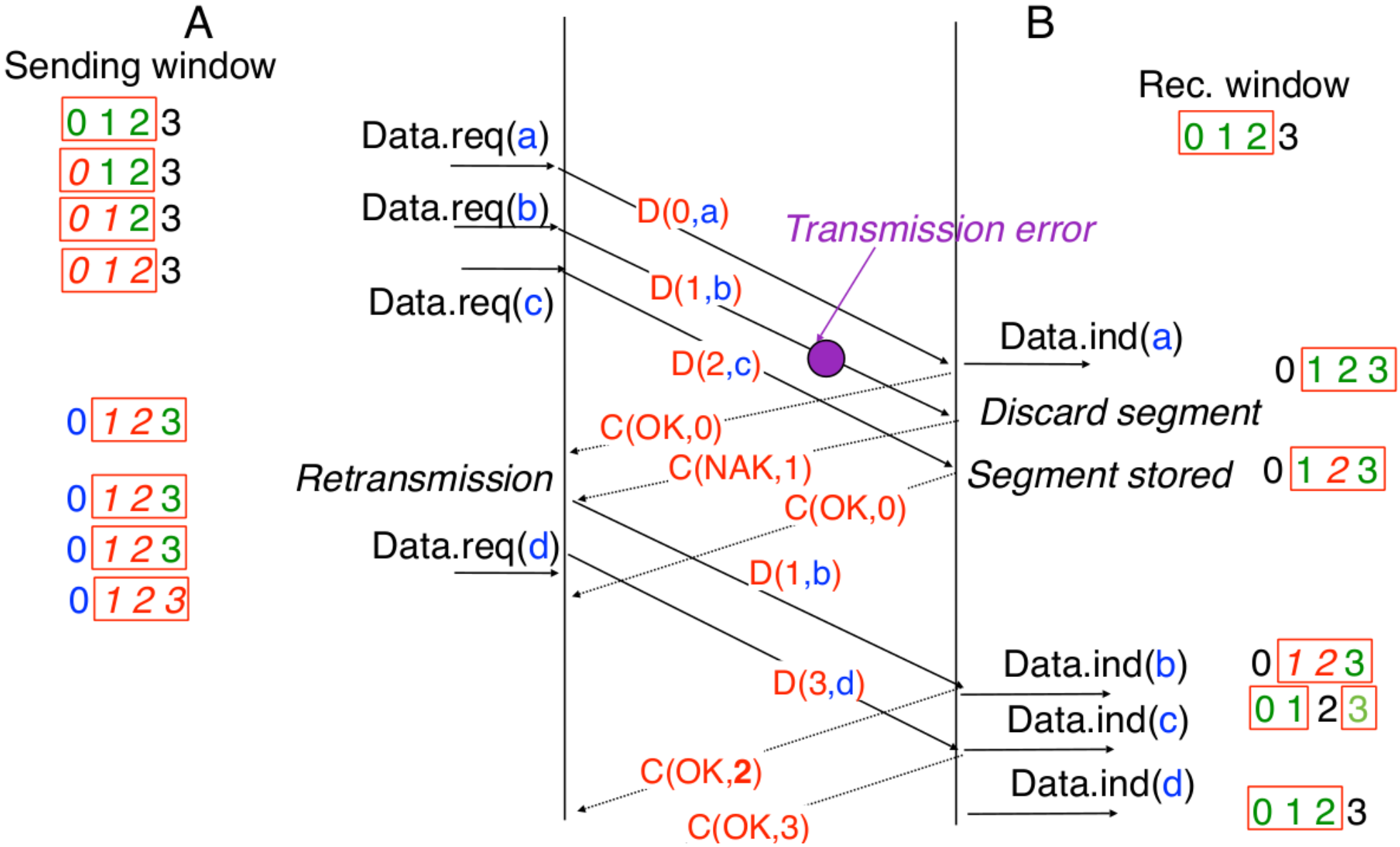
# Répétition sélective : exemple de scénario 1



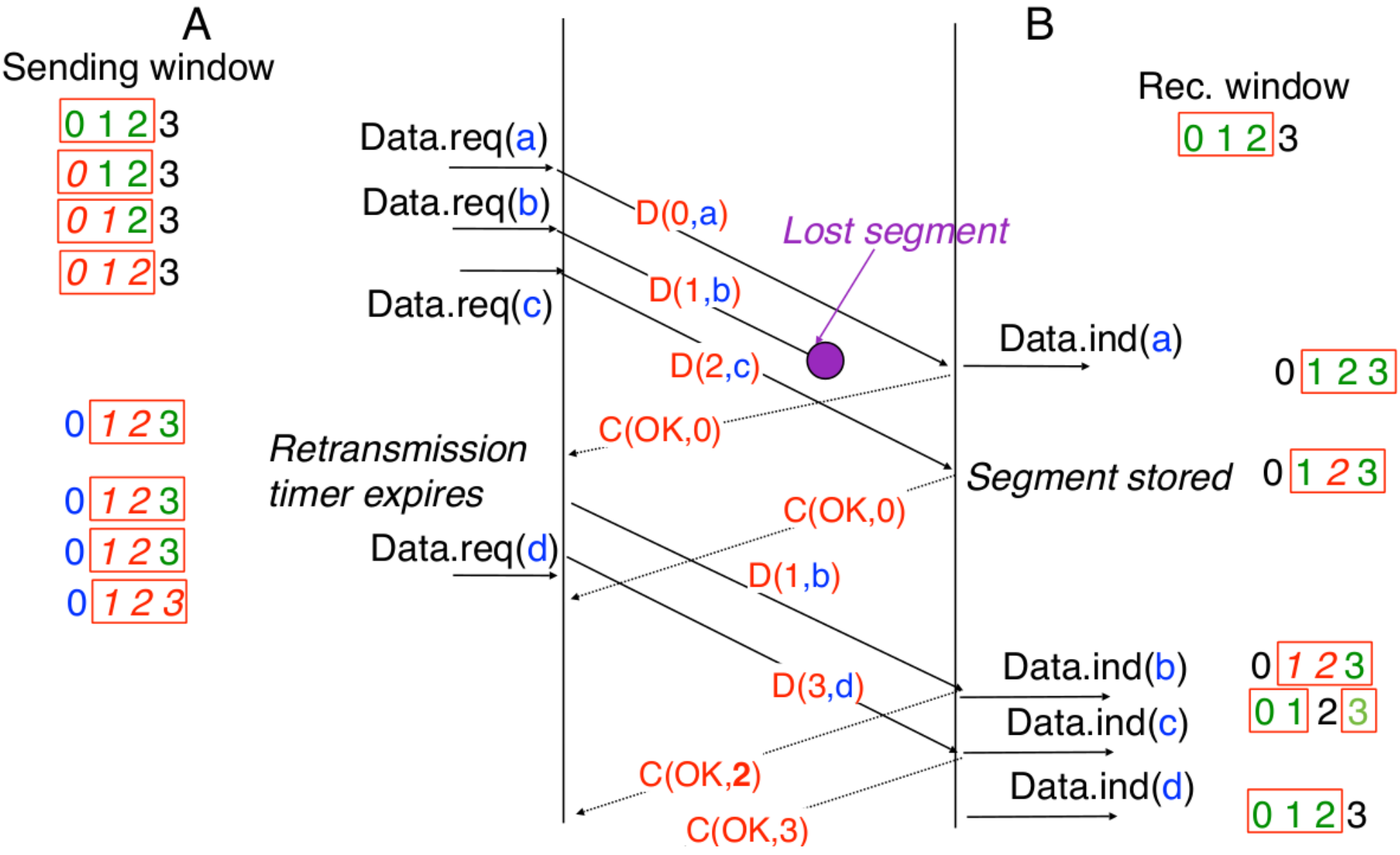
# Répétition sélective : exemple de scénario 1



# Répétition sélective : exemple de scénario 1



# Répétition sélective : exemple de scénario 2



# Répétition sélective : Taille de la fenêtre

$$W \leq 2^{k-1}$$

- $W$ taille de la fenêtre
- $k$ nombre de bits pour la numérotation des segments

* Comme on peut acquitter individuellement pour lever l'ambiguïté ancienne/nouvelle trame on prend la moitié de la numération au plus.

# La taille des buffers

La couche transport peut avoir à gérer plusieurs connections.

- Comment garantir qu'il n'y aura pas de débordements

–> Le nombre de connections varie dans le temps

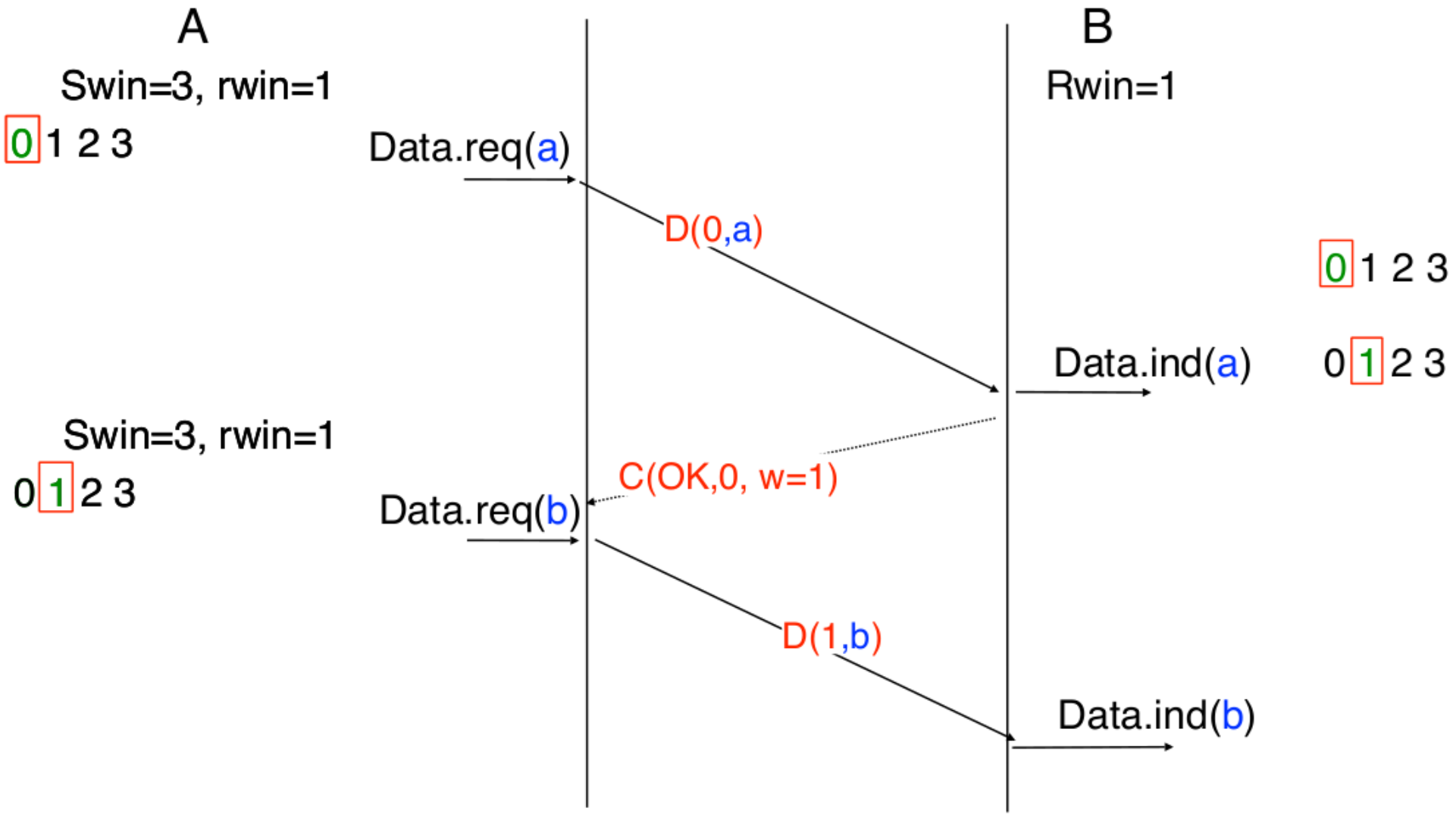
–> Les ressources nécessaires pour chaque connection diffèrent

# Gestion des buffers

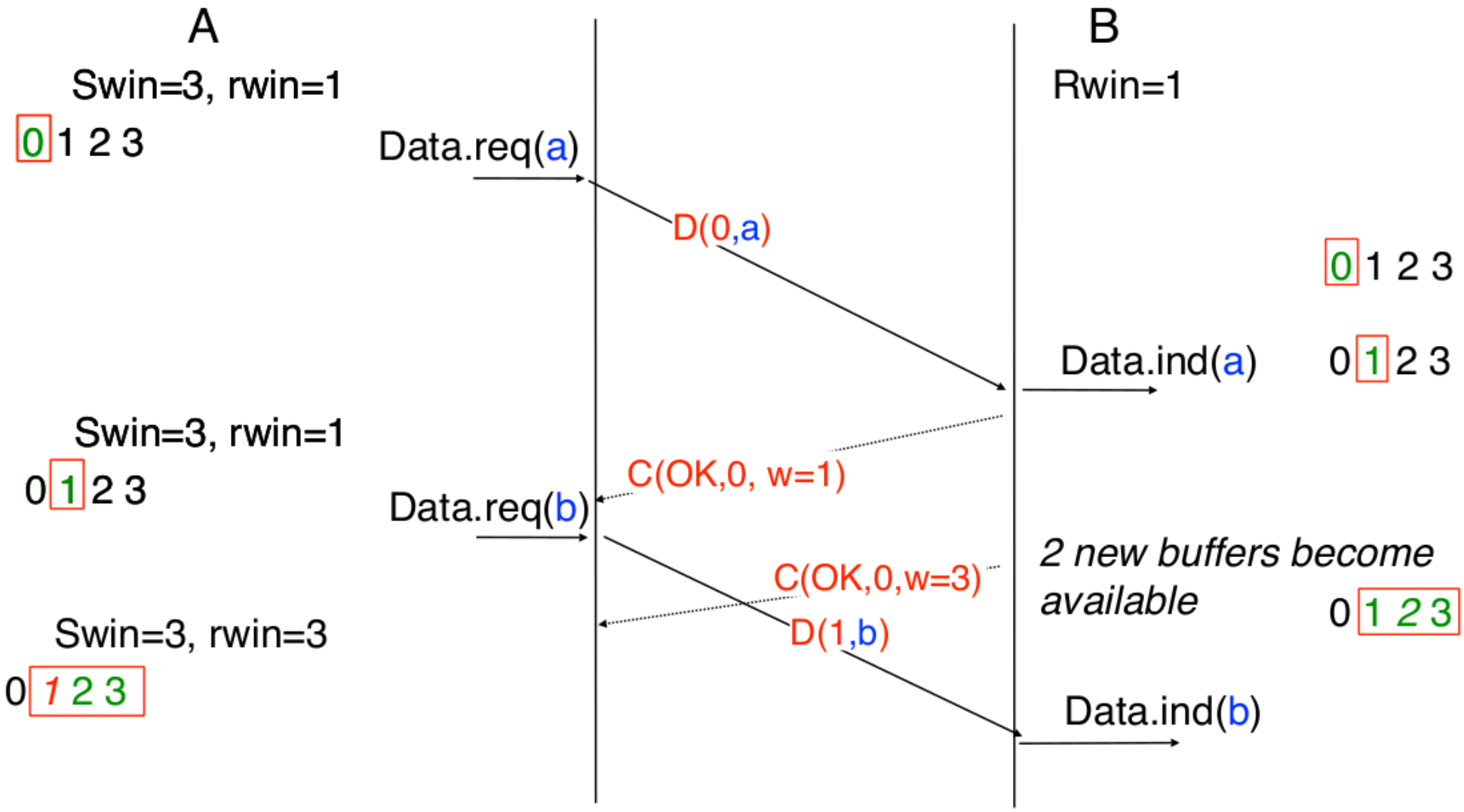
- **Principe :**
  - Le *receiver* ajuste la taille de sa fenêtre de réception.
  - Il annonce dynamiquement la taille de sa fenêtre de réception. Cette information est incluse dans les segments de contrôle.
- **Modifications côté *sender* :**
  - La fenêtre d'émission (`swin`) dépend de la mémoire disponible.
  - Une variable d'état stocke la taille de la fenêtre de réception annoncée par le *receiver* (`rwin`).
  - Le *sender* ne peut envoyer que des segments dont le numéro respecte la contrainte :

$$\text{Numéro de séquence} \in \left[0, \min(\mathit{rwin}, \mathit{swin})\right]$$

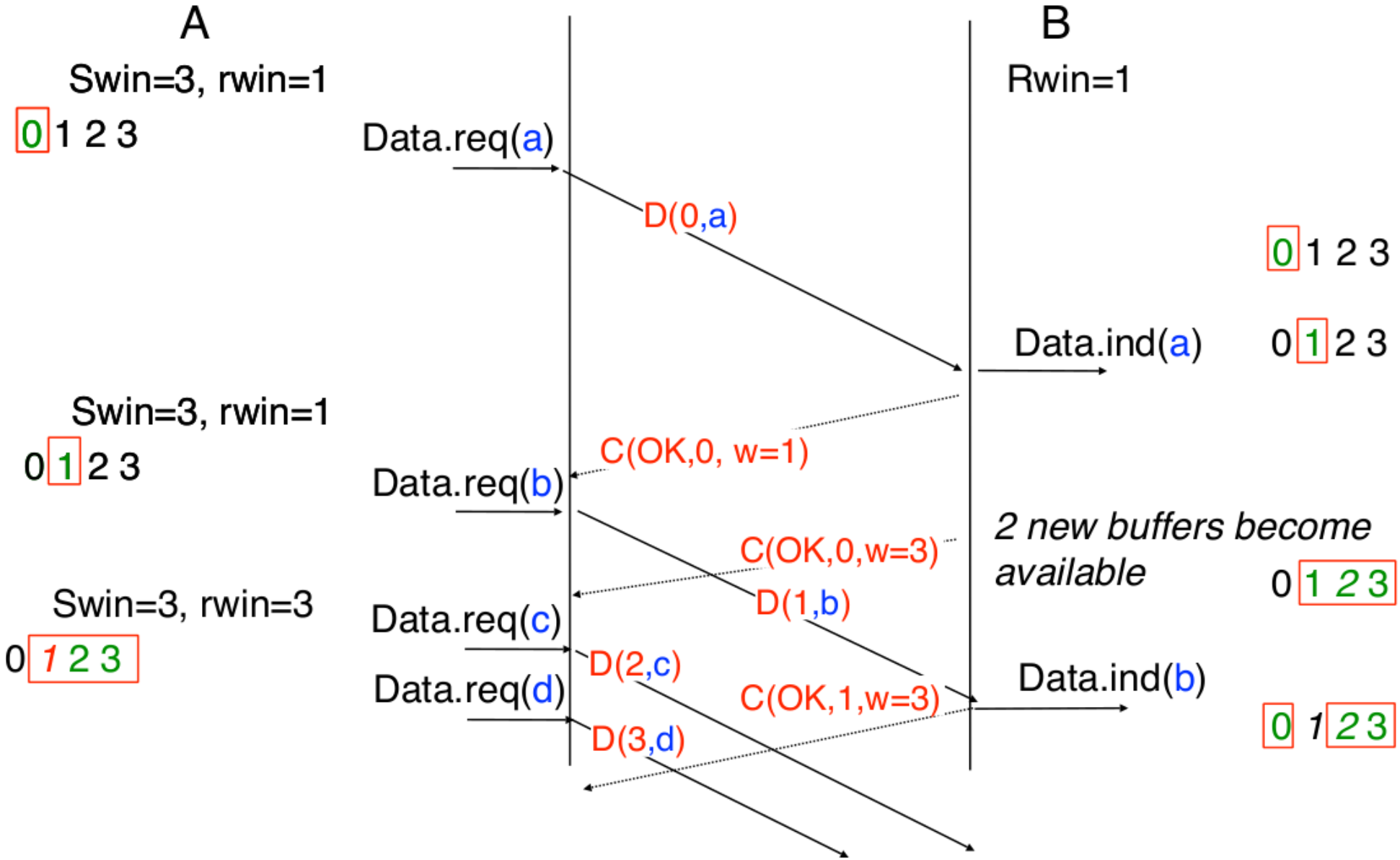
# Gestion des buffers



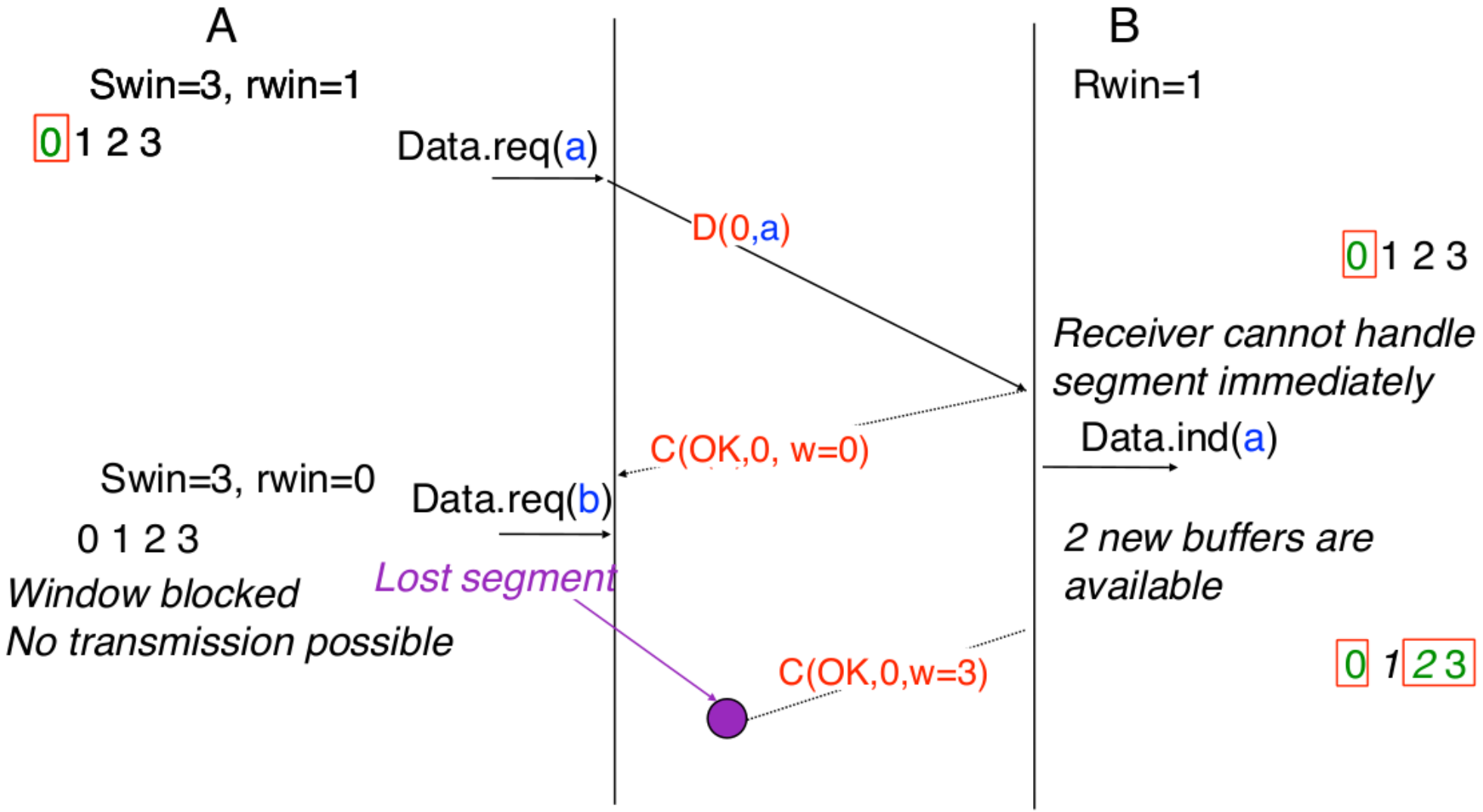
# Gestion des buffers



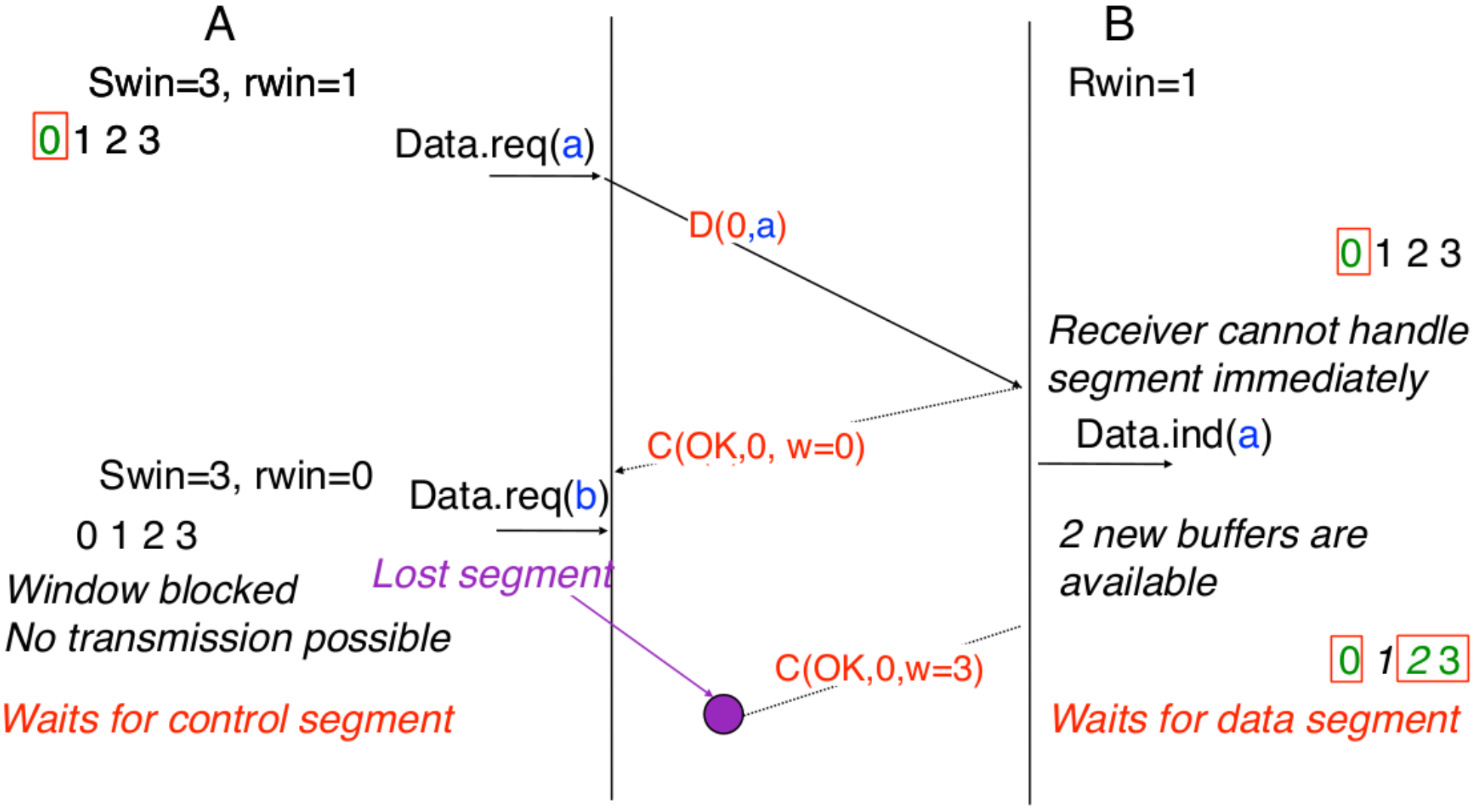
# Gestion des buffers



# Gestion des buffers : Attention



# Gestion des buffers : Attention




- un timer sur les segments contrôle

# TCP

Protocole **connecté** bâti au-dessus d'**IP**.

A la création, on alloue des structures de chaque côté pour s'occuper spécifiquement de la connexion.

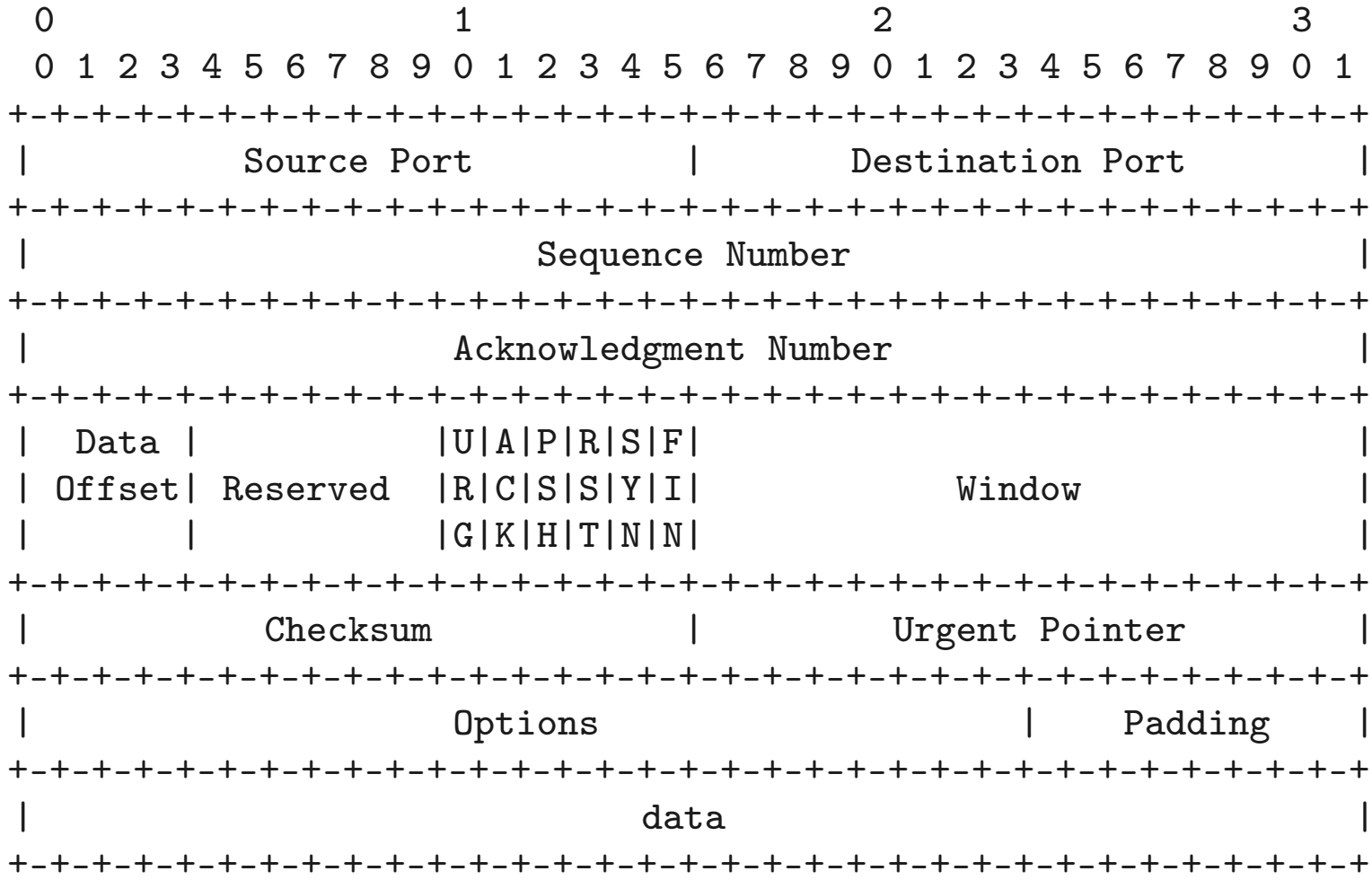
Ajout de la **fiabilité**.

Contrôle de la **congestion**.

L'unité de base en **TCP** s'appelle le **segment**

**MSS** : Max Segment Size.

# Entête TCP


0 1 2 3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
Source Port
Destination Port
Sequence Number
Acknowledgment Number
Data Offset
Reserved
U R G
A C K
P S H
R S T
S Y N
F I N
Window
Checksum
Urgent Pointer
Options
Padding
data

# Transfert fiable

**TCP** utilise une variante de **GoBackN**.

Contrôle au niveau des **octets** et non pas des segments.

**ACK** *n* : j'ai tout reçu jusqu'au *n*-ième octet non compris.

Pas de **NACK**.

# Transfert fiable

Position initiale (sur 32 bits) choisie **aléatoirement** et envoyée au correspondant lors de l'établissement de la connexion

Vise à **compliquer** l'usurpation d'identité lors d'une connexion **TCP**.

Optimisation : **fast retransmit** de $n$ si 3 **ACK** $n$ successifs.

Piggybacking + Delayed ACK

- Attendre 200ms avant de réenvoyer un ACK, au cas où il y aurait une réponse sur laquelle on peut se greffer (échange bidirectionnel)
- Si un deuxième paquet arrive, envoyer le ACK
- En régime "permanent", seulement un paquet sur deux sera acquitté

# Timeout

Comment **estimer le délai** avant réémission ?

Estimation du temps AR (ERTT : estimated round-trip time)

$$ERTT := \alpha ERTT + (1 - \alpha) RTT$$

RTT : temps AR du dernier paquet acquitté

$0.8 \leq \alpha \leq 0.9$

Comment fixer la durée de l'alarme ?

$$TIMER := ERTT + 4 Deviation$$

$$Deviation := \alpha Deviation + (1 - \alpha)|RTT - ERTT|$$

# Création de la connexion

Vérifier que les deux parties veulent communiquer

Allouer les buffers et autres structures nécessaires

Se mettre d'accord sur les numéros de séquence (positions initiales)

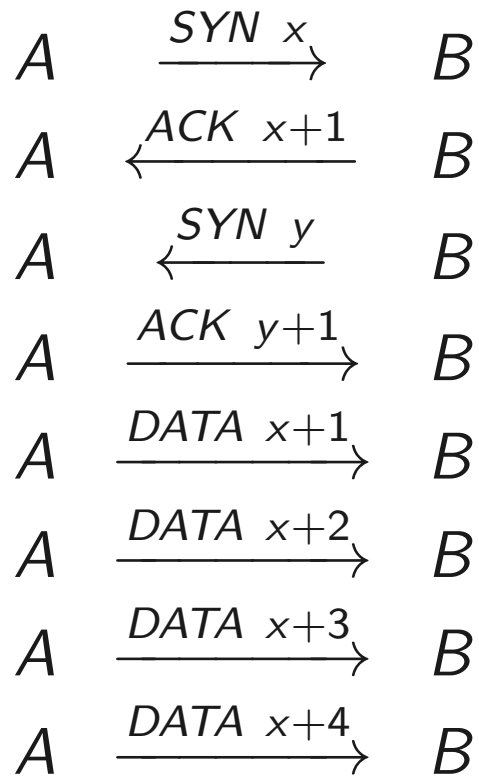
# Création de la connexion

Paquet **SYN** $x$ : je commence à numéroter à $x$

$$
\begin{array}{lcl}
A & \xrightarrow{SYN\ x} & B \\
A & \xleftarrow{ACK\ x+1} & B \\
A & \xleftarrow{SYN\ y} & B \\
A & \xrightarrow{ACK\ y+1} & B
\end{array}
$$

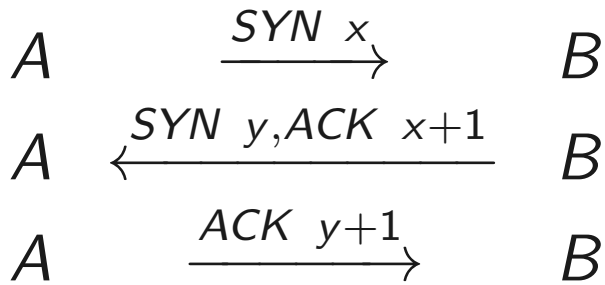
# Création de la connexion

Paquet **SYN** $x$ : je commence à numéroter à $x$



# Création de la connexion

On peut faire SYN+ACK en même temps.




**Three-way handshake** : poignée de main en 3 temps.

En pratique $A$ peut envoyer des données dès le 3e paquet.

# Fermeture de la connexion

Prévenir qu'on n'a plus rien à envoyer

Libérer les ressources allouées

Attention, l'autre partenaire peut avoir quelque chose à envoyer

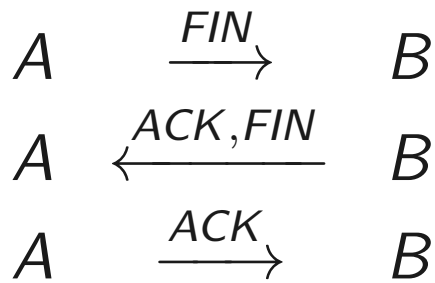
# Fermeture

$$A \xrightarrow{FIN} B$$
$$A \xleftarrow{ACK} B$$
$$A \xleftarrow{FIN} B$$
$$A \xrightarrow{ACK} B$$

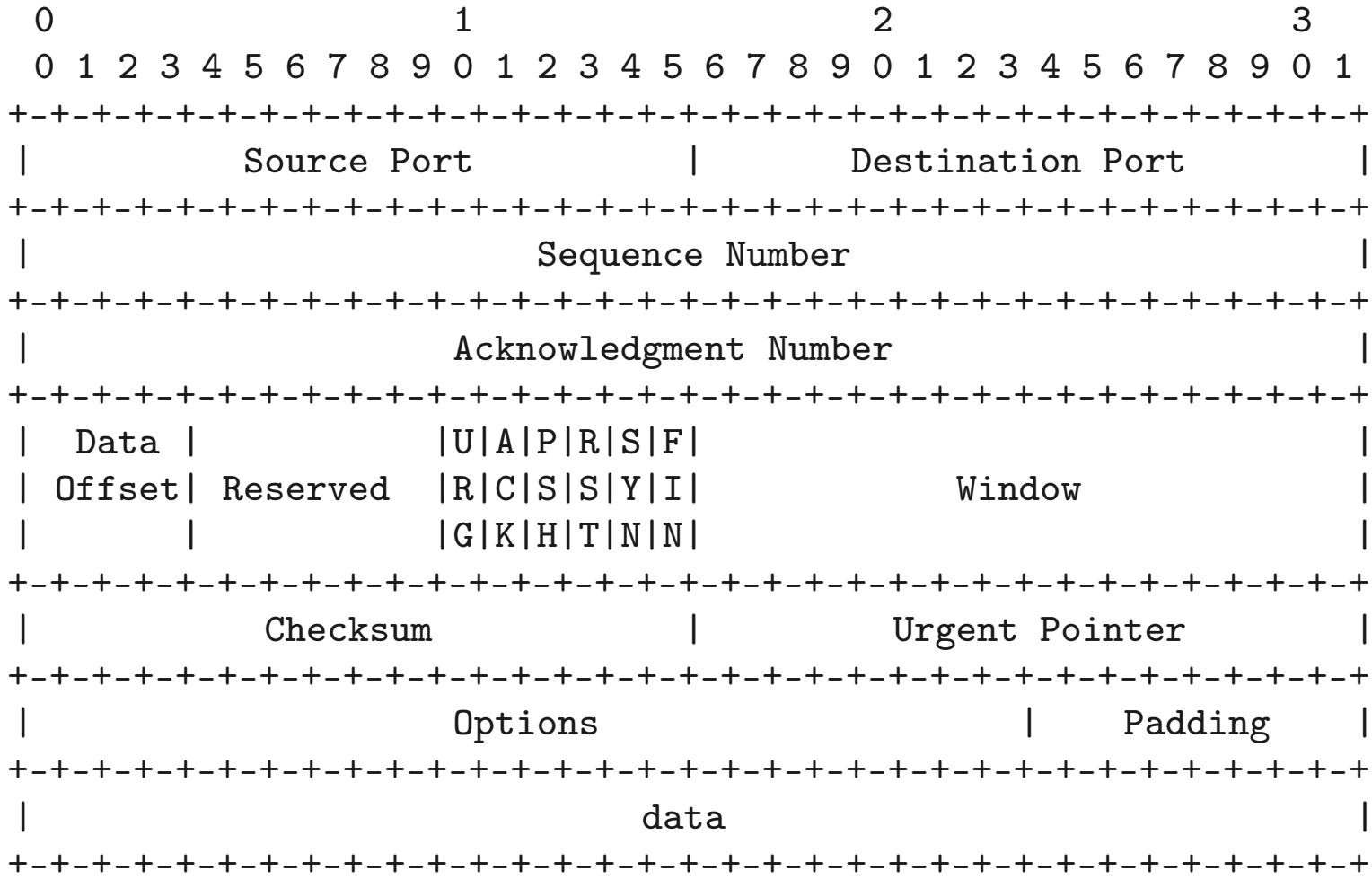
Si les deux segments *FIN* et *ACK* envoyés par $B$ arrivent dans le désordre ?

*TIME-WAIT* pour éviter les problèmes (libération du port, conflit avec d'anciennes trames).

# Fermeture avec Piggybacking



# Entête TCP


0 1 2 3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
Source Port
Destination Port
Sequence Number
Acknowledgment Number
Data Offset
Reserved
U R G
A C K
P S H
R S T
S Y N
F I N
Window
Checksum
Urgent Pointer
Options
Padding
data

# Contrôle de flux

- Chacun des deux participants a un buffer de réception
- Prévenir que le buffer est presque plein
- $B$ prévient $A$ qu'il lui reste $p$ octets disponibles dans son buffer à chaque message qu'il envoie (champ *Window* de l'entête TCP)
- $A$ doit s'arranger pour que

$$DernierBitEnvoye - DernierBitAcquitte \leq p$$

# Problème

v1 - avec un timer

- $B$ envoie un message comme quoi $p = 0$
- Que se passe-t-il ? Comment le *sender* sait qu'il peut à nouveau envoyer des segments

# Problème

v1 - avec un timer

- $B$ envoie un message comme quoi $p = 0$
- Que se passe-t-il ? Comment le *sender* sait qu'il peut à nouveau envoyer des segments
- Solution
  - $A$ envoie des paquets de 1 octet (*probe*) que $B$ acquitte.
  - Tant que $B$ acquitte avec $p = 0$ le *Persistence* Timer est relancé pour renvoyer le *probe*.

# Problème

v2 - le récepteur envoie une fenêtre trop petite trop souvent

- $B$ est beaucoup plus lent que $A$
- En régime “permanent”, chaque segment fera un seul octet
- *Silly Window Syndrome* (SWS)

# Solution au SWS

Uniquement côté destinataire

- Attendre que $p$ redevienne suffisamment grand
- Par exemple taille max d'un segment ou la moitié de la taille totale
- Que faire en attendant ?
  - Ne pas envoyer de ACK
  - Envoyer des ACK avec $p = 0$

# Problème côté émetteur

v3 tinygram

Eviter les petits paquets niveau émetteur (*Algorithme de Nagle*)

- *data.req(a)* avec *a* de taille $>$ MSS
  $->$ les envoyer
- *data.req(a)* avec *a* petits
  - S'il y a des données non acquittées, attendre pour cumuler sur un segment
  - Sinon envoyer immédiatement

Attention

- Peut être indésirable (mouvement de la souris)
- Conflit avec le Delayed ACK
- Peut être désactivé (`TCP_NODELAY`)

# Problématique

- Contrôler la congestion (qui vient du réseau lui-même)
- Essentiellement au niveau des routeurs qu'on ne contrôle pas
- Quelques infos par ICMP (globales !), mais aussi dans TCP lui même (par connexion)

# Congestion

Problèmes

Dès que le débit est trop élevé

- Délais importants
- Retransmission de segments dans la file d'attente d'un routeur
- Paquets dupliqués envoyés par le routeur
- Les paquets abandonnées par le routeur correspondent à du trafic gâché

# S'en occuper

- Qui doit s'en occuper ?
  - Couche réseau (tous les participants, routeurs compris)
  - Couche transport (uniquement les extrêmités)
- Choix TCP/IP : Dans la couche transport

# S'en occuper

Comment s'en occuper

- Prévenir plutôt que guérir
- Comment contrôler le débit dans TCP ?

# S'en occuper

Comment s'en occuper

- Prévenir plutôt que guérir
- Comment contrôler le débit dans TCP ?
  - En changeant la taille de la fenêtre... (déjà vu)

# Congestion dans TCP : AIMD

Additive Increase - Multiplicative Decrease

Point de vue : les pertes de paquets viennent uniquement des files d'attente dans les routeurs

- Fenêtre de congestion de taille $F$
- Seuil $\lambda$ de congestion.
- Si $F < \lambda$
  - Au début, taille de la fenêtre de 2 MSS (*slow start*)
  - A chaque ACK, on double la fenêtre
  - Augmentation *exponentielle*
- Si $F \geq \lambda$
  - Augmentation de 1MSS tous les $F$ ACK
  - Augmentation *linéaire*
- Si perte de paquet (timeout) :
  - $\lambda = F/2$
  - La fenêtre passe à 1 MSS

# Conséquences

Conséquence pour l'utilisateur

- A cause du *slow start*, il vaut mieux une connexion pour transférer deux fichiers que deux connexions consécutives.
- Sans oublier les coûts de connexion...
- Exemple : HTTP
  - Keep-Alive
  - Pipelining

# Routeurs

- $n$ machines derrière un routeur
- Stratégie du routeur : supprimer les nouveaux paquets quand la file est pleine
- Problème : quand la file d'attente est pleine, *toutes* les connexions vont être pertubées et redémarrer

# RED

Random Early Detection/Drop/Discard

- File d'attente du routeur séparée en deux "moitiés"
- On remplit la première partie
- Si la première partie est pleine, on accepte dans la file d'attente un nouveau paquet IP qu'avec probabilité $p$
- Bien choisir $p$

# Droits

Merci à l'équipe historique Réseaux en L3 Nicolas Ollinger, Martin Delacourt et Mathieu Liedloff.
Les chronogrammes et les FSM sont issus de CNP3 diffusés sous licence CC-BY-SA-3.0 et produits par Olivier Bonaventure (Université catholique de Louvain).