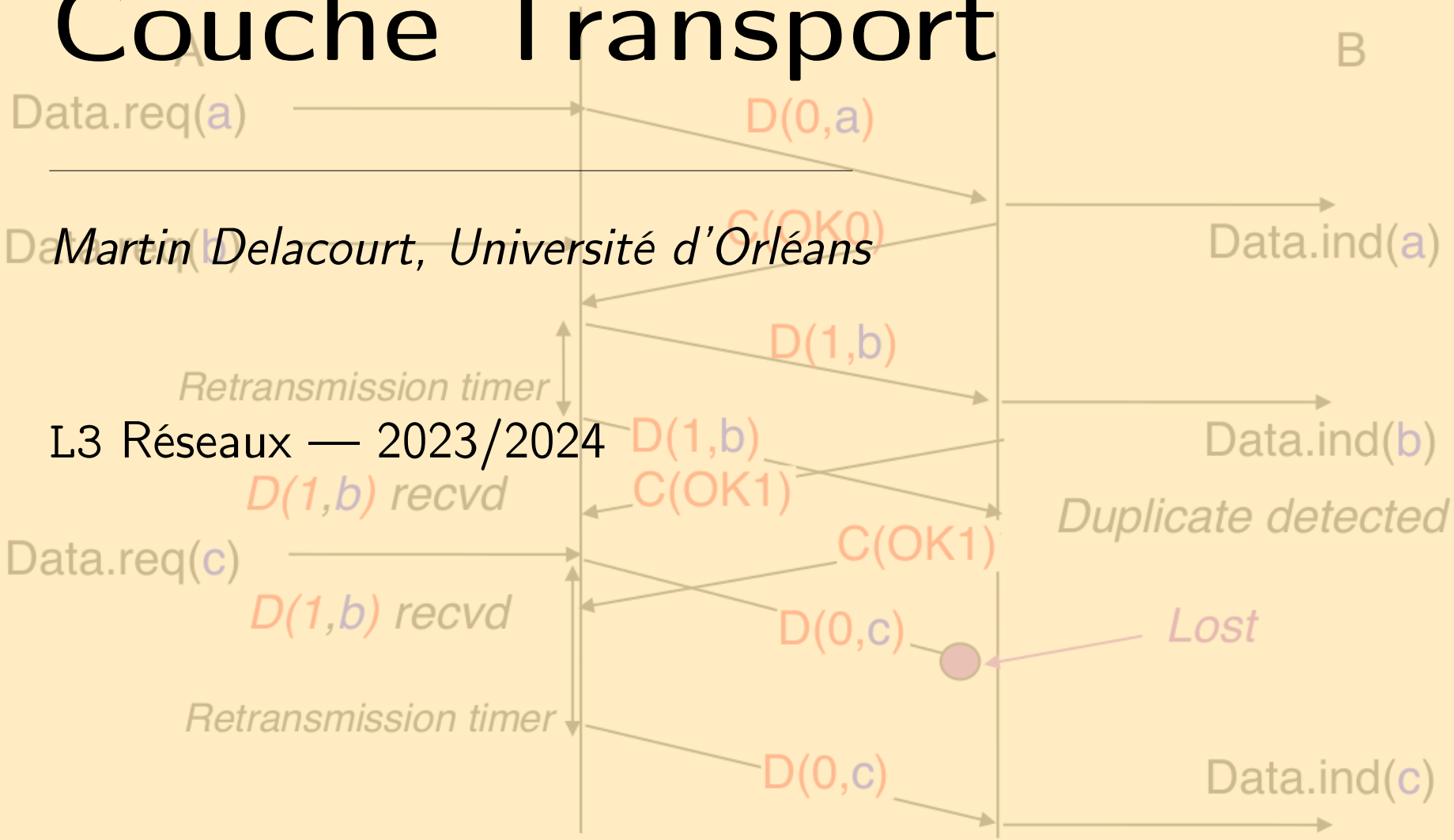
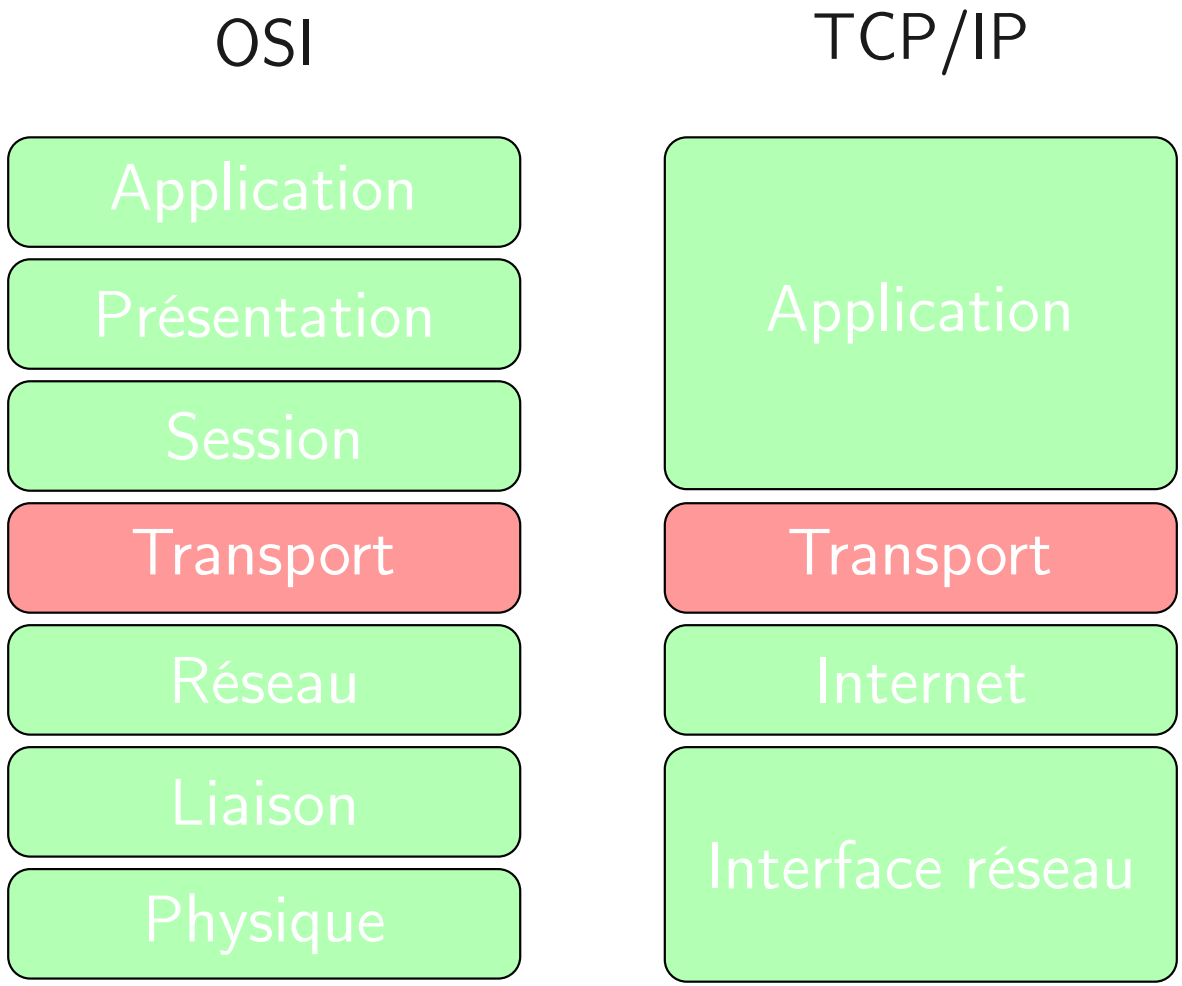
Couche Transport



Martin Delacourt, Université d'Orléans

L3 Réseaux — 2023/2024

Les modèles


OSI
TCP/IP
Application
Présentation
Session
Transport
Réseau
Liaison
Physique
Application
Transport
Internet
Interface réseau

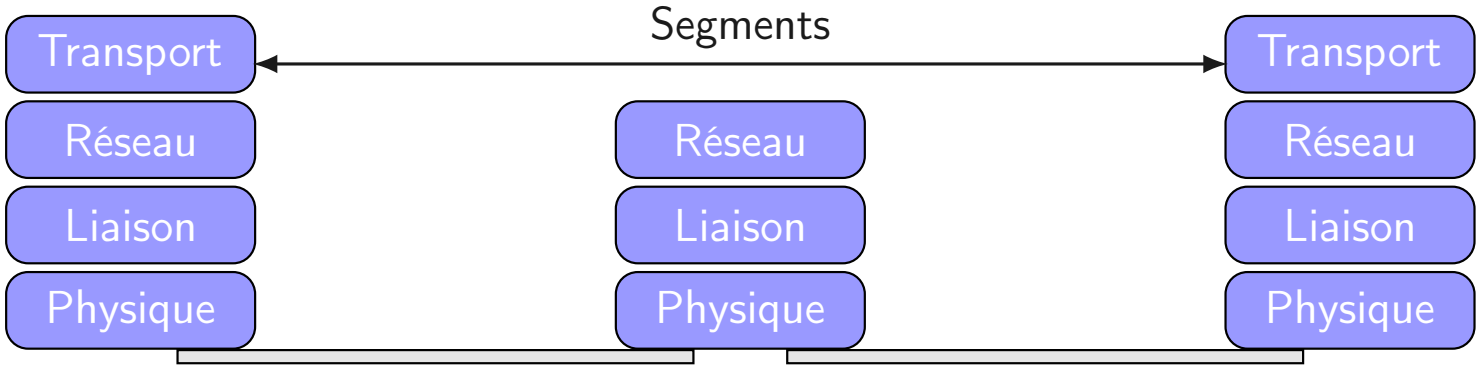
Couche réseau (Network)

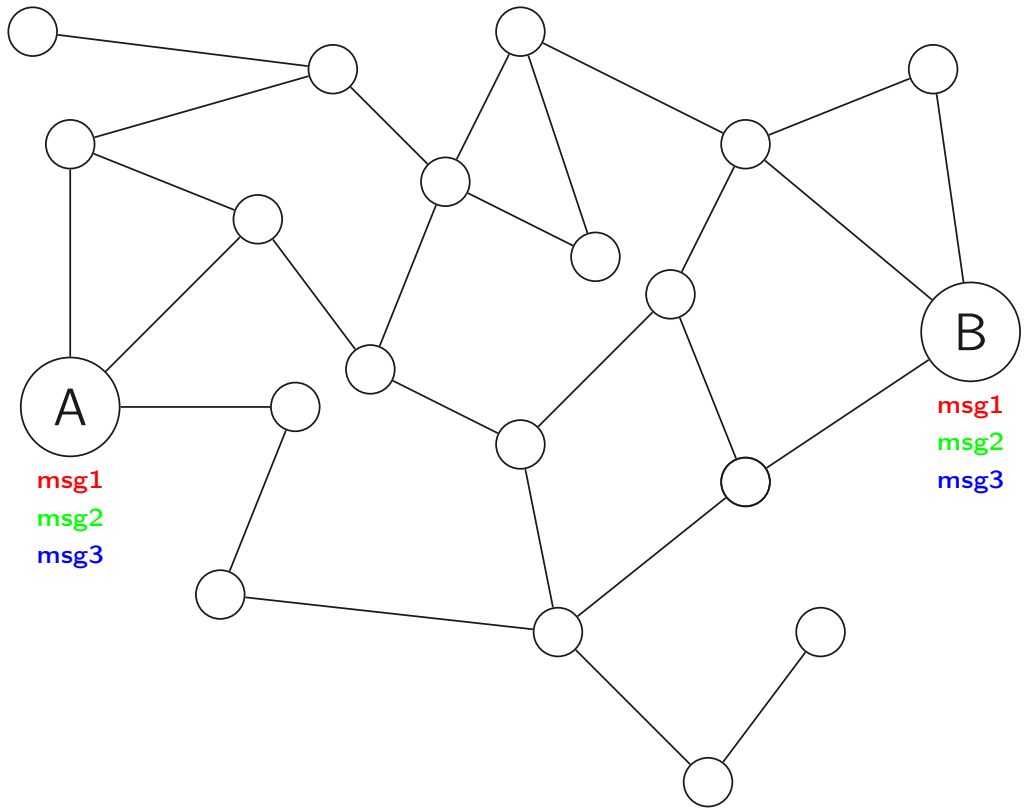
- Lien entre des points distants avec intermédiaires.
- Spécification des adresses pour le routage.
- Pertes, corruptions ou inversions potentielles de parties de messages.

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.

Communication identifiée par un **quadruplet**

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

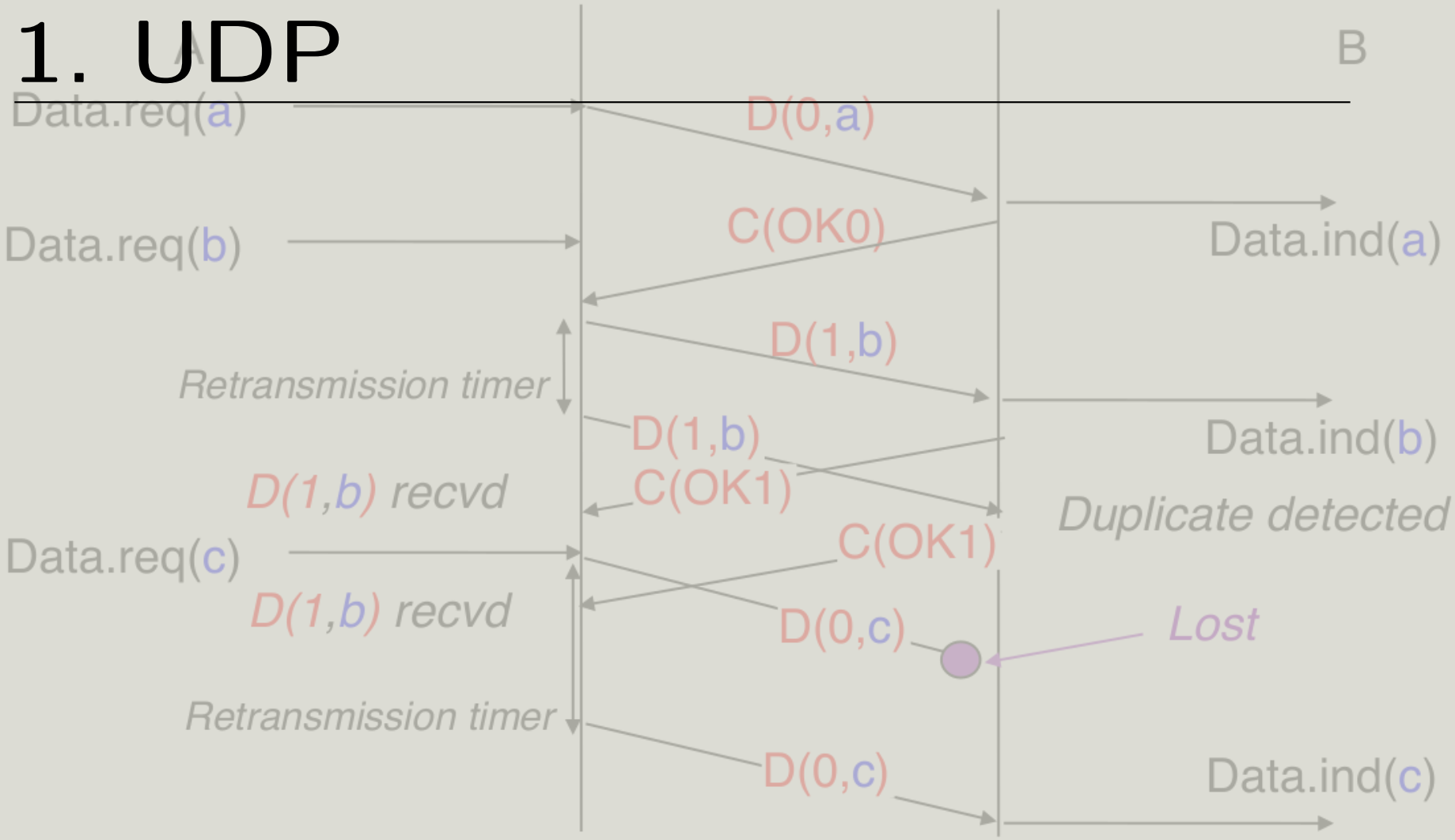
Network Address Translation

Si la machine A veut parler à la machine B sur le port b, elle doit d'abord choisir un port a.

Cf fonctionnement de **NAT**.

PAT (Port Address Translation)

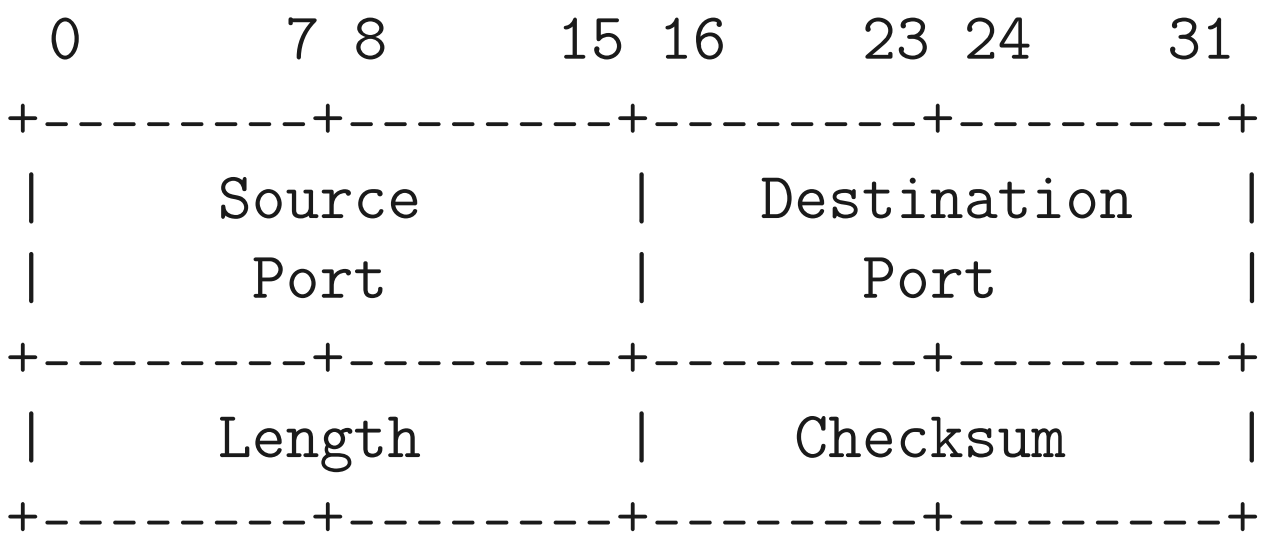
1. UDP


A
B
Data.req(a)
D(0,a)
Data.req(b)
C(OK0)
Data.ind(a)
D(1,b)
Retransmission timer
D(1,b)
Data.ind(b)
D(1,b) recvd
C(OK1)
Duplicate detected
Data.req(c)
C(OK1)
D(1,b) recvd
D(0,c)
Lost
Retransmission timer
D(0,c)
Data.ind(c)

UDP

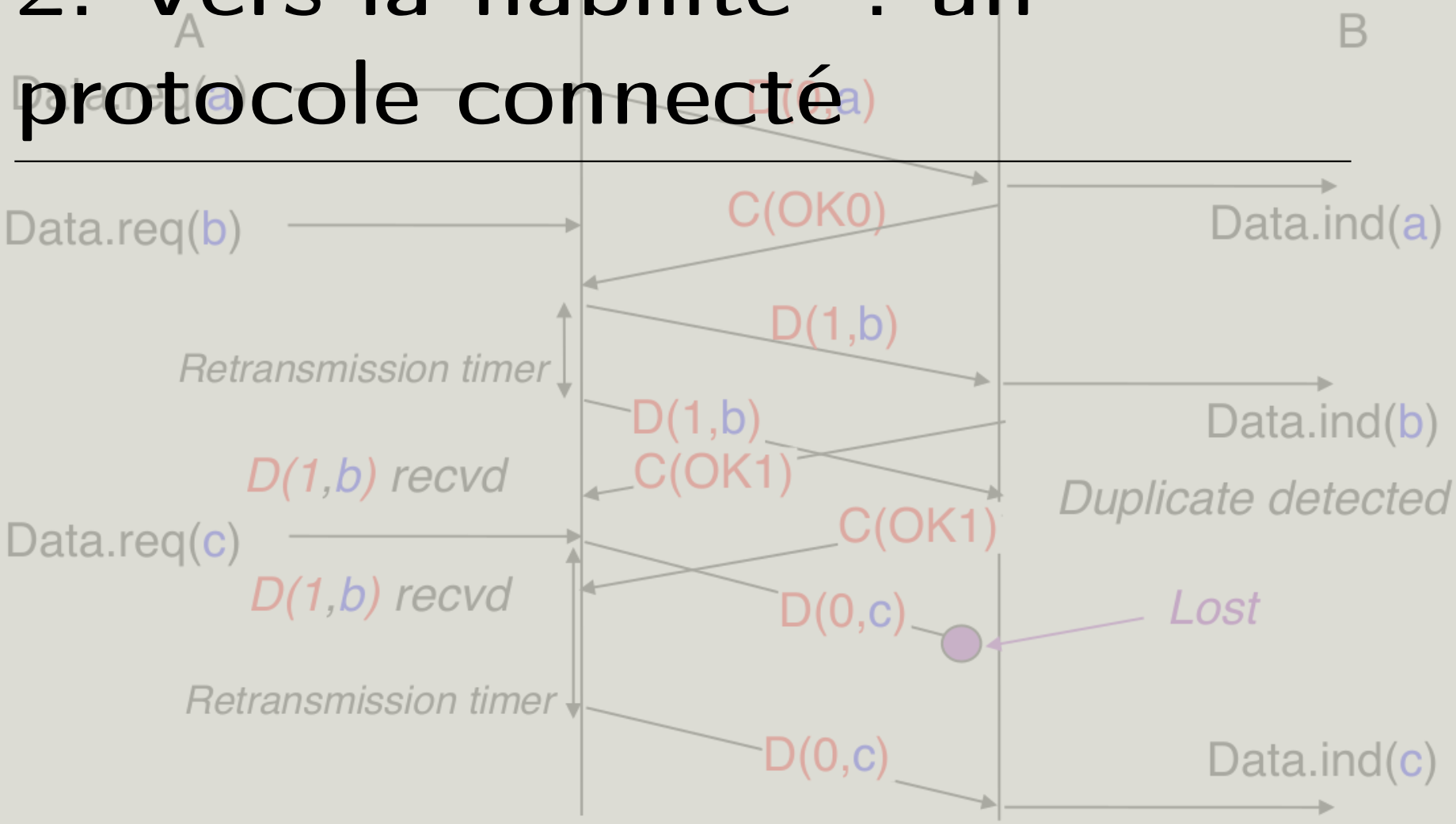
Ajoute les **ports** à IP, et un code détecteur d'erreurs.

Encapsulé dans IP.



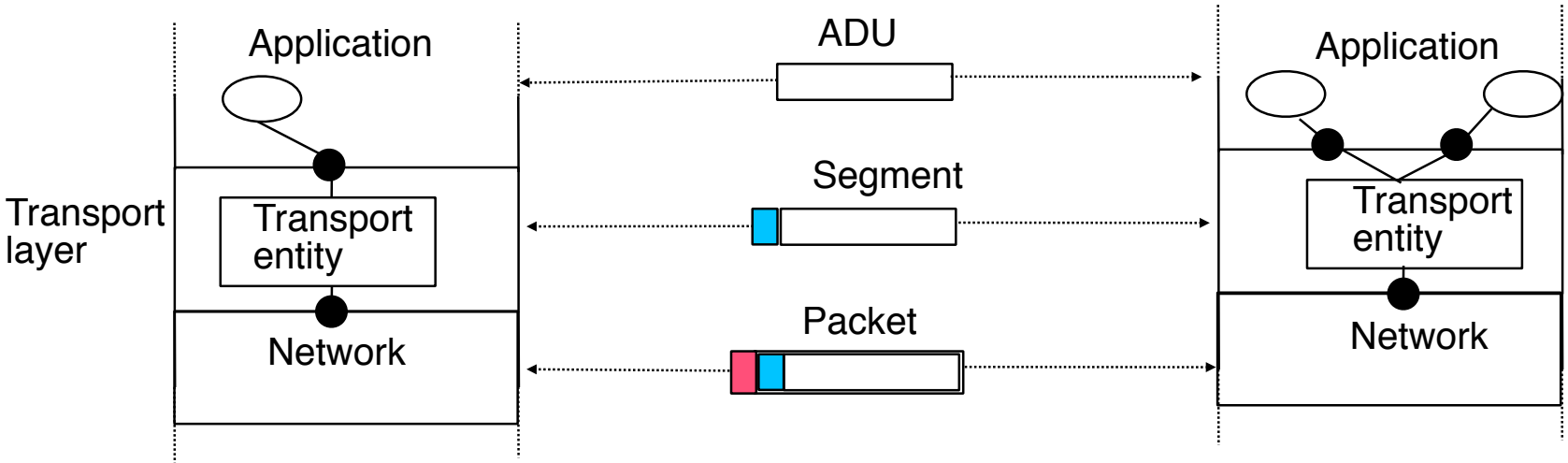
Checksum : complément à 1 de la somme des compléments à 1 des segments de 16 bits de (UDP + pseudo-header IP)

2. Vers la fiabilité : un protocole connecté



Transport layer protocols (2)

Reference environment



Notations

`data.req` and `data.ind` primitives for application/ transport interactions

`recv()` and `send()` for interactions between transport entity and network layer

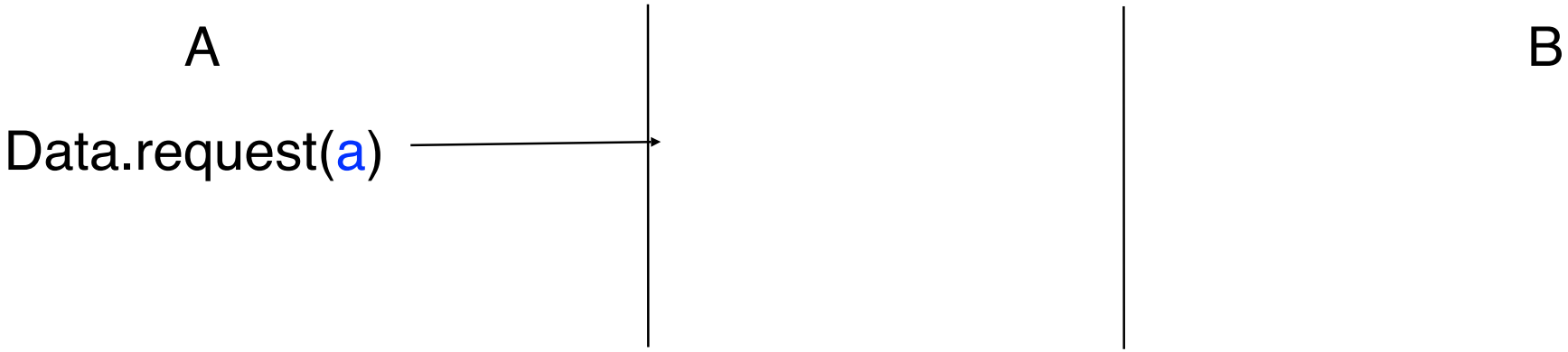
Protocol 1 : Basics

Principle

Upon reception of `data.request(SDU)`, the transport entity sends a segment containing this SDU through the network layer (`send(SDU)`)

Upon reception of the contents of one packet from the network layer (`recv(SDU)`), transport entity delivers the SDU found in the packet to its user by using `data.ind(SDU)`

Protocol 1 : Basics

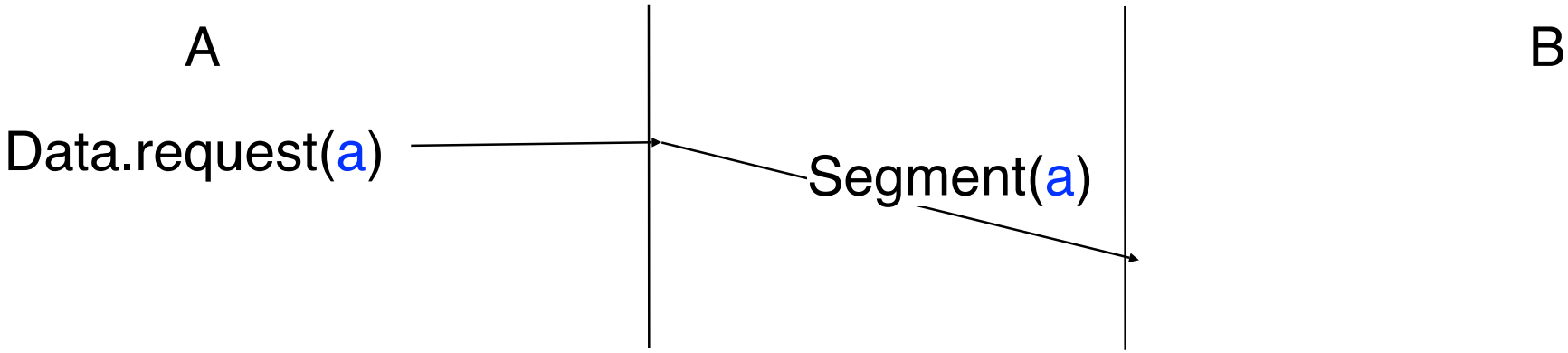


Principle

Upon reception of `data.request(SDU)`, the transport entity sends a segment containing this SDU through the network layer (`send(SDU)`)

Upon reception of the contents of one packet from the network layer (`recv(SDU)`), transport entity delivers the SDU found in the packet to its user by using `data.ind(SDU)`

Protocol 1 : Basics

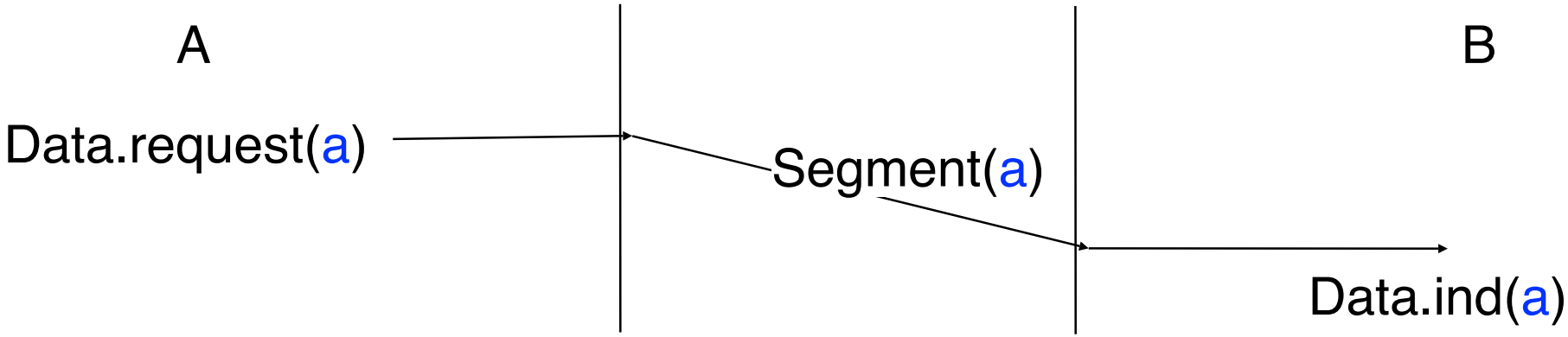


Principle

Upon reception of `data.request(SDU)`, the transport entity sends a segment containing this SDU through the network layer (`send(SDU)`)

Upon reception of the contents of one packet from the network layer (`recv(SDU)`), transport entity delivers the SDU found in the packet to its user by using `data.ind(SDU)`

Protocol 1 : Basics



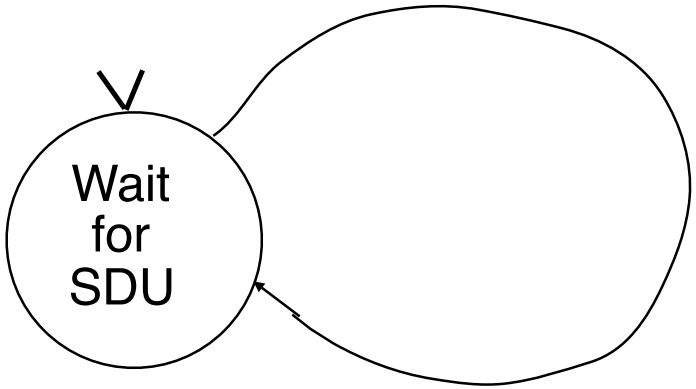
Principle

Upon reception of `data.request(SDU)`, the transport entity sends a segment containing this SDU through the network layer (`send(SDU)`)

Upon reception of the contents of one packet from the network layer (`recv(SDU)`), transport entity delivers the SDU found in the packet to its user by using `data.ind(SDU)`

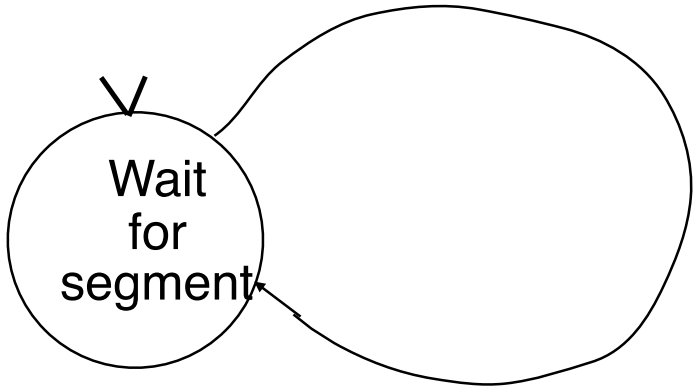
Protocol 1 as a FSM

Sender



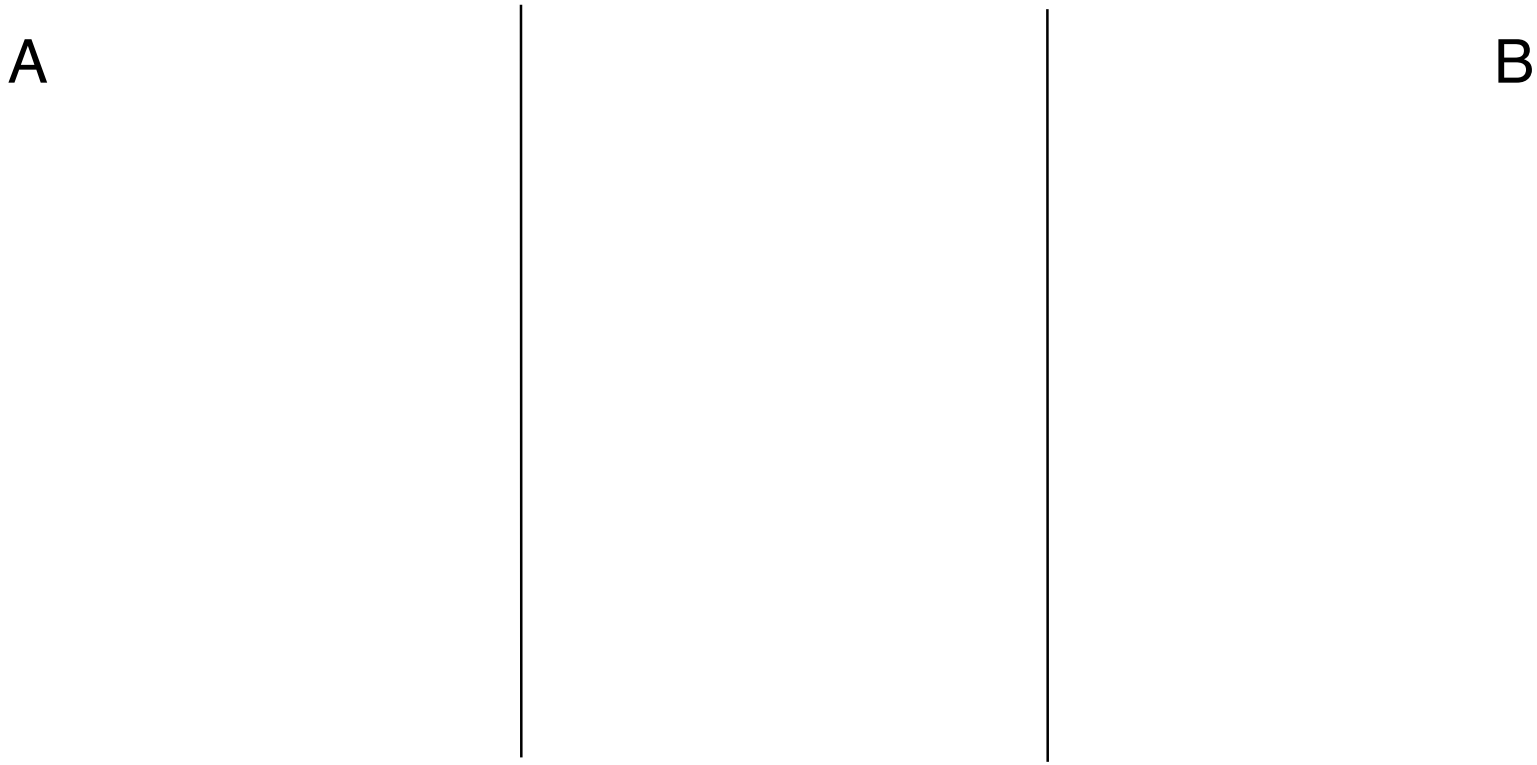
Data.req(SDU)
send(SDU)

Receiver



recvd(SDU)
Data.ind(SDU)

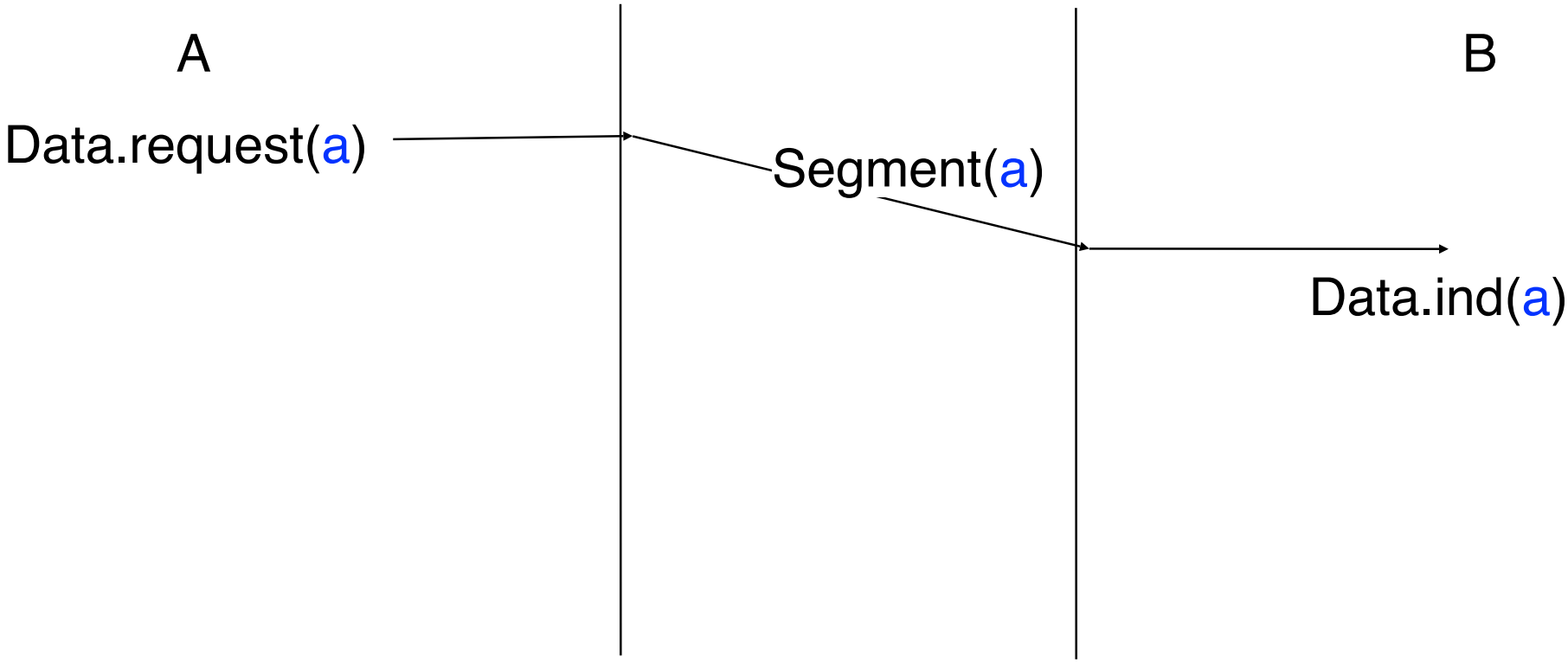
Protocol 1 : Example



Issue

- What happens if the receiver is much slower than the sender ?
  - e.g. receiver can process one segment per second while sender is producing 10 segments per second ?

Protocol 1 : Example

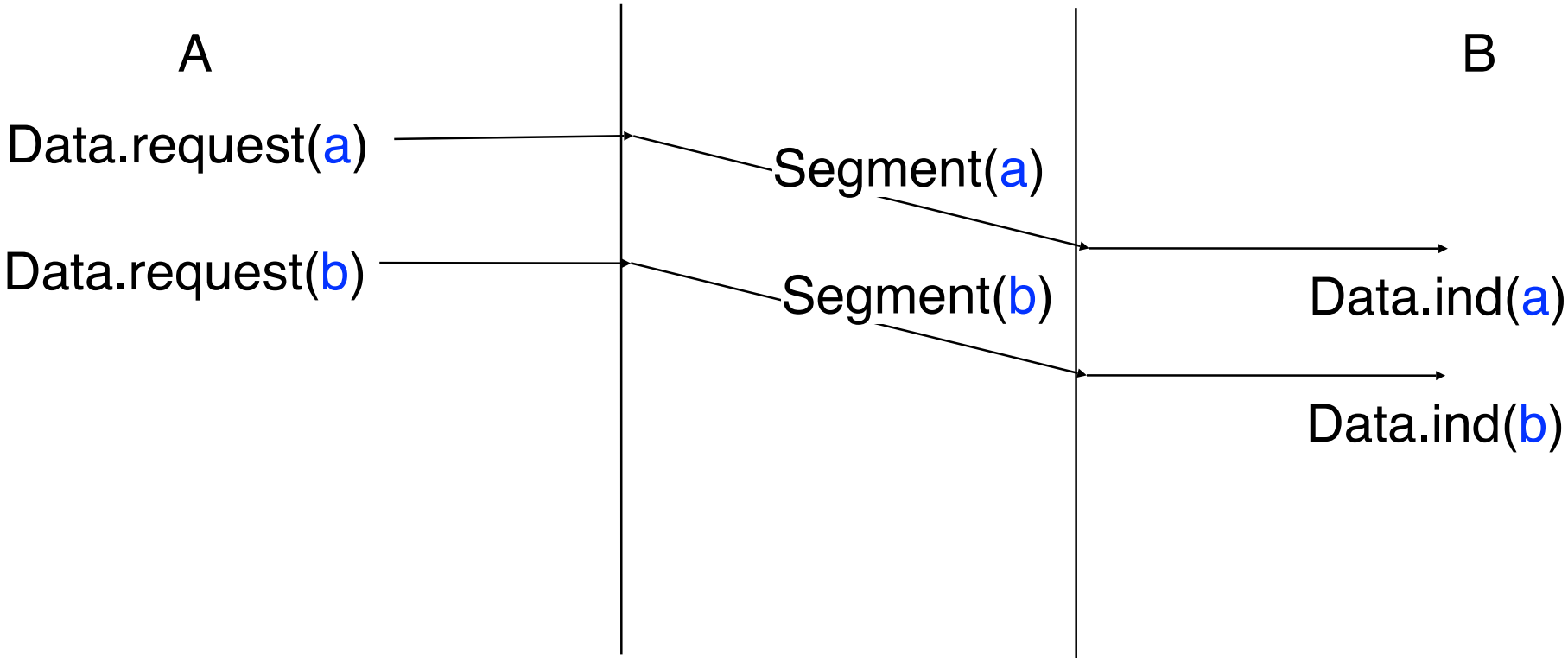


Issue

What happens if the receiver is much slower than the sender ?

e.g. receiver can process one segment per second while sender is producing 10 segments per second ?

Protocol 1 : Example

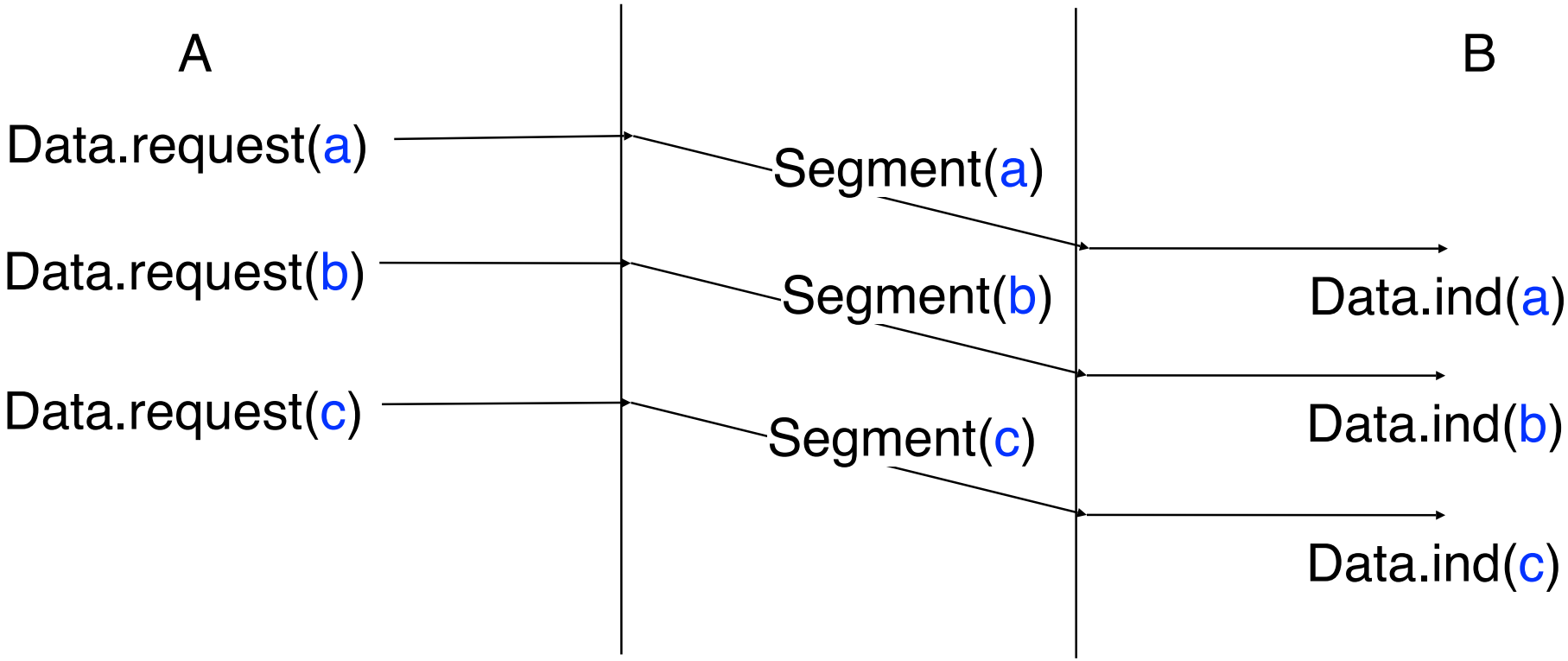


Issue

What happens if the receiver is much slower than the sender ?

e.g. receiver can process one segment per second while sender is producing 10 segments per second ?

Protocol 1 : Example



Issue

What happens if the receiver is much slower than the sender ?

e.g. receiver can process one segment per second while sender is producing 10 segments per second ?

Protocol 2

Principle

Use a control segment (OK) that is sent by the receiver after having processed the received segment

creates a feedback loop between sender and receiver

Consequences

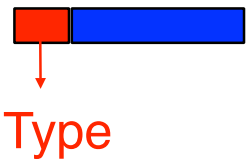
Two types of segments

Data segment containing on SDU

Notation : D(SDU)

Control segment

Notation : C(OK)

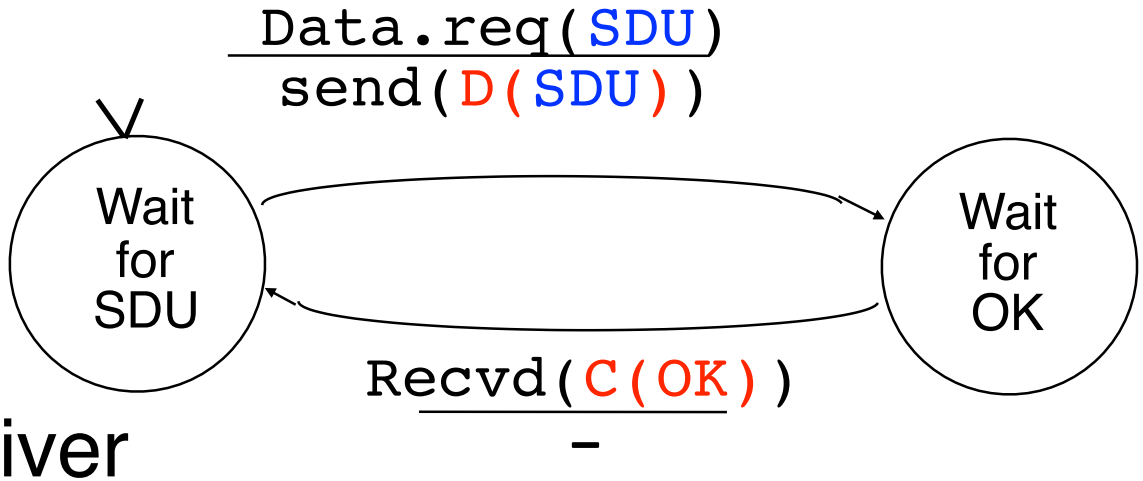


Segment format

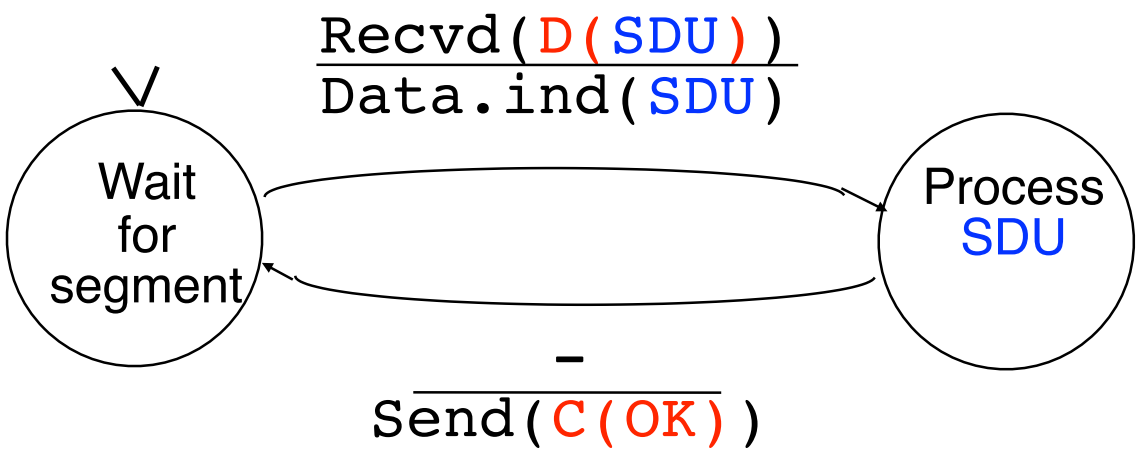
At least one bit in the segment header is used to indicate the type of segment

Protocol 2 (cont.)

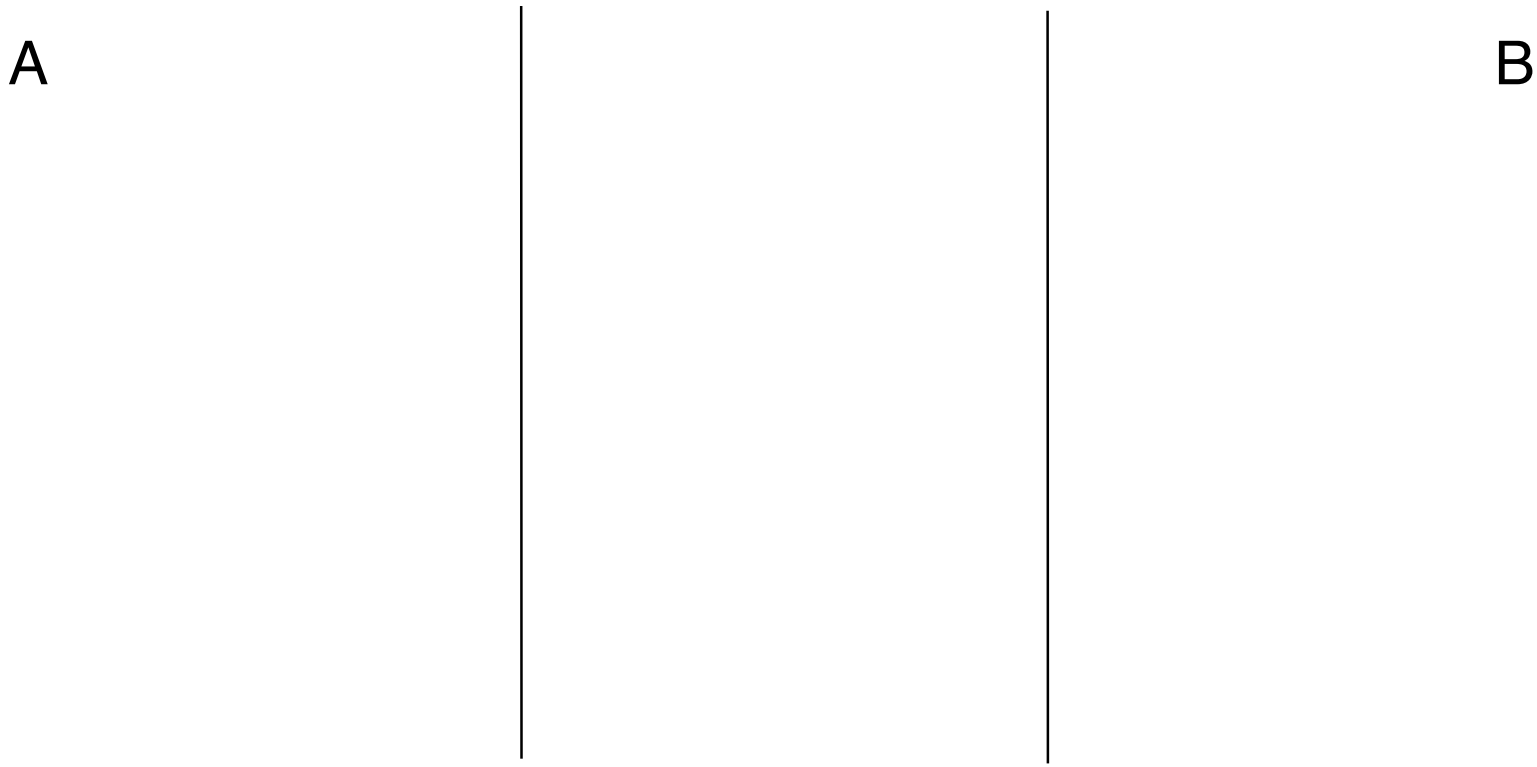
Sender



Receiver

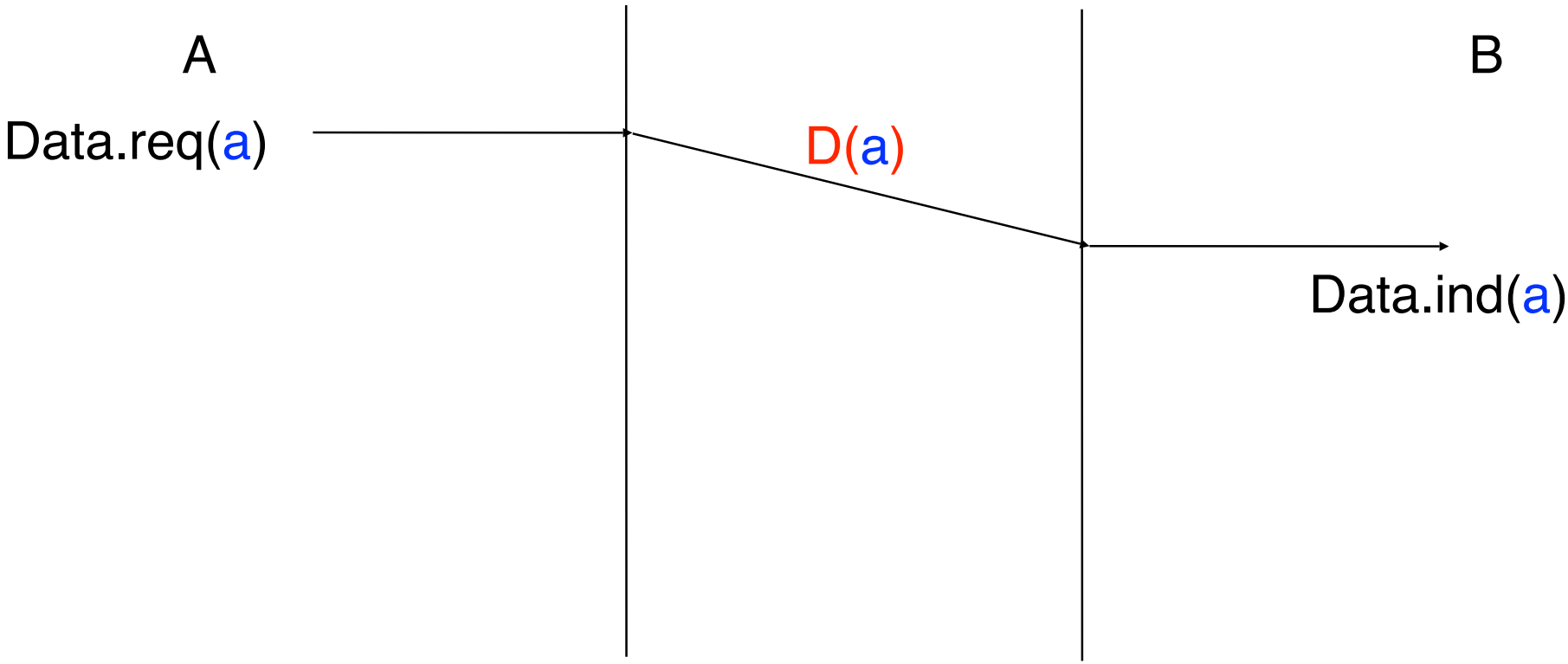


Protocol 2 : Example



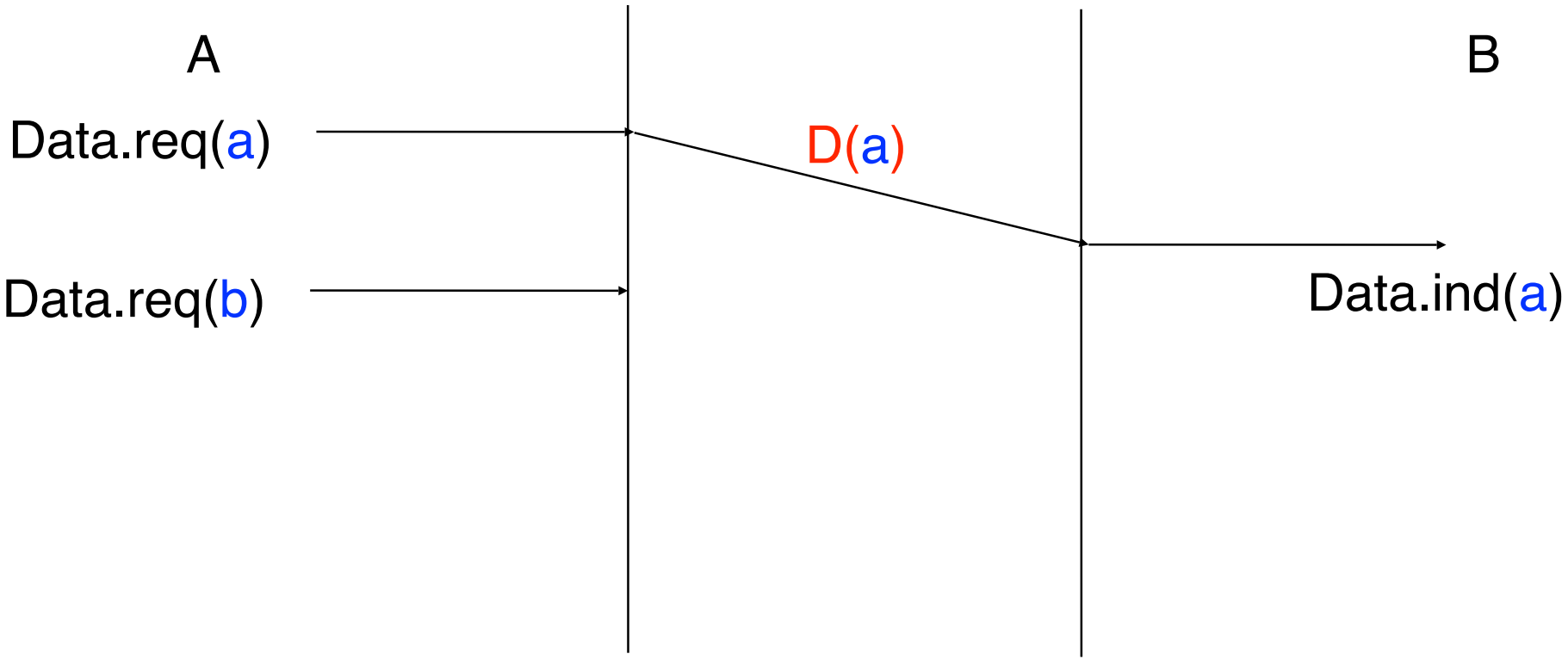
The sender only sends segments when authorised by the receiver

Protocol 2 : Example



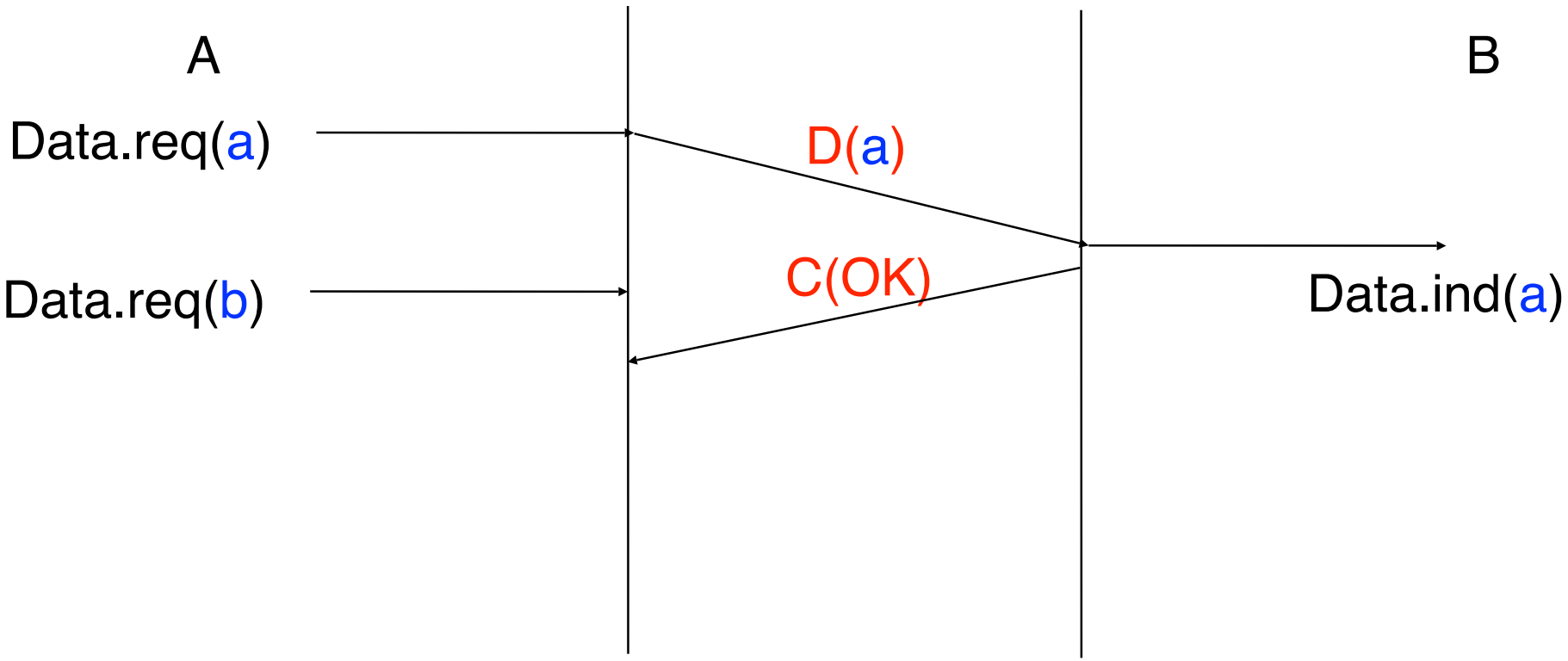
The sender only sends segments when authorised by the receiver

Protocol 2 : Example



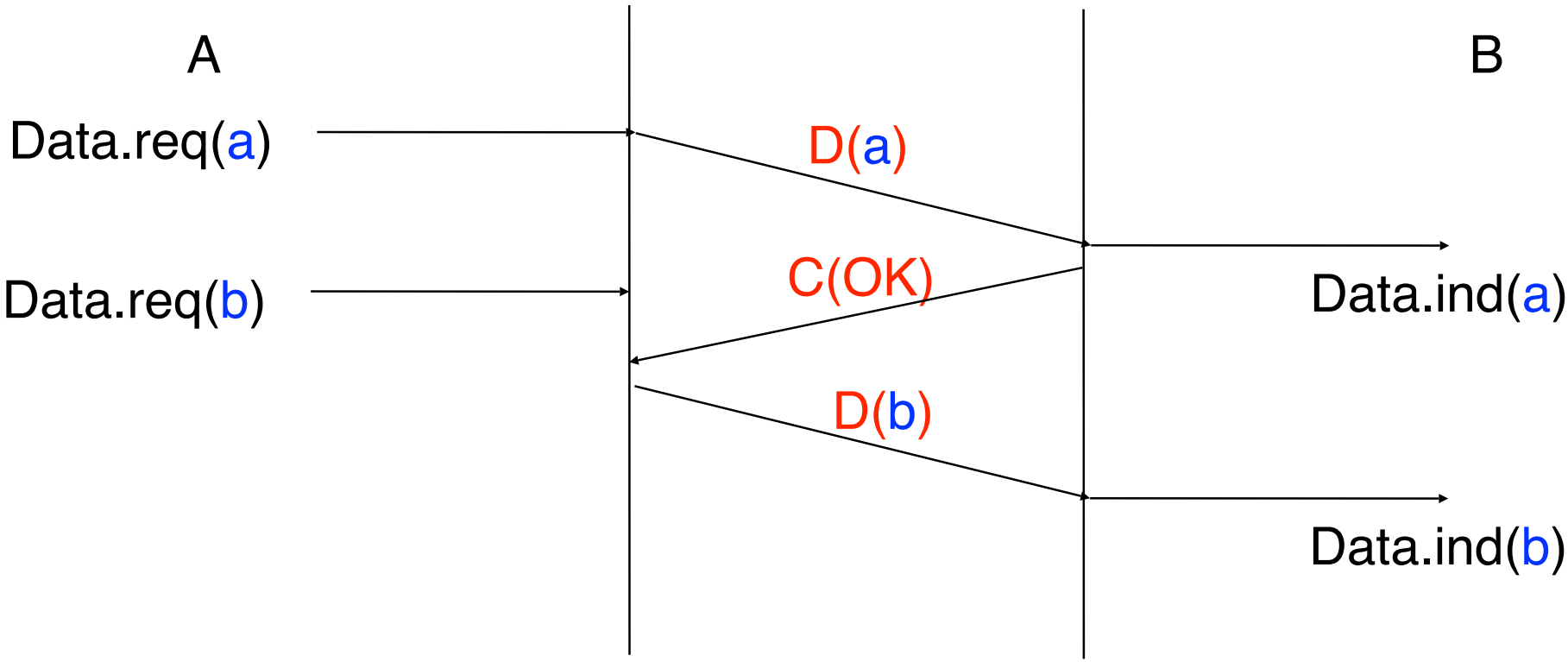
The sender only sends segments when authorised by the receiver

Protocol 2 : Example



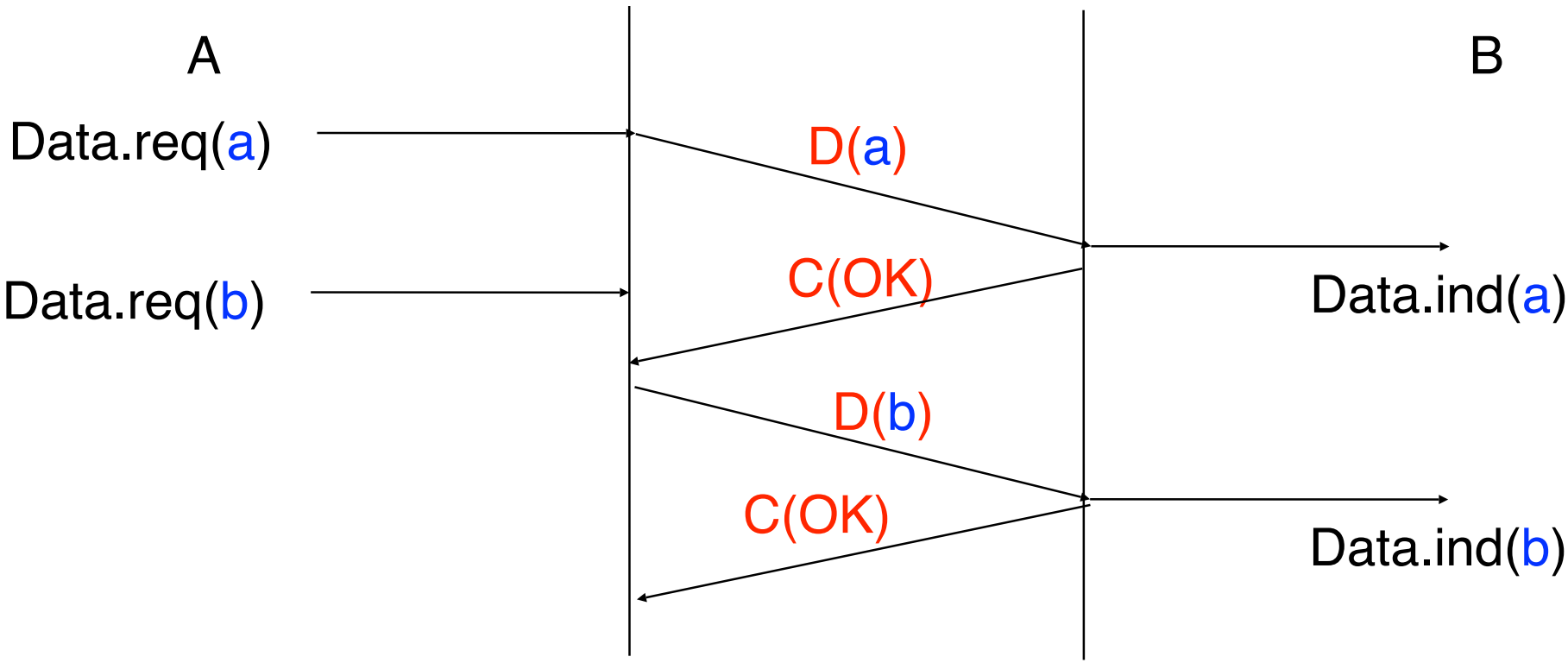
The sender only sends segments when authorised by the receiver

Protocol 2 : Example



The sender only sends segments when authorised by the receiver

Protocol 2 : Example



The sender only sends segments when authorised by the receiver

Protocol 3

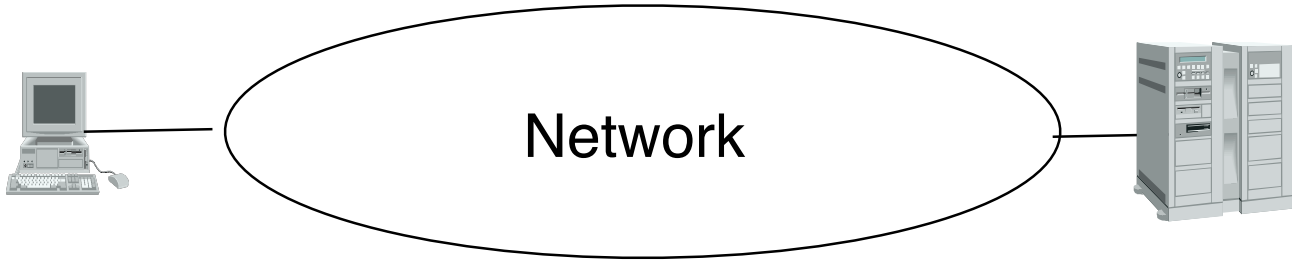
How can we provide a reliable service in the transport layer

Hypotheses

1. The application sends small SDUs
2. The network layer provides a perfect service
   1. Transmission errors are possible
   2. No packet is lost
   3. There is no packet reordering
   4. There are no duplications of packets
3. Data transmission is unidirectional

Transmission errors

Which types of transmission errors do we need to consider in the transport layer ?

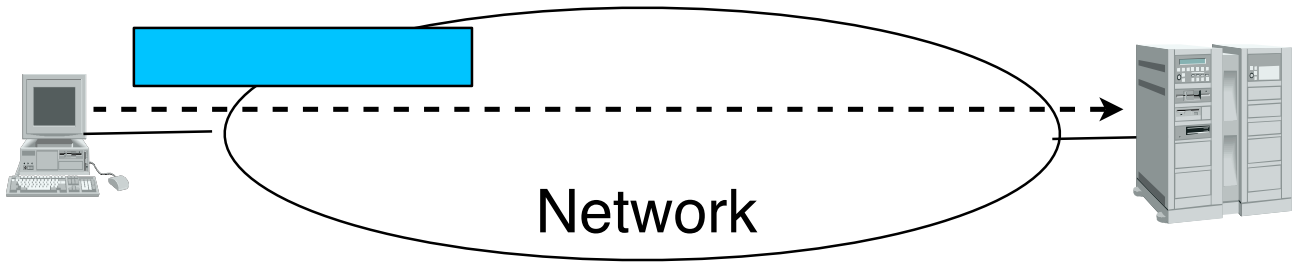


Physical-layer transmission errors caused by nature

- Random isolated error
  - one bit is flipped in the segment
- Random burst error
  - a group of n bits inside the segment is errored
    - most of the bits in the group are flippe

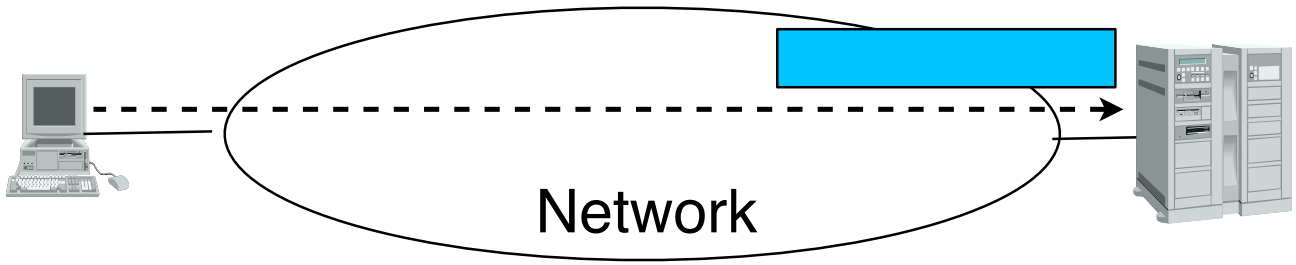
Security issues versus transmission errors

Information sent over a network may become corrupted for other reasons than transmission errors



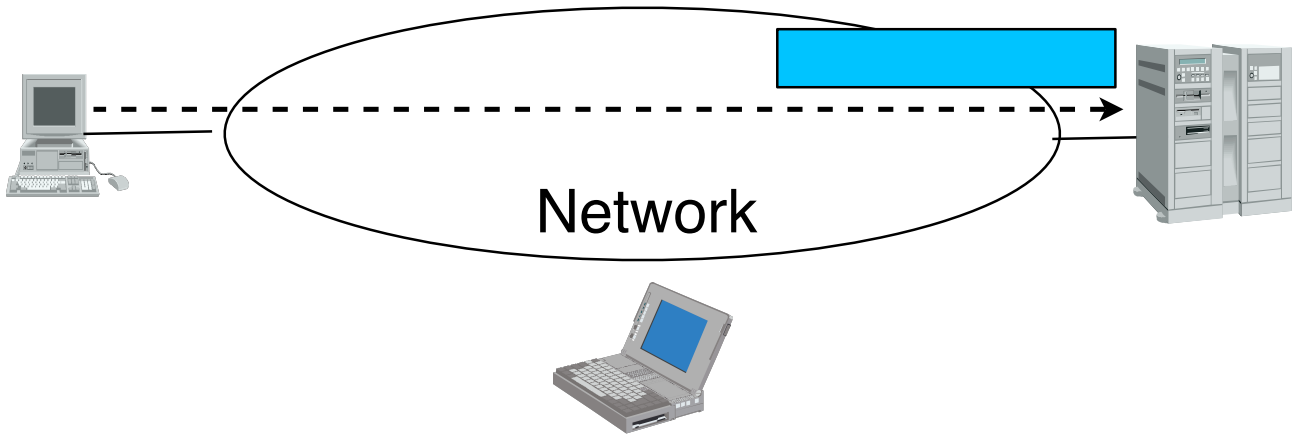
Security issues versus transmission errors

Information sent over a network may become corrupted for other reasons than transmission errors



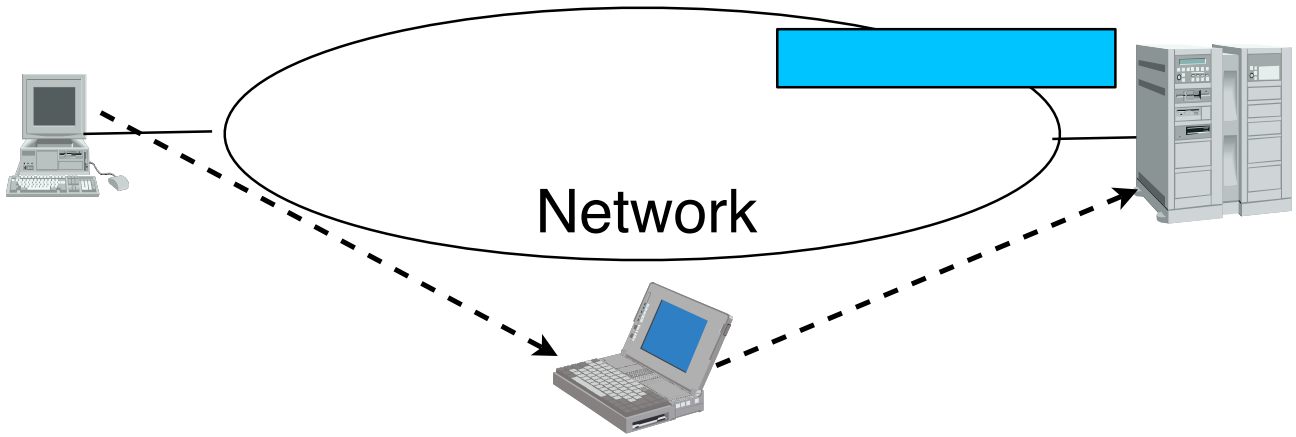
Security issues versus transmission errors

Information sent over a network may become corrupted for other reasons than transmission errors



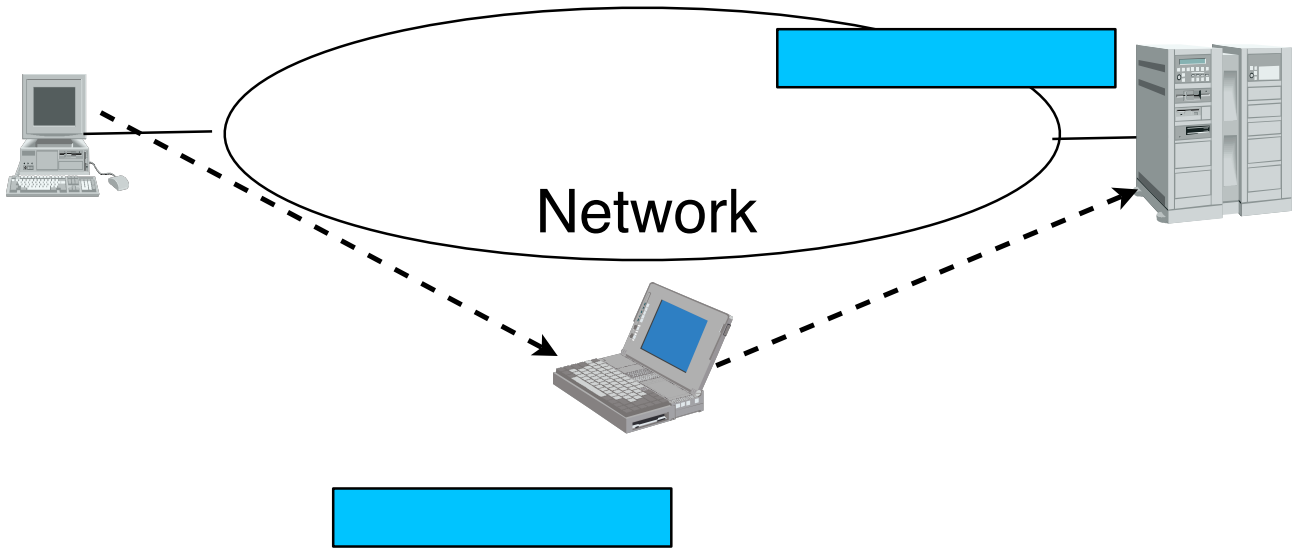
Security issues versus transmission errors

Information sent over a network may become corrupted for other reasons than transmission errors



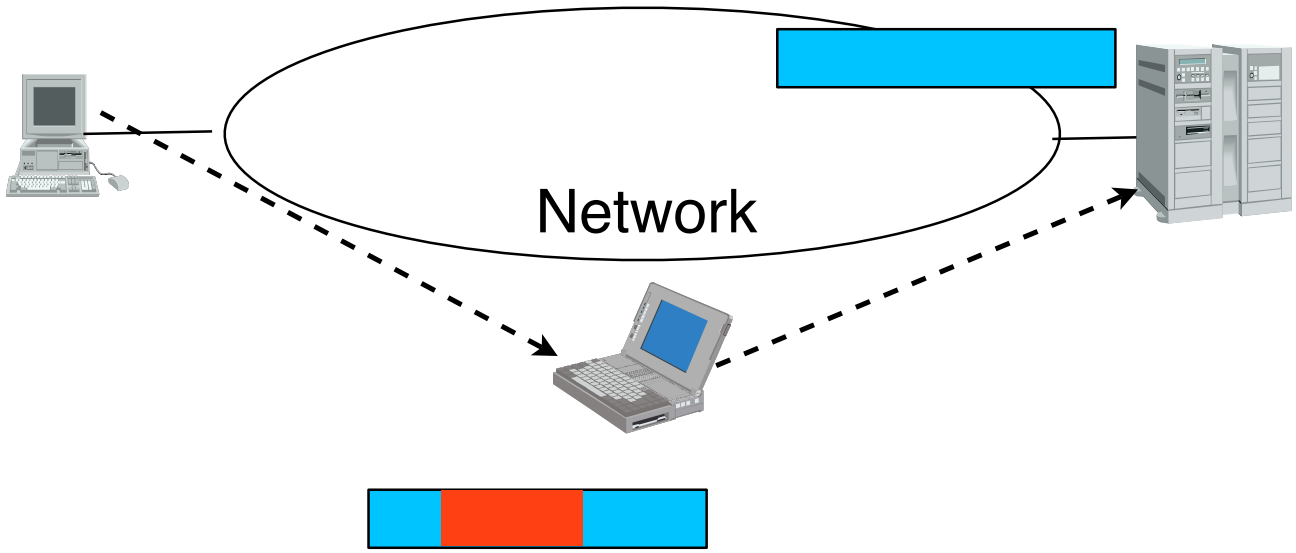
Security issues versus transmission errors

Information sent over a network may become corrupted for other reasons than transmission errors



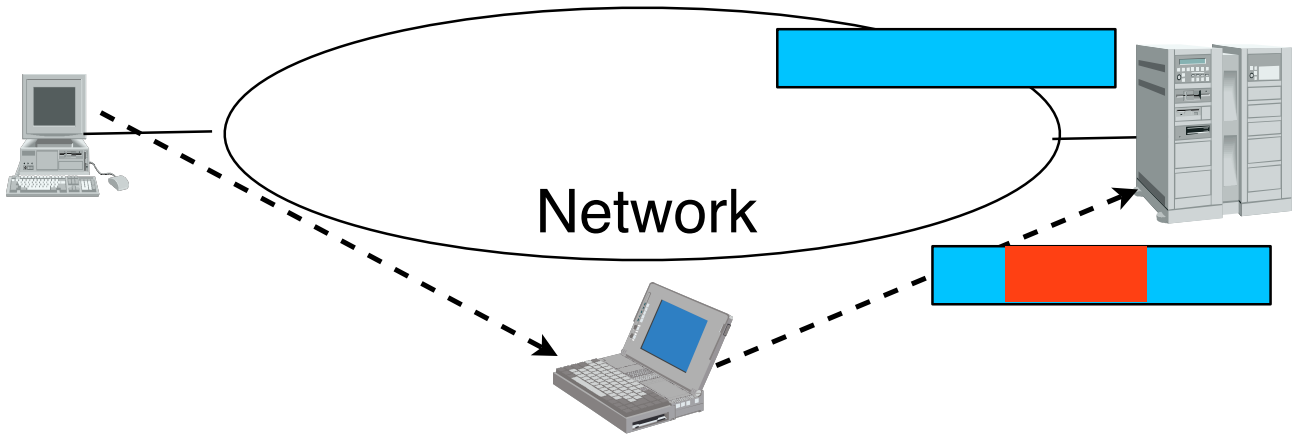
Security issues versus transmission errors

Information sent over a network may become corrupted for other reasons than transmission errors



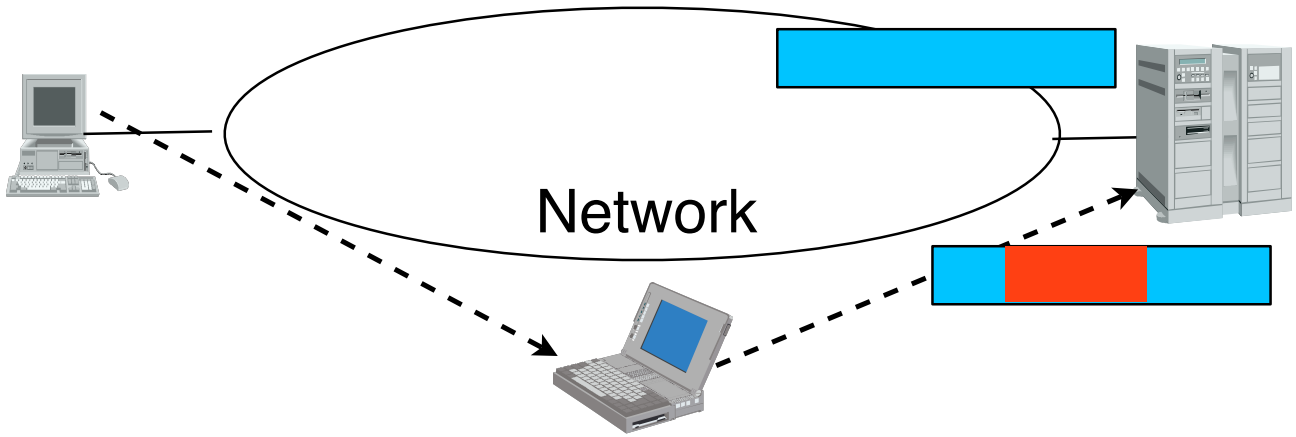
Security issues versus transmission errors

Information sent over a network may become corrupted for other reasons than transmission errors



Security issues versus transmission errors

Information sent over a network may become corrupted for other reasons than transmission errors



These attacks are dealt by using special security protocols and mechanisms outside the transport layer

How to detect transmission errors ?

Principle

Sender adds some control information inside the segment

- control information is computed over the entire segment and placed in the segment header or trailer

Receiver checks that the received control information is correct by recomputing it

Parity bits

Simple solution to detect transmission errors

Used on slow-speed serial lines
- e.g. modems connected to the telephone network

Odd Parity
- For each group of n bits, sender computes the n+1th bit so that the n+1 group contains an odd number of bits set to 1

Examples

0011010 1101100

Even Parity

Parity bits

Simple solution to detect transmission errors

Used on slow-speed serial lines
e.g. modems connected to the telephone network

Odd Parity
For each group of n bits, sender computes the n+1th bit so that the n+1 group contains an odd number of bits set to 1

Examples

0011010 0 1101100

Even Parity

Parity bits

Simple solution to detect transmission errors

Used on slow-speed serial lines
- e.g. modems connected to the telephone network

Odd Parity
- For each group of n bits, sender computes the n+1th bit so that the n+1 group contains an odd number of bits set to 1

Examples

0011010 0

11011001

Even Parity

Internet checksum

Motivation

- Internet protocols are implemented in software and we would like to have efficient algorithms to detect transmission errors that are easy to implement

Solution

- Internet checksum
  - Sender computes for each segment and over the entire segment the 1s complement of the sum of all the 16 bits words in the segment

    Receiver recomputes the checksum of each received segment and verifies that it is correct. Otherwise, the segment is discarded

Cyclical Redundancy Check (CRC)

Principle

Improve the performance of the Internet checksum by using polynomial codes

Sender and receiver agree on *r+1* bits pattern called Generator (G)
Sender adds *r* bits of CRC to a *d* bits data segment such that the *d+r* bits pattern is exactly divisible by G using modulo 2 arithmetic

D * 2^r XOR R = n*G

All computations are done in modulo 2 arithmetic by using XOR

1011 + 0101 = 1110 1001 + 1101 = 0100
1011 - 0101 = 1110 1001 - 1101 = 0100

Detection of transmission errors (2)

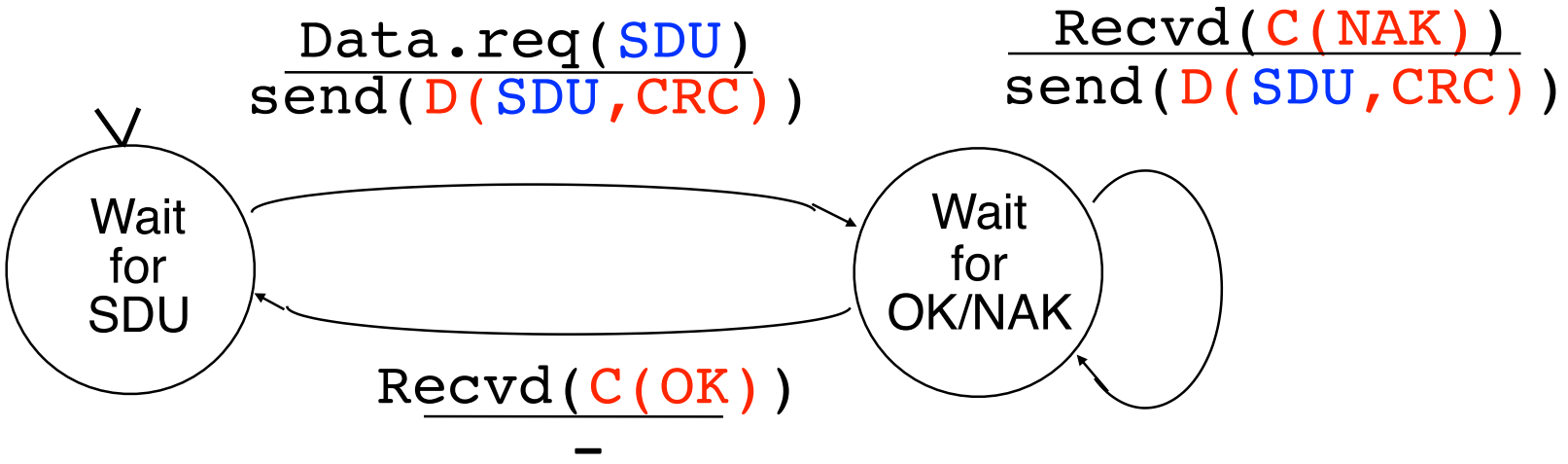
Behaviour of the receiver

If the checksum is correct
- Send an OK control segment to the sender to
  - confirm the reception of the data segment
  - allow the sender to send the next segment

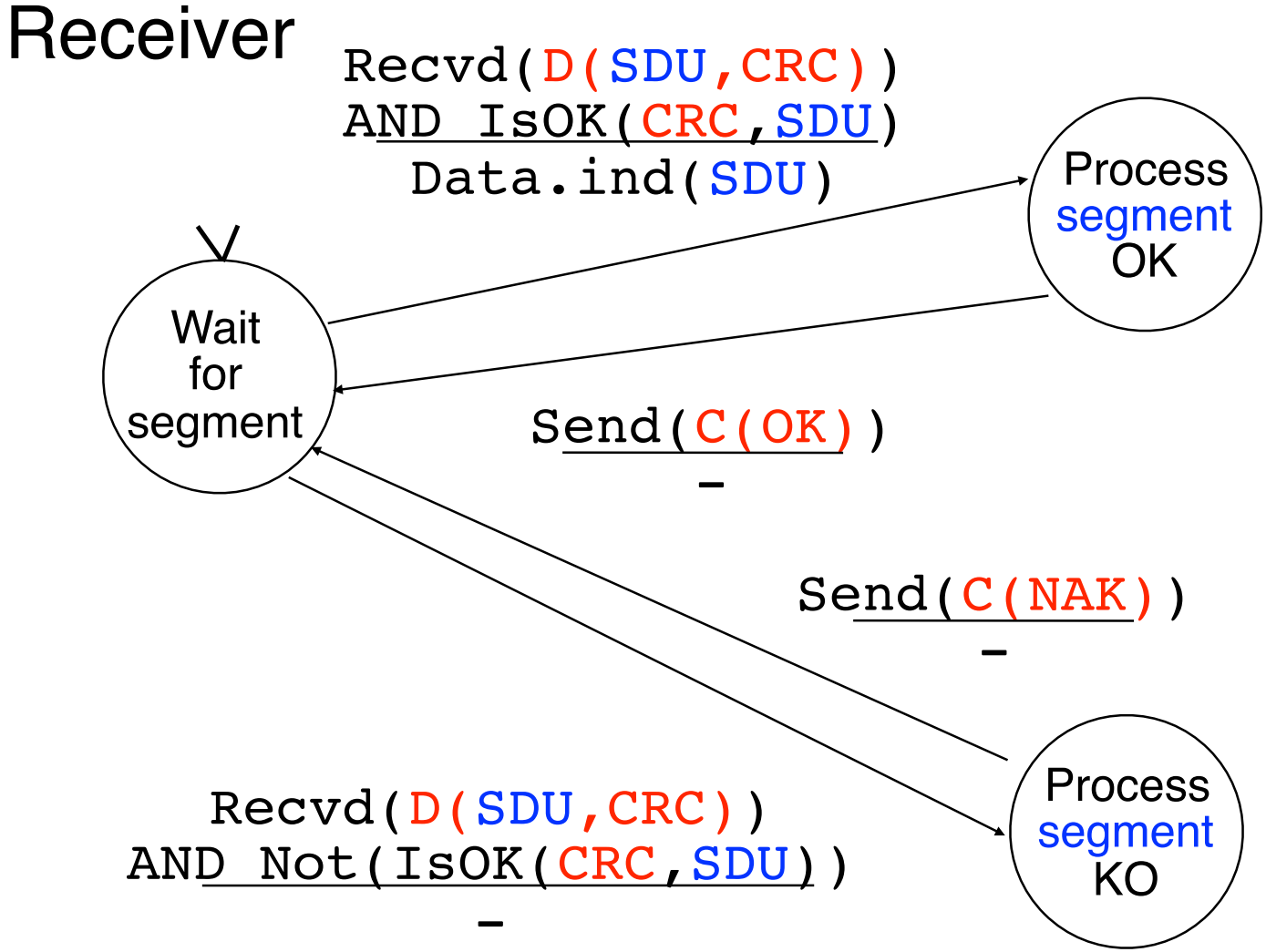
If the checksum is incorrect
- The content of the segment is corrupted and must be discarded
- Send a special control segment (NAK) to the sender to ask it to retransmit the corrupted data segment

Protocol 3a : Sender

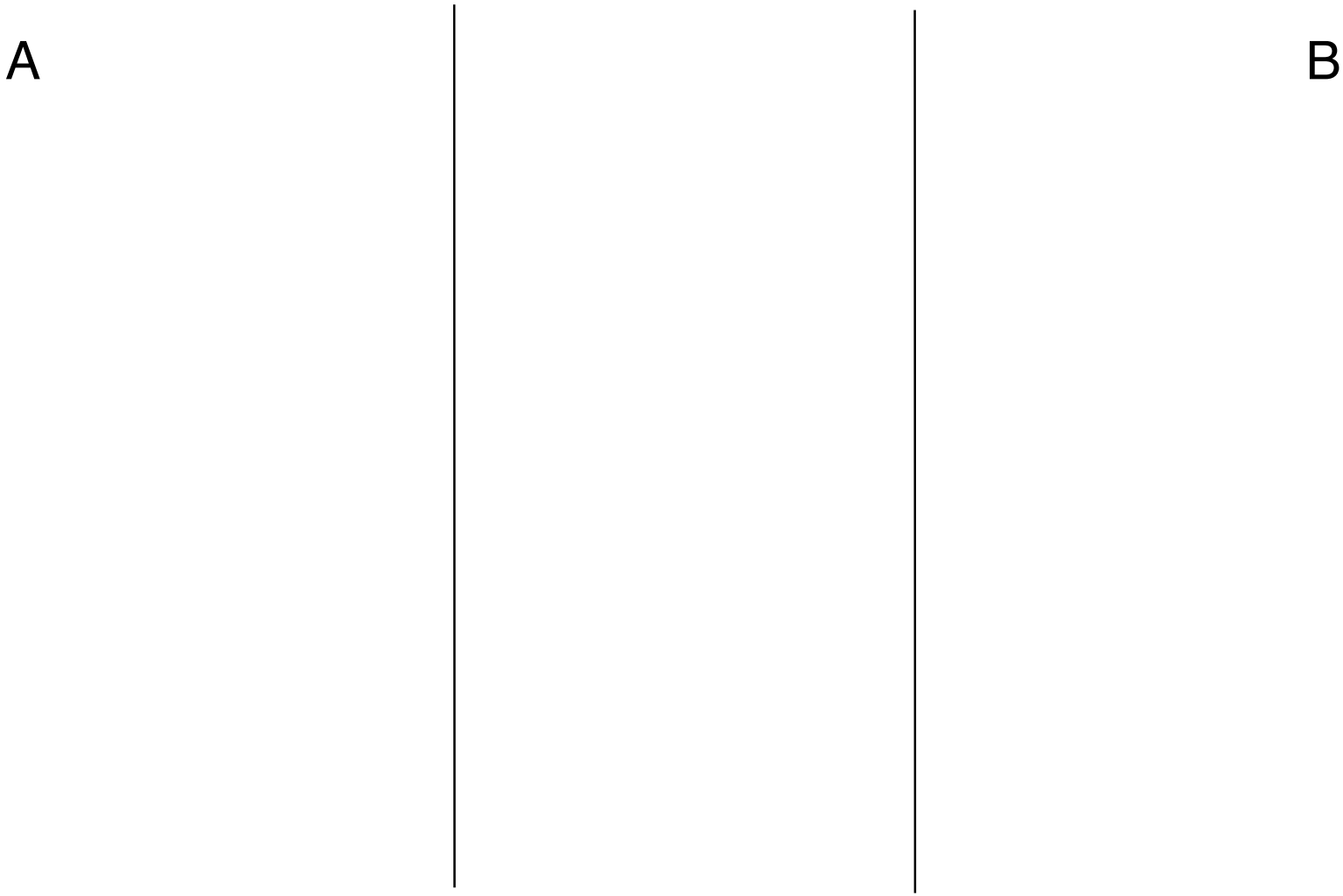
Sender



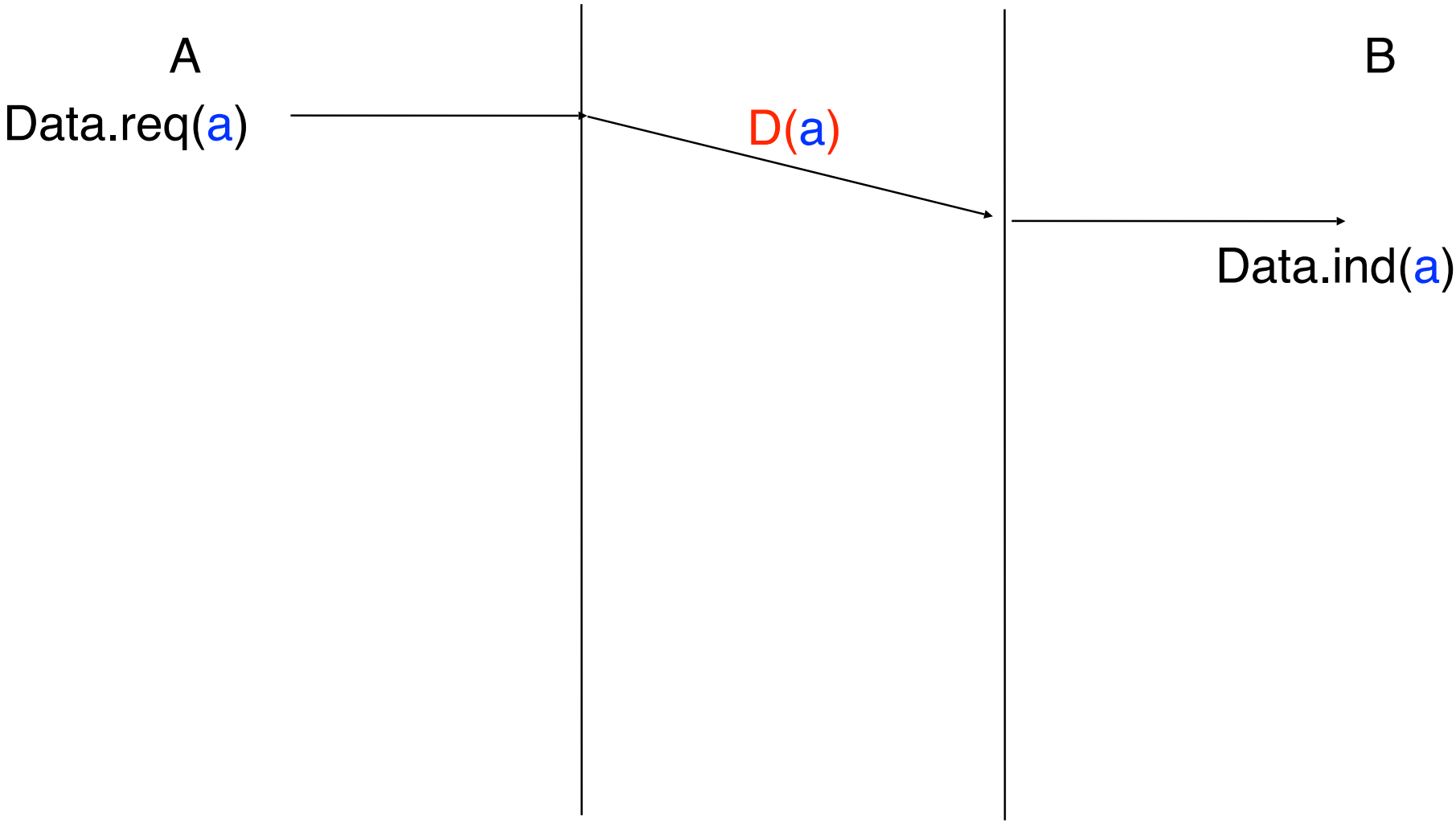
Protocol 3a : Receiver



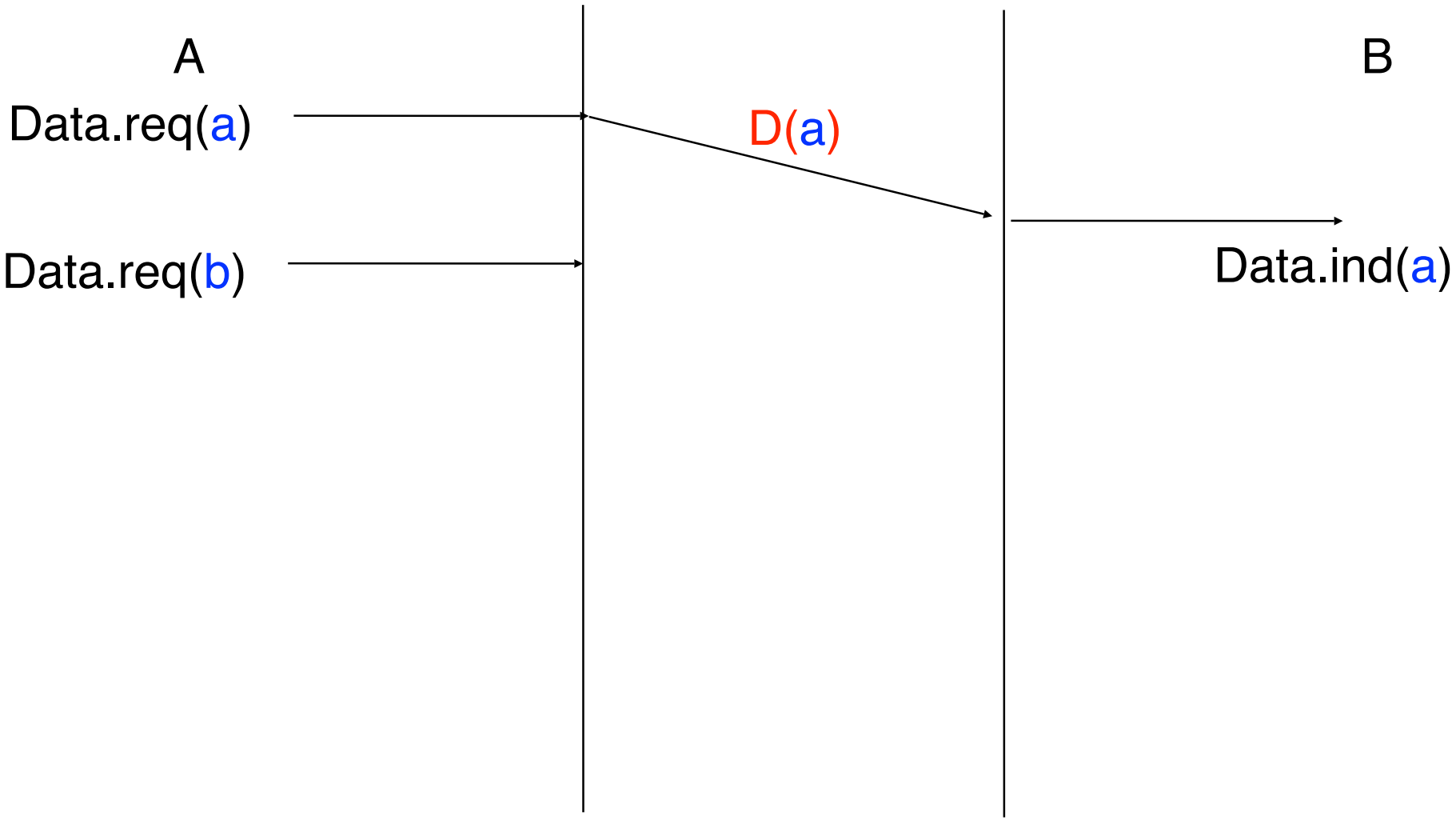
Protocol 3a : Example



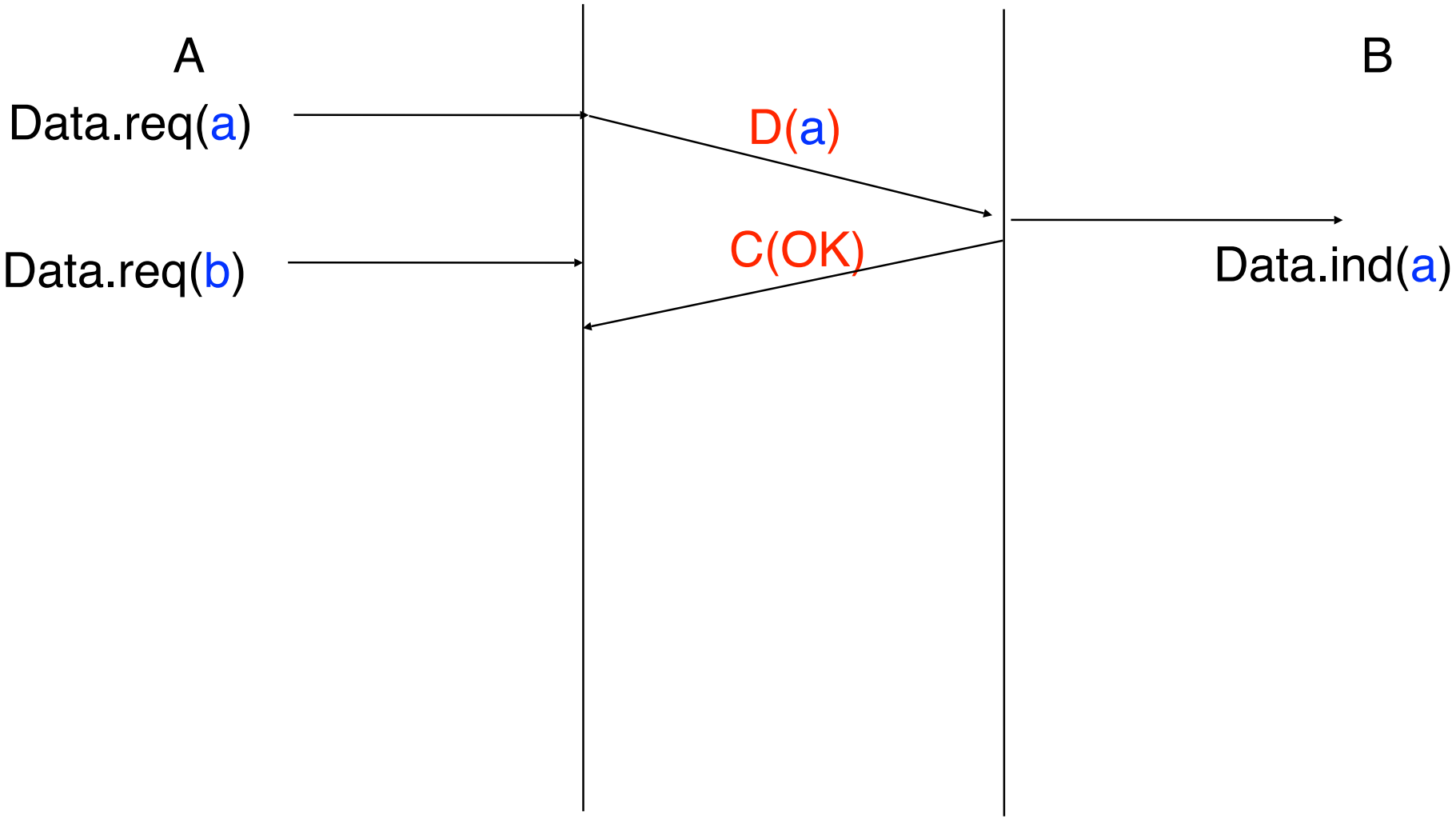
Protocol 3a : Example



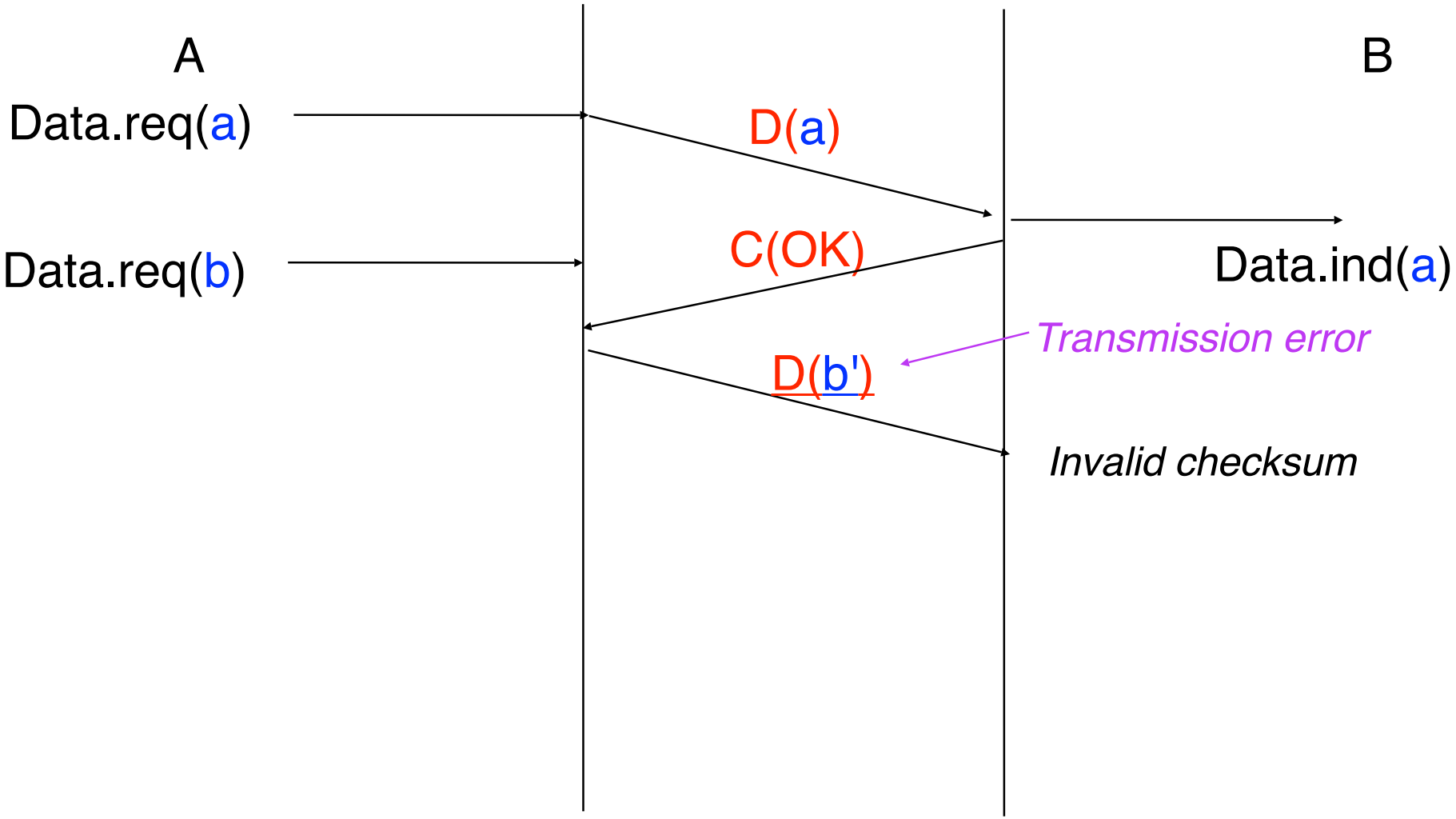
Protocol 3a : Example



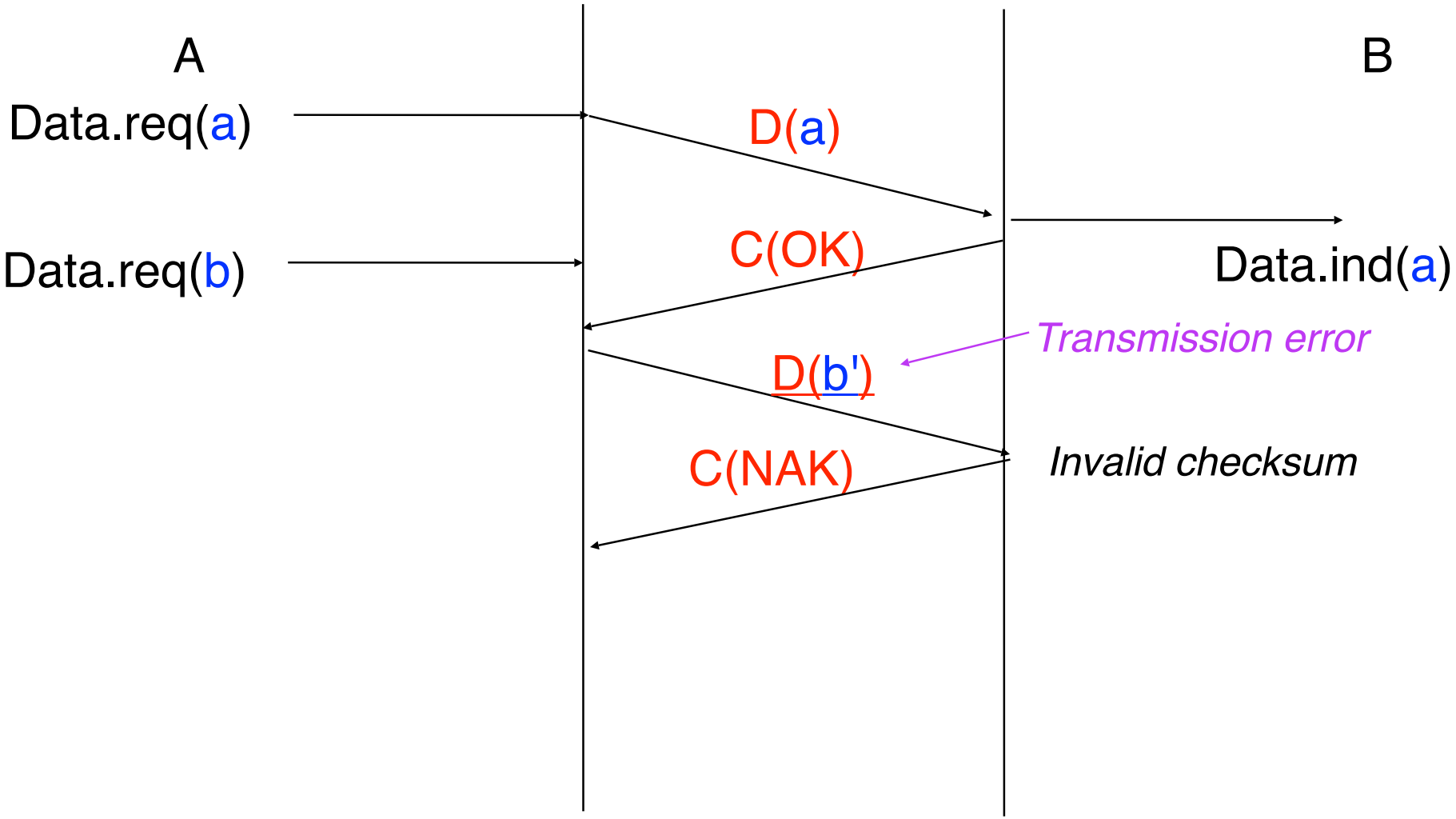
Protocol 3a : Example



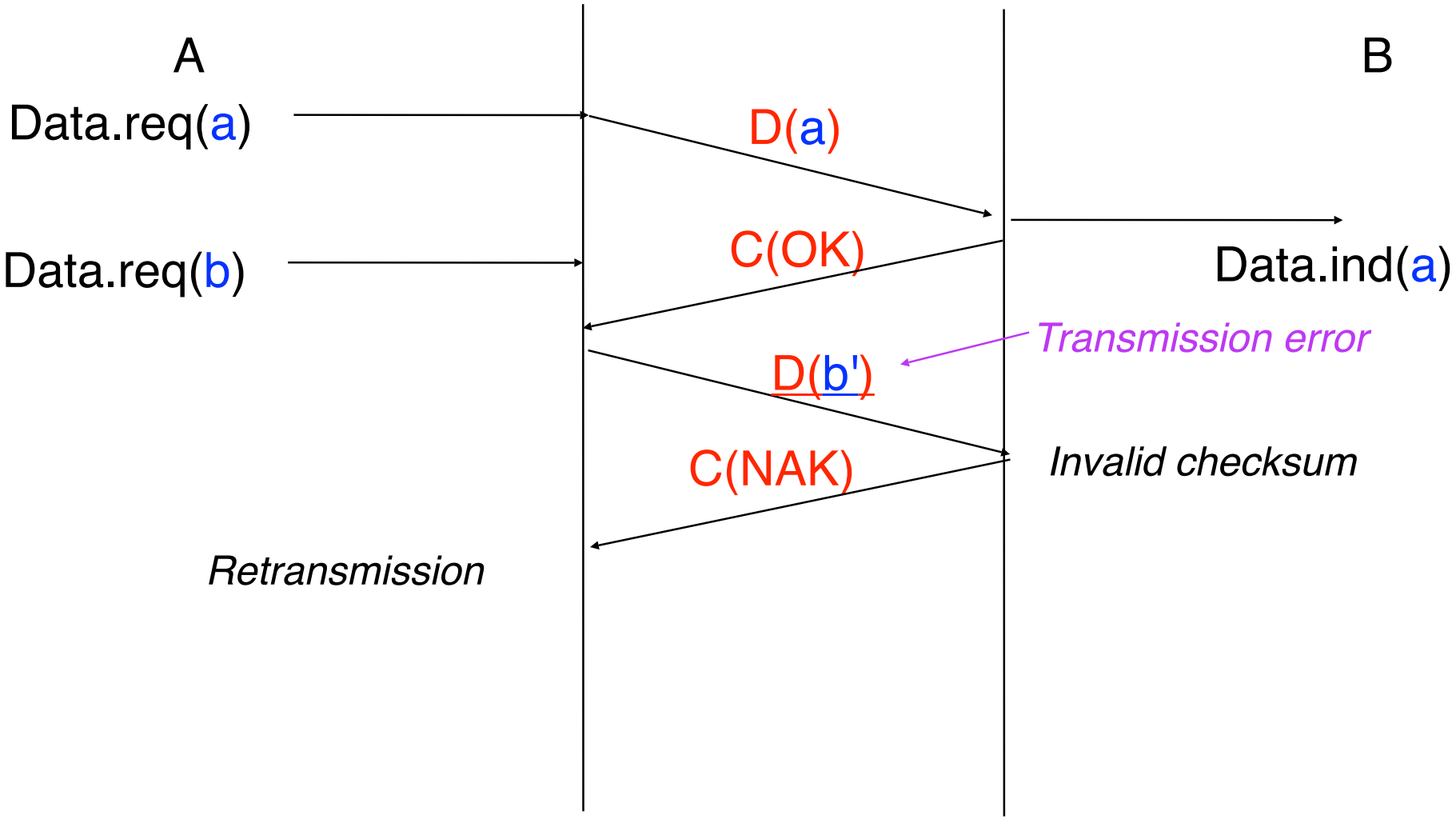
Protocol 3a : Example



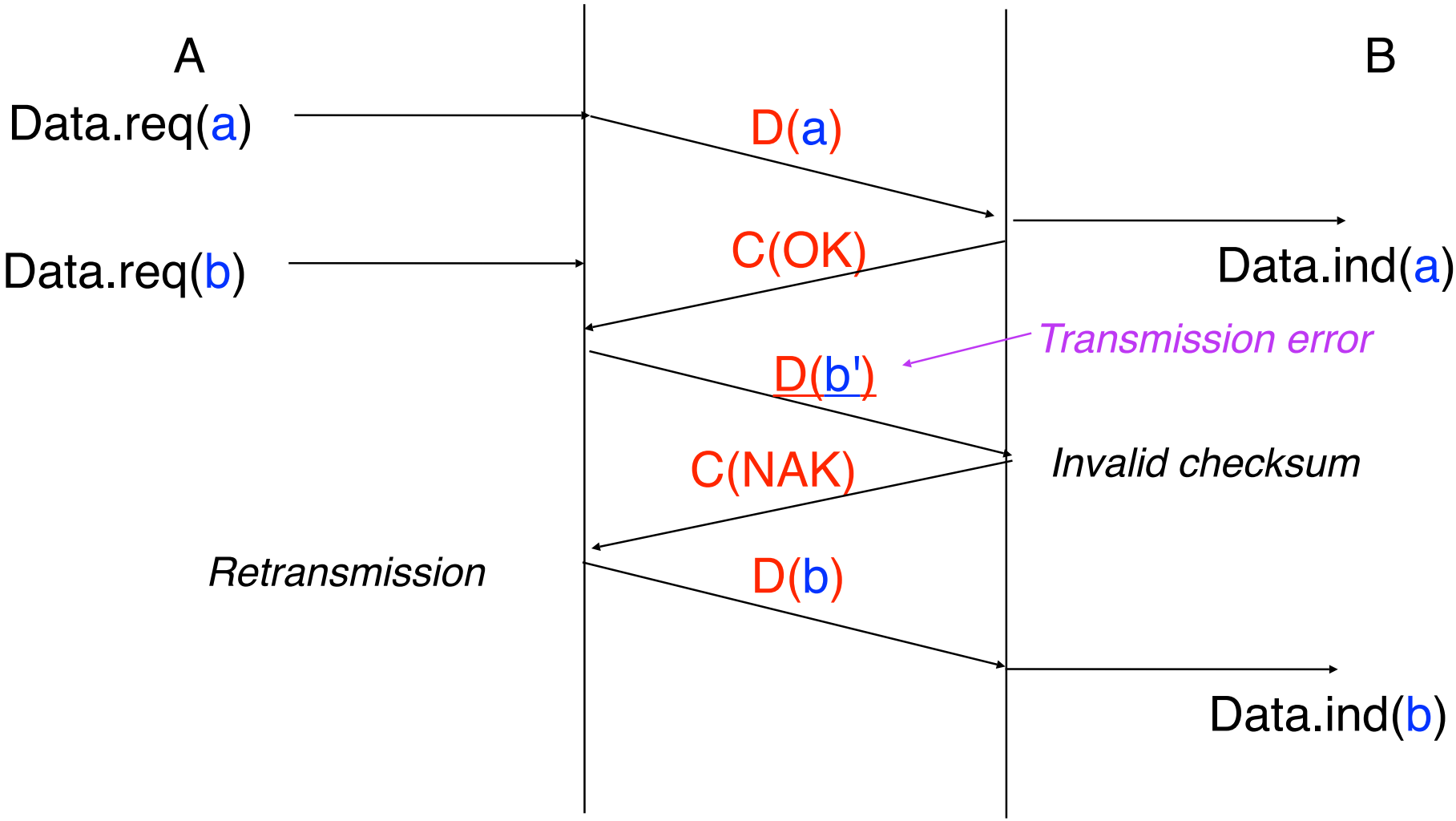
Protocol 3a : Example



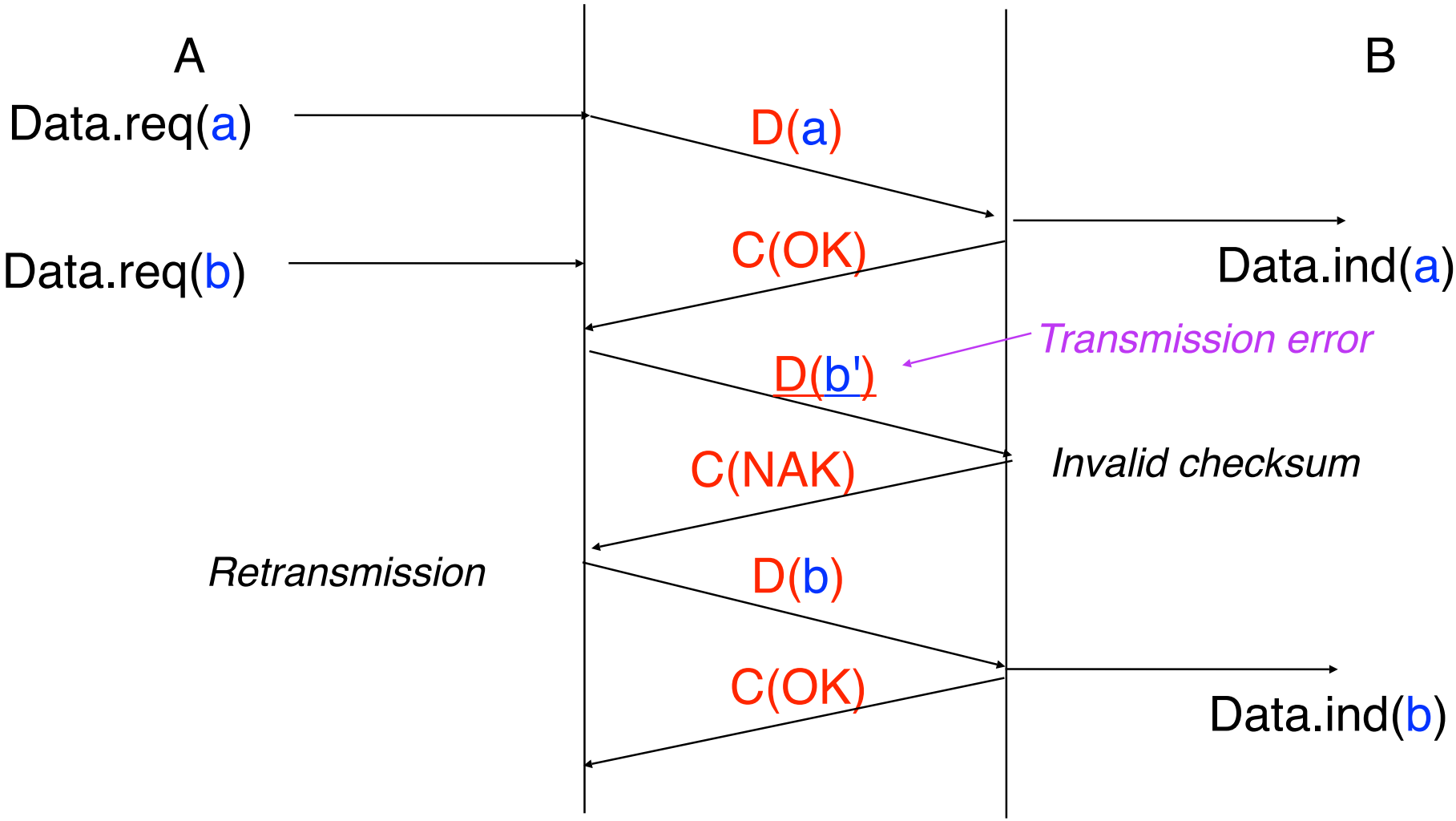
Protocol 3a : Example



Protocol 3a : Example



Protocol 3a : Example



Protocol 3b

How can we provide a reliable service in the transport layer ?

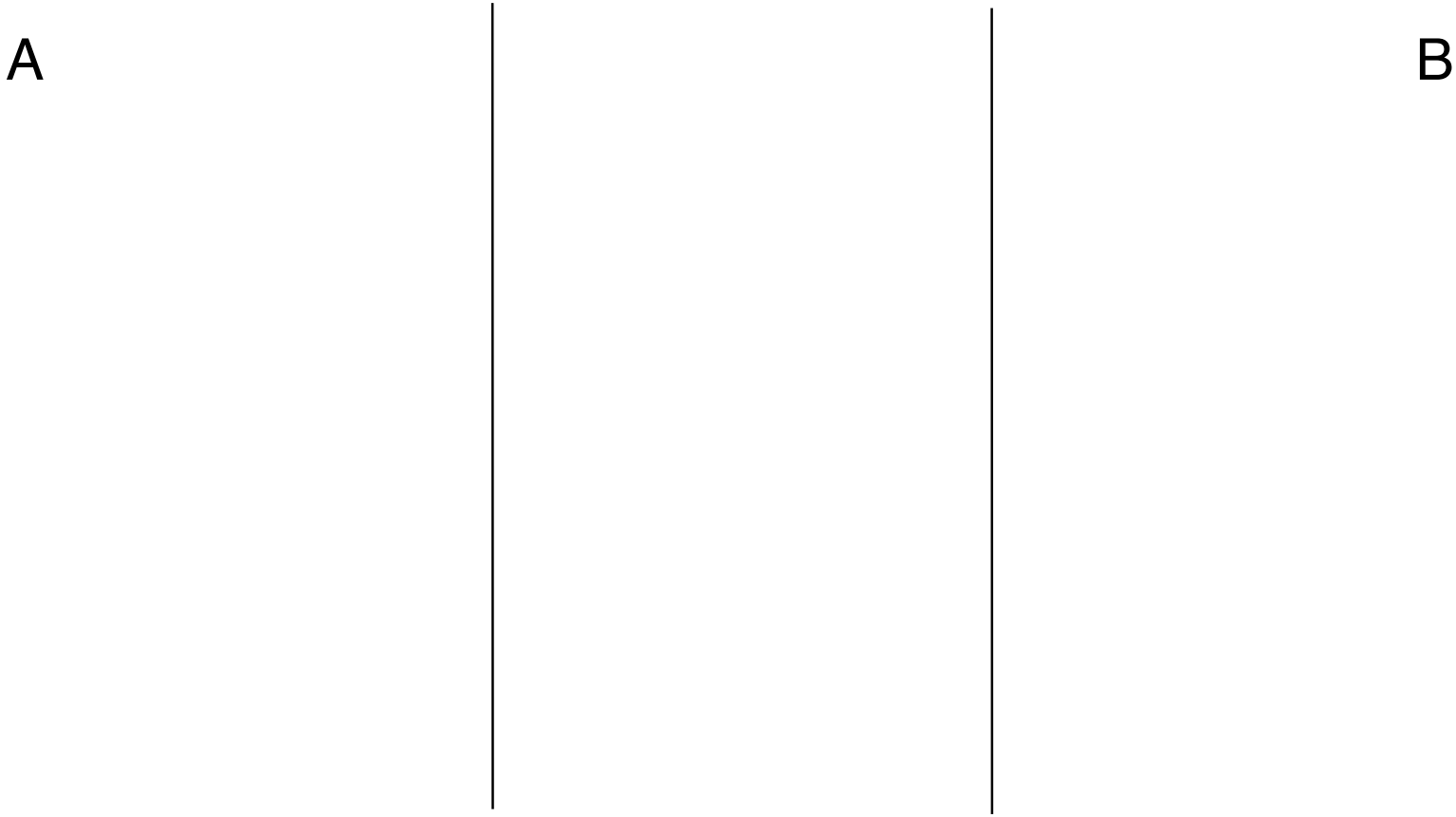
Hypotheses

1. The application sends small SDUs
2. The network layer provides a perfect service
   1. Transmission errors are possible
   2. Packets can be lost
   3. There is no packet reordering
   4. There are no duplications of packets
3. Data transmission is unidirectional

2. How to deal with these problems ?

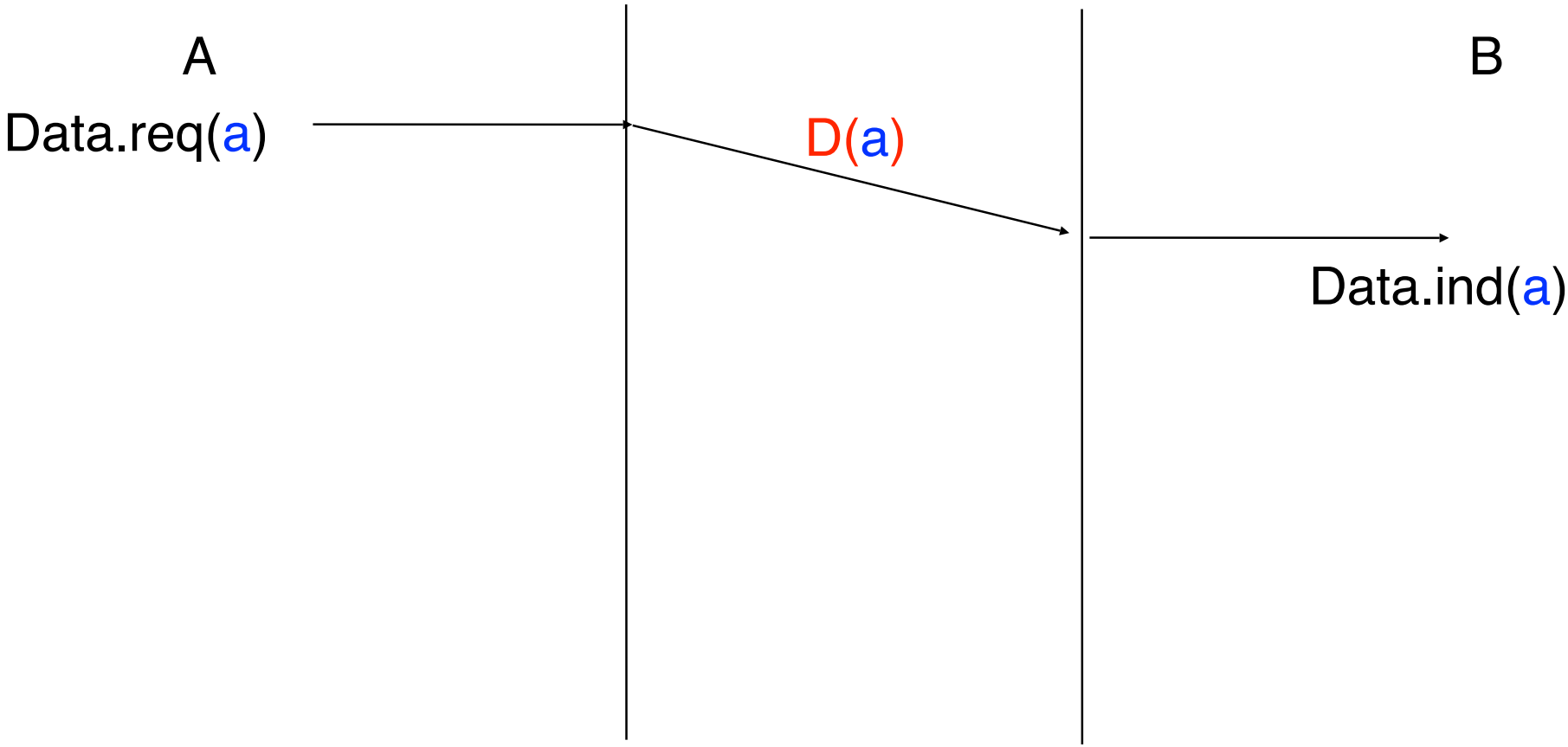
Protocol 3a and segment losses

How do segment losses affect protocol 3a ?



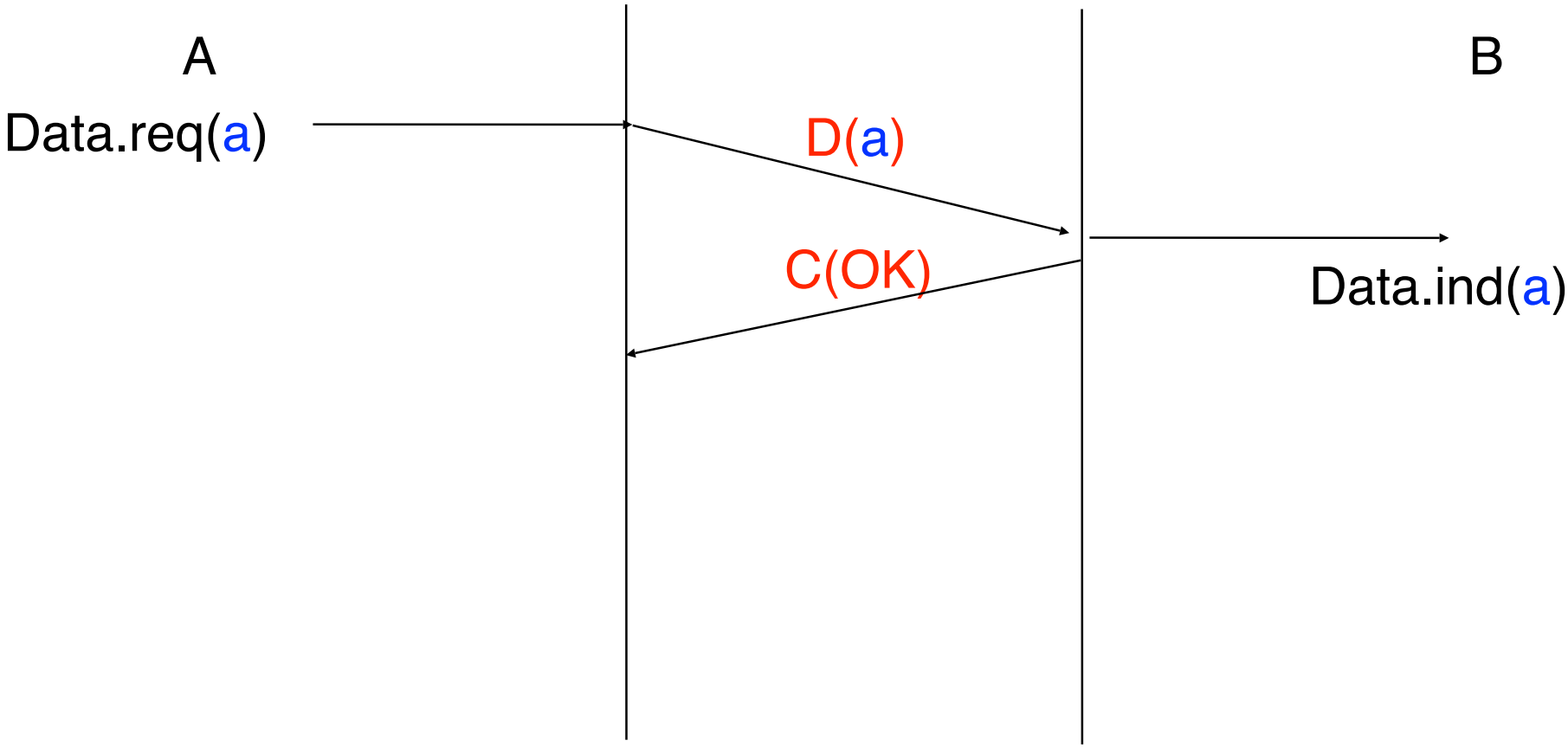
Protocol 3a and segment losses

How do segment losses affect protocol 3a ?



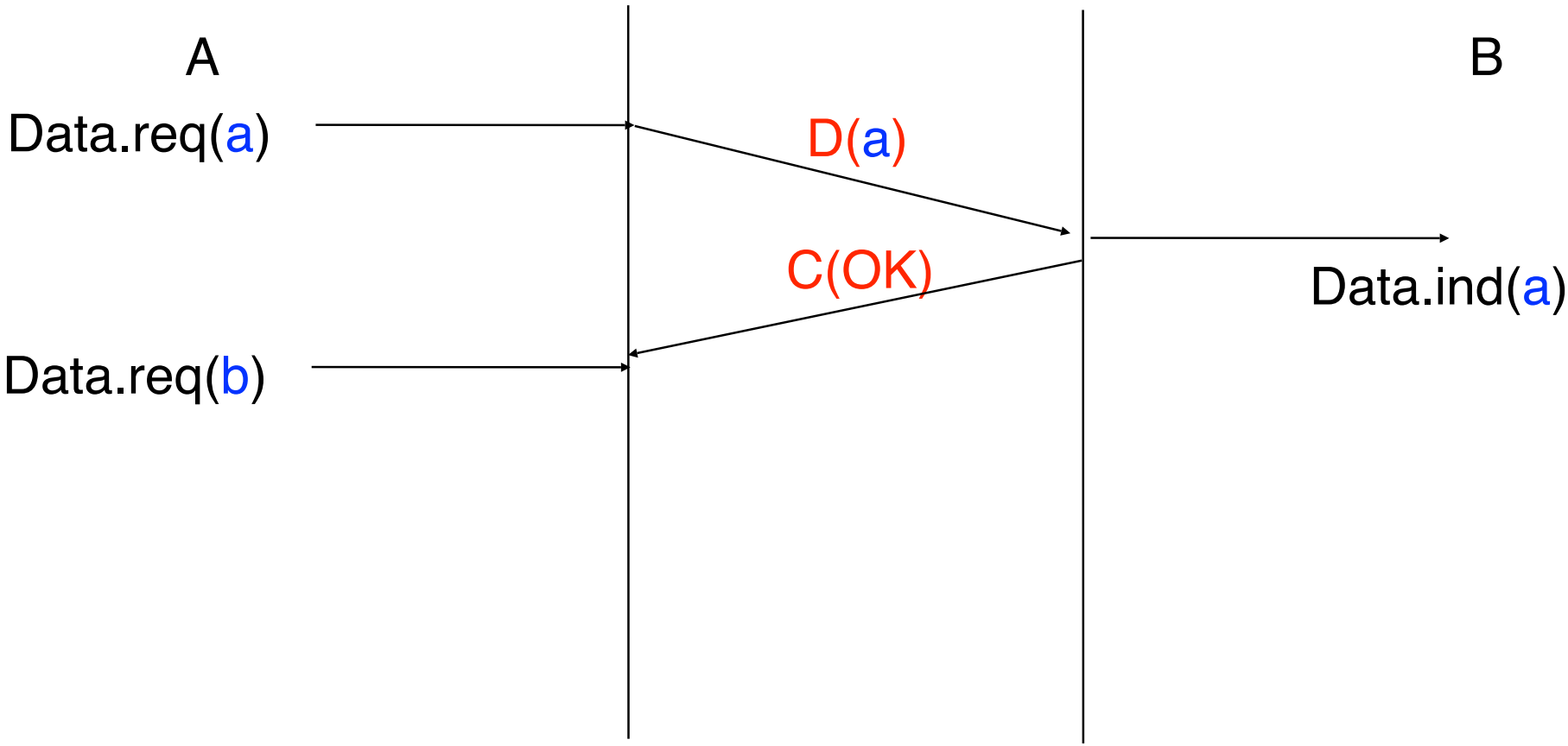
Protocol 3a and segment losses

How do segment losses affect protocol 3a ?



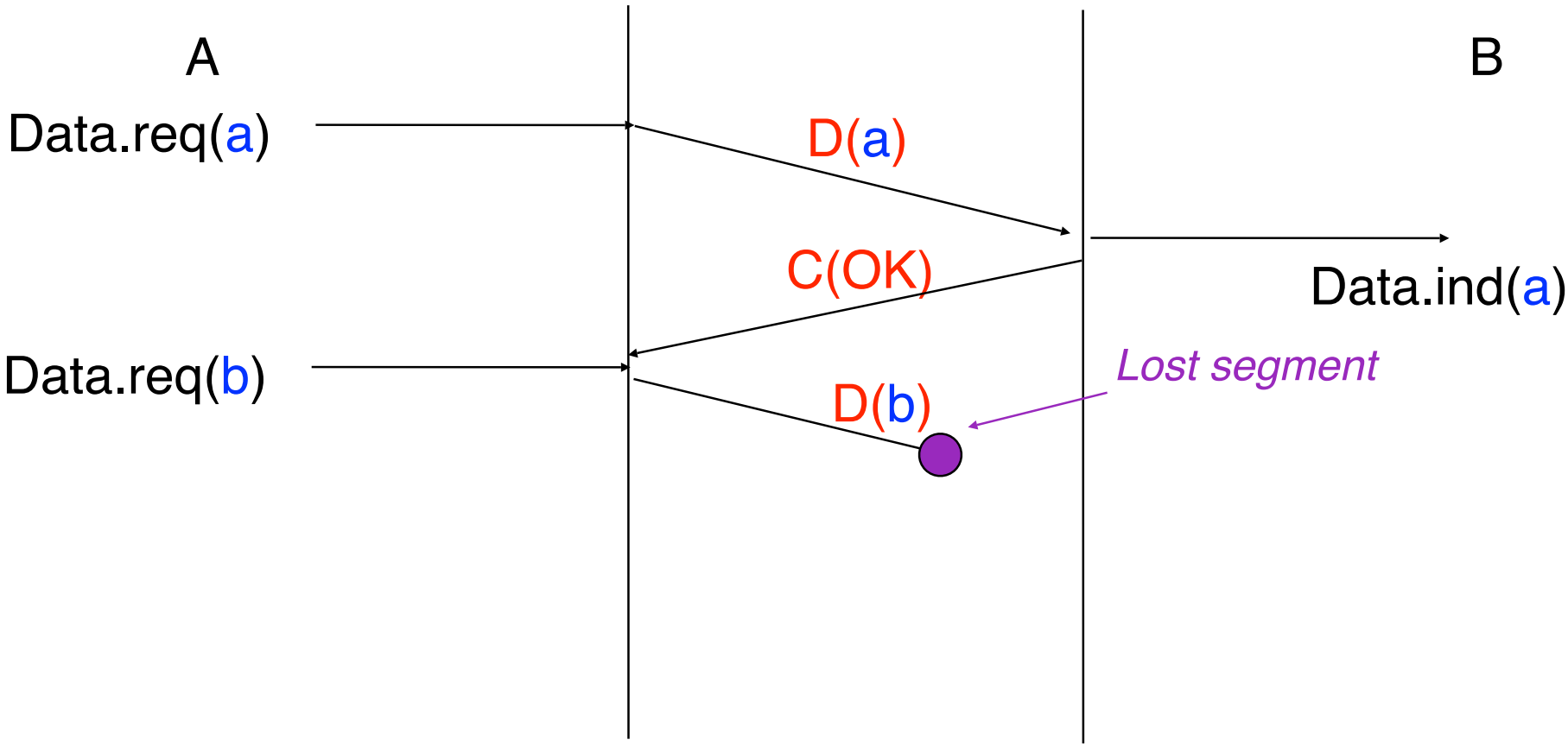
Protocol 3a and segment losses

How do segment losses affect protocol 3a ?



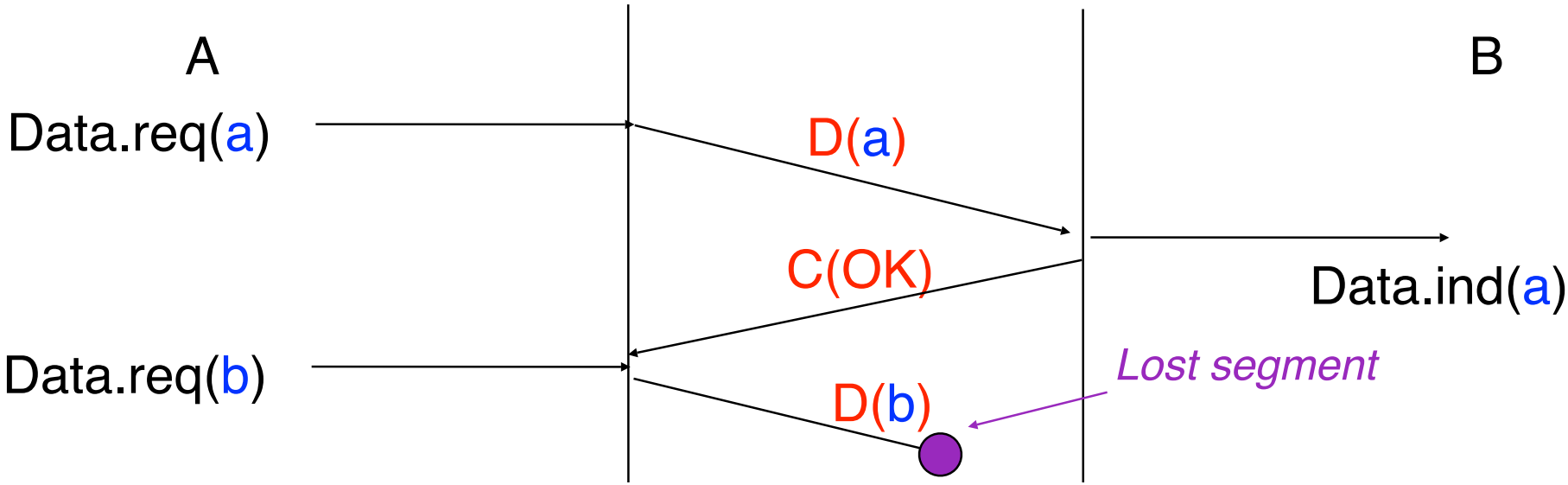
Protocol 3a and segment losses

How do segment losses affect protocol 3a ?



Protocol 3a and segment losses

How do segment losses affect protocol 3a ?



DEADLOCK

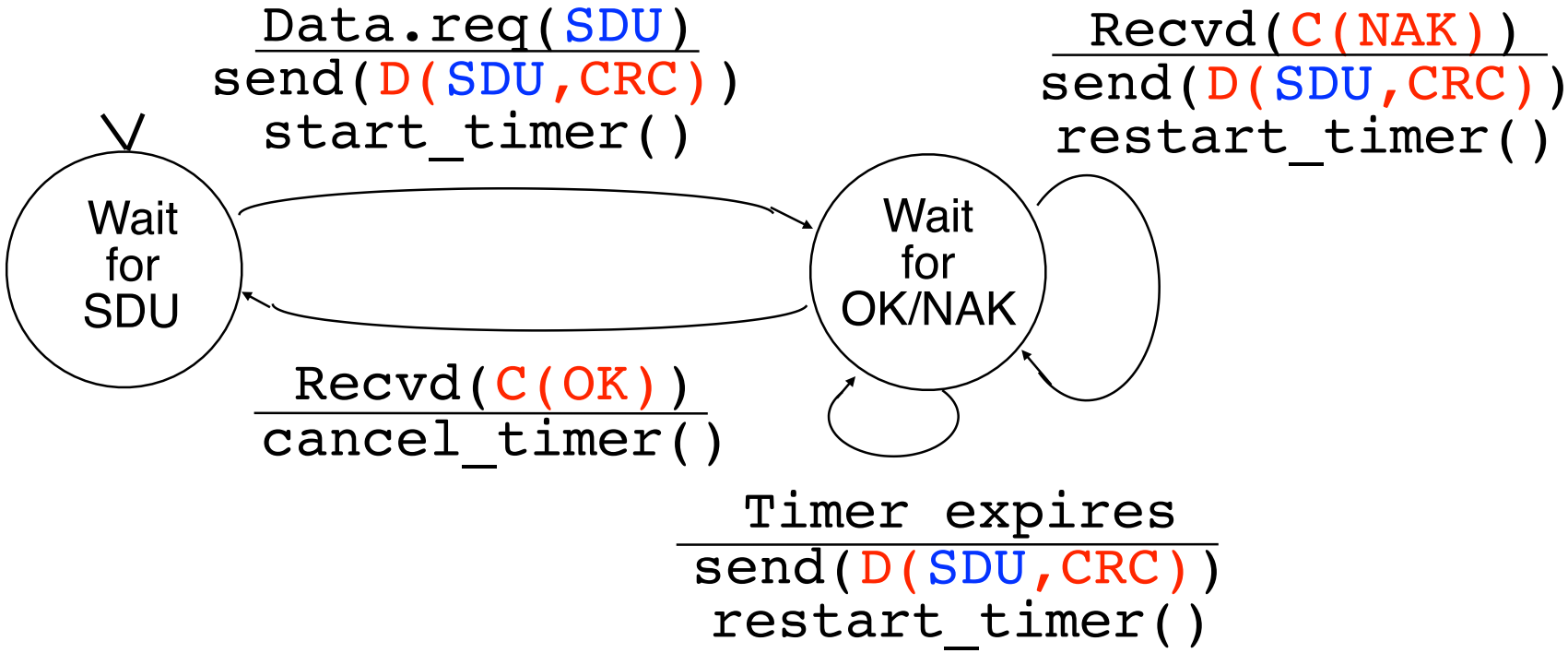
A is waiting for a control segment

B is waiting for a data segment

Protocol 3b

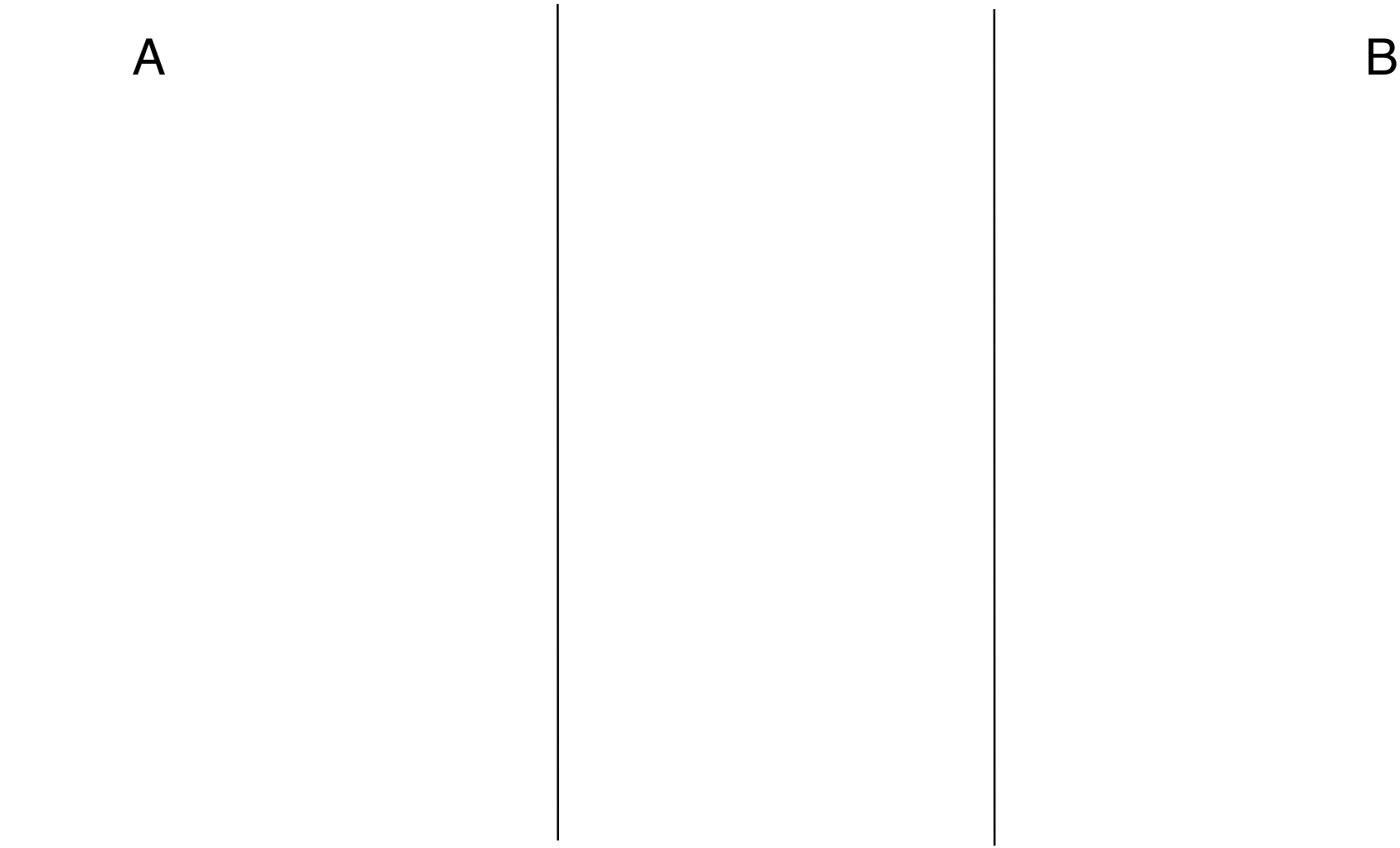
Modification to the sender

Add a retransmission timer to retransmit the lost segment after some time

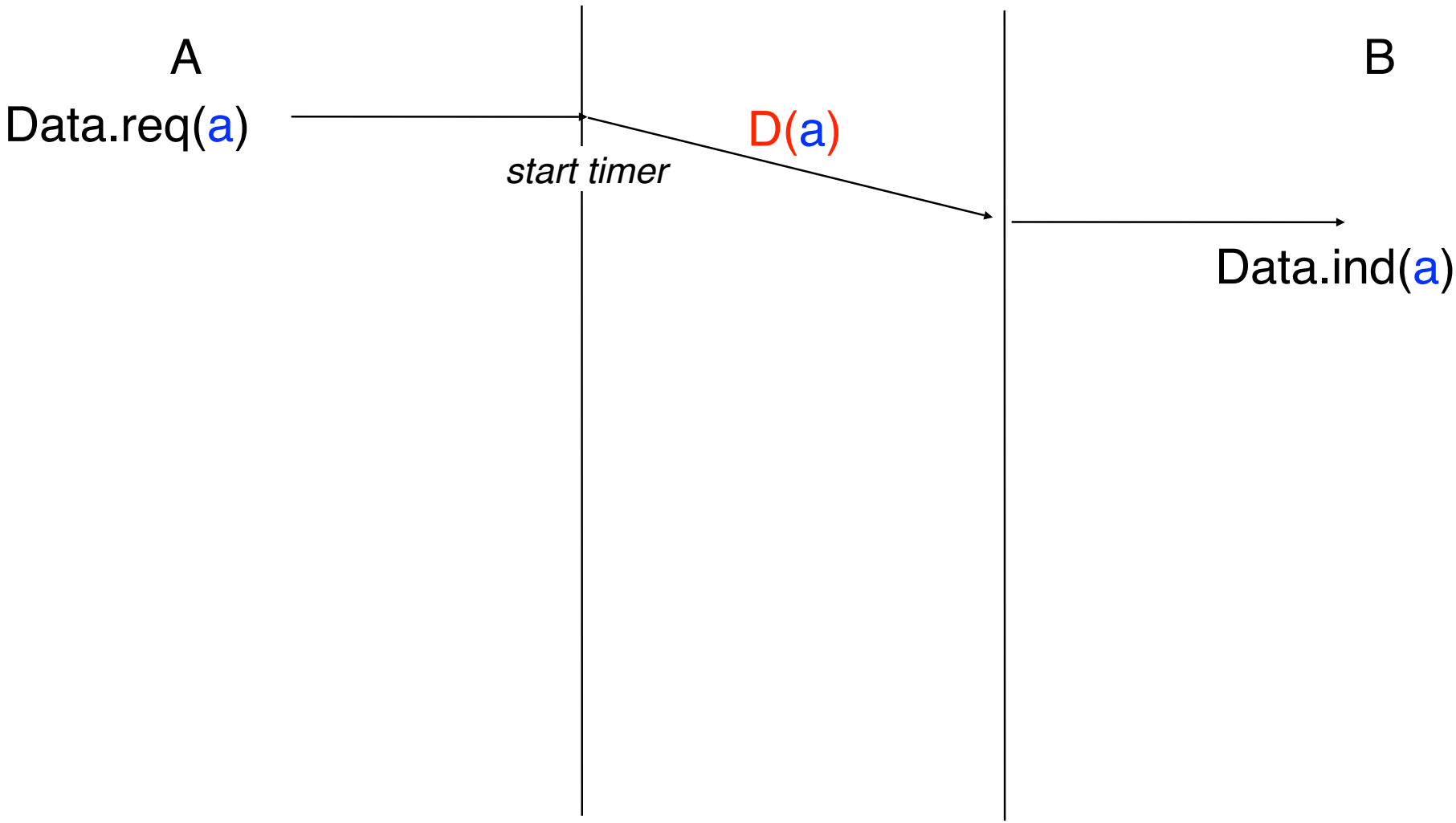


No modification to the receiver

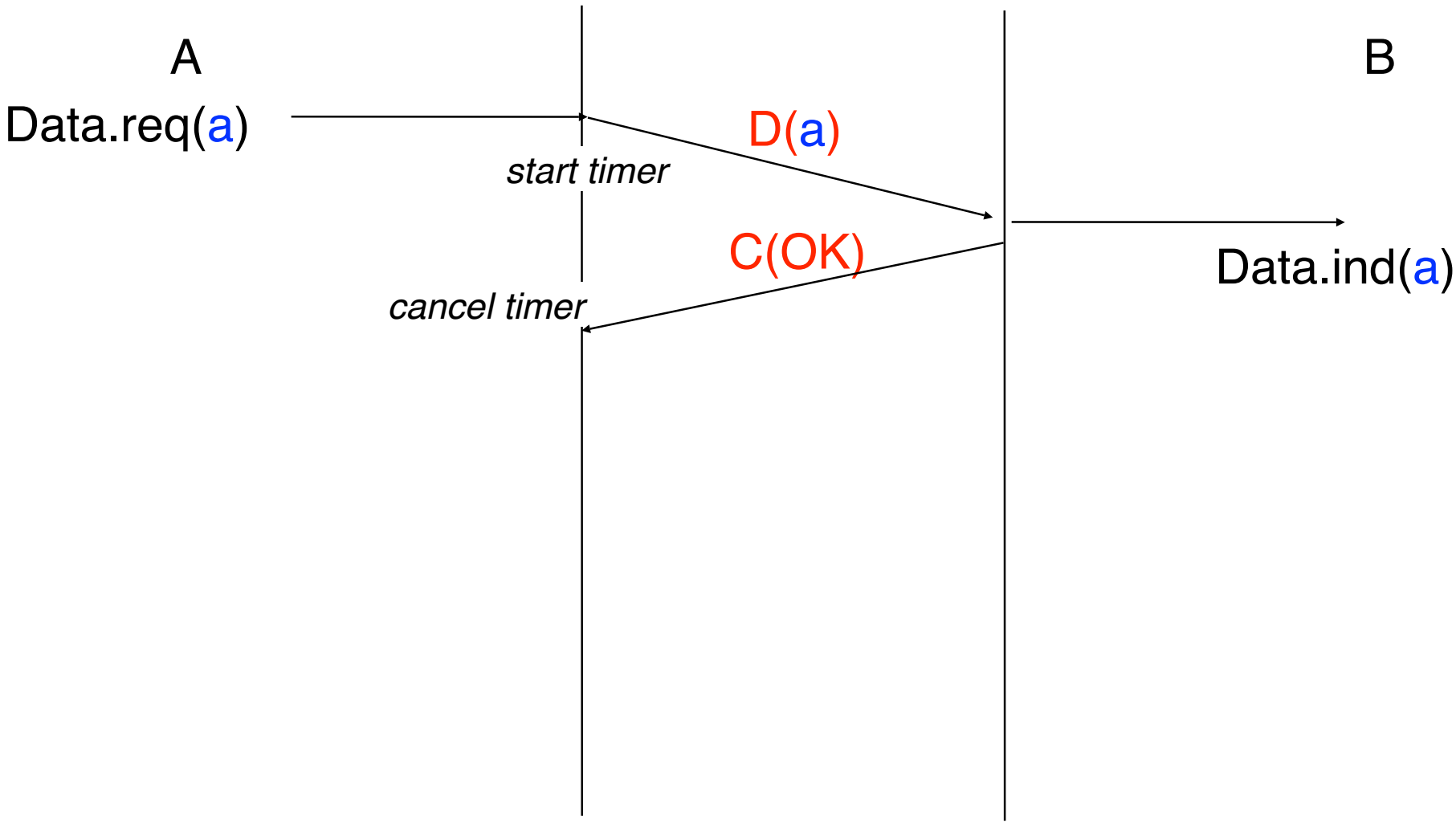
Protocol 3b : Example



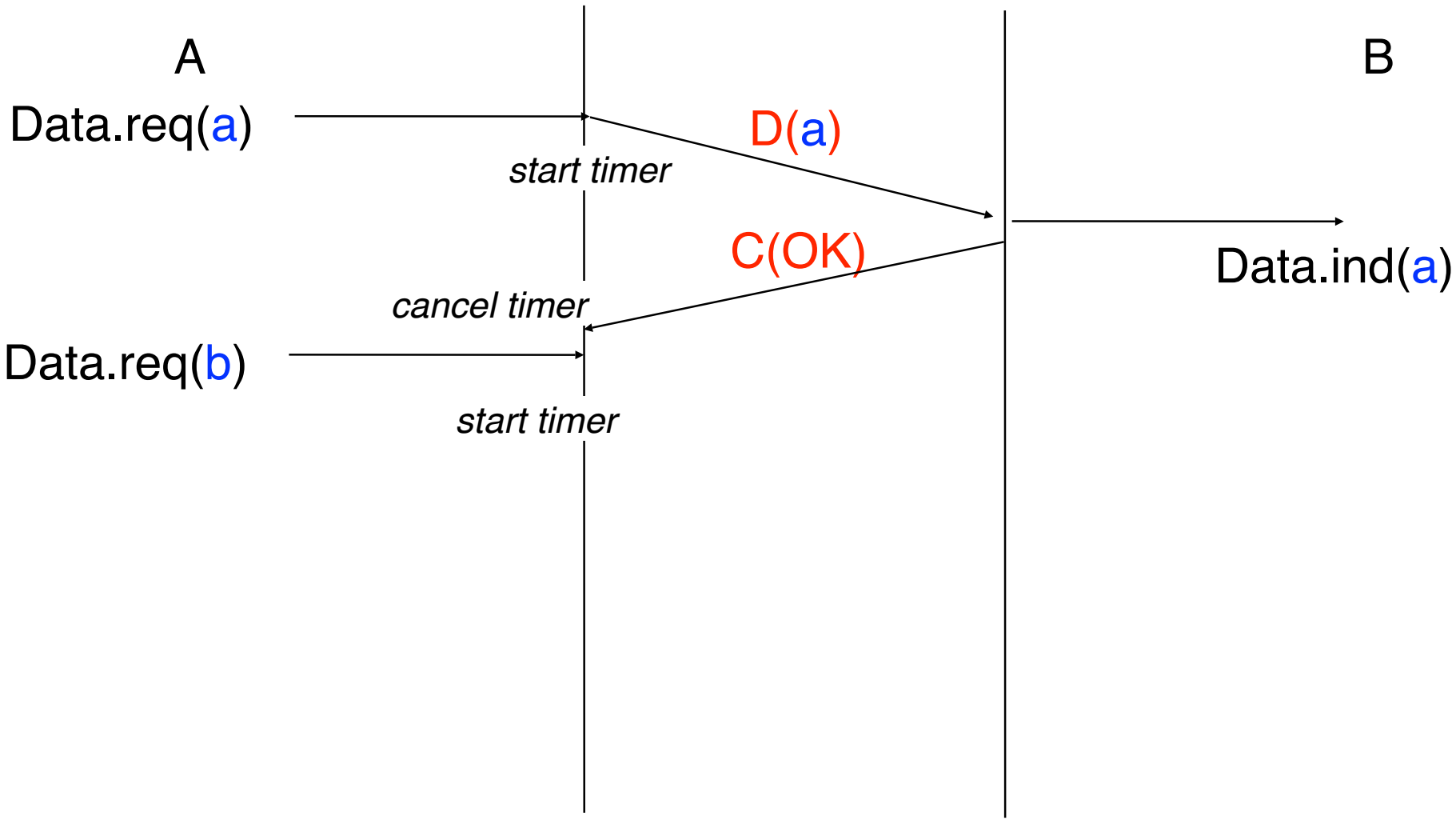
Protocol 3b : Example



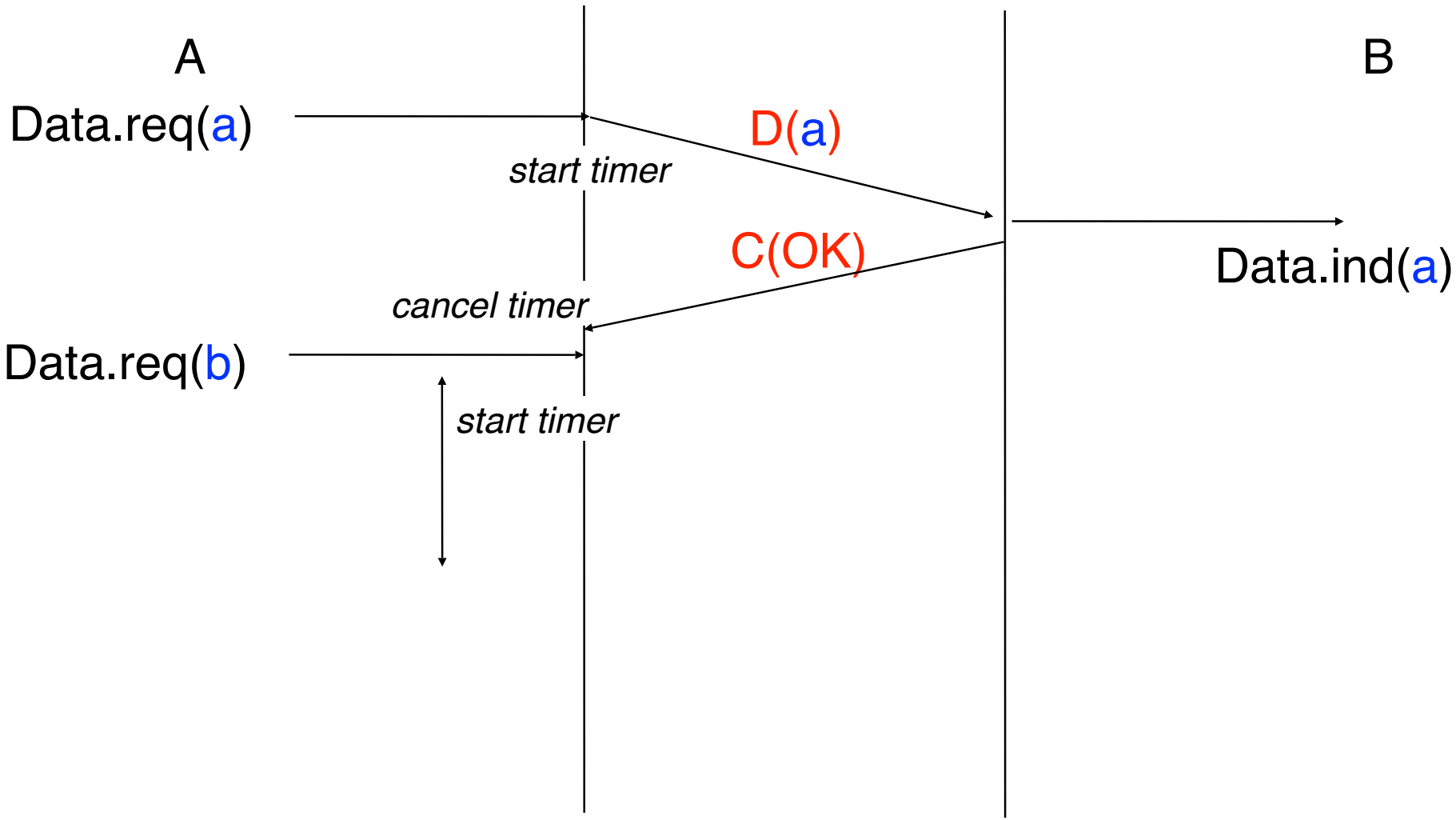
Protocol 3b : Example



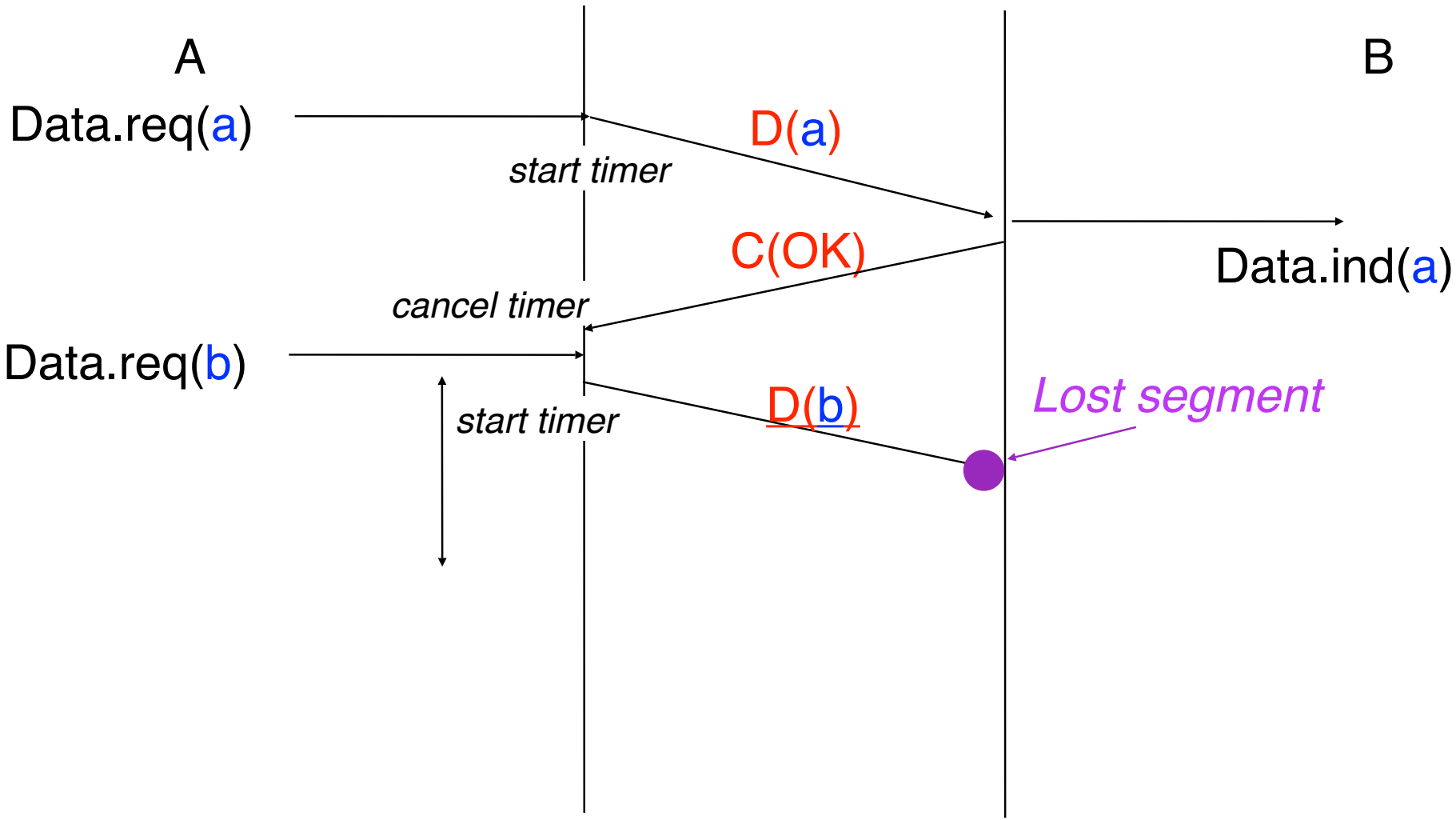
Protocol 3b : Example



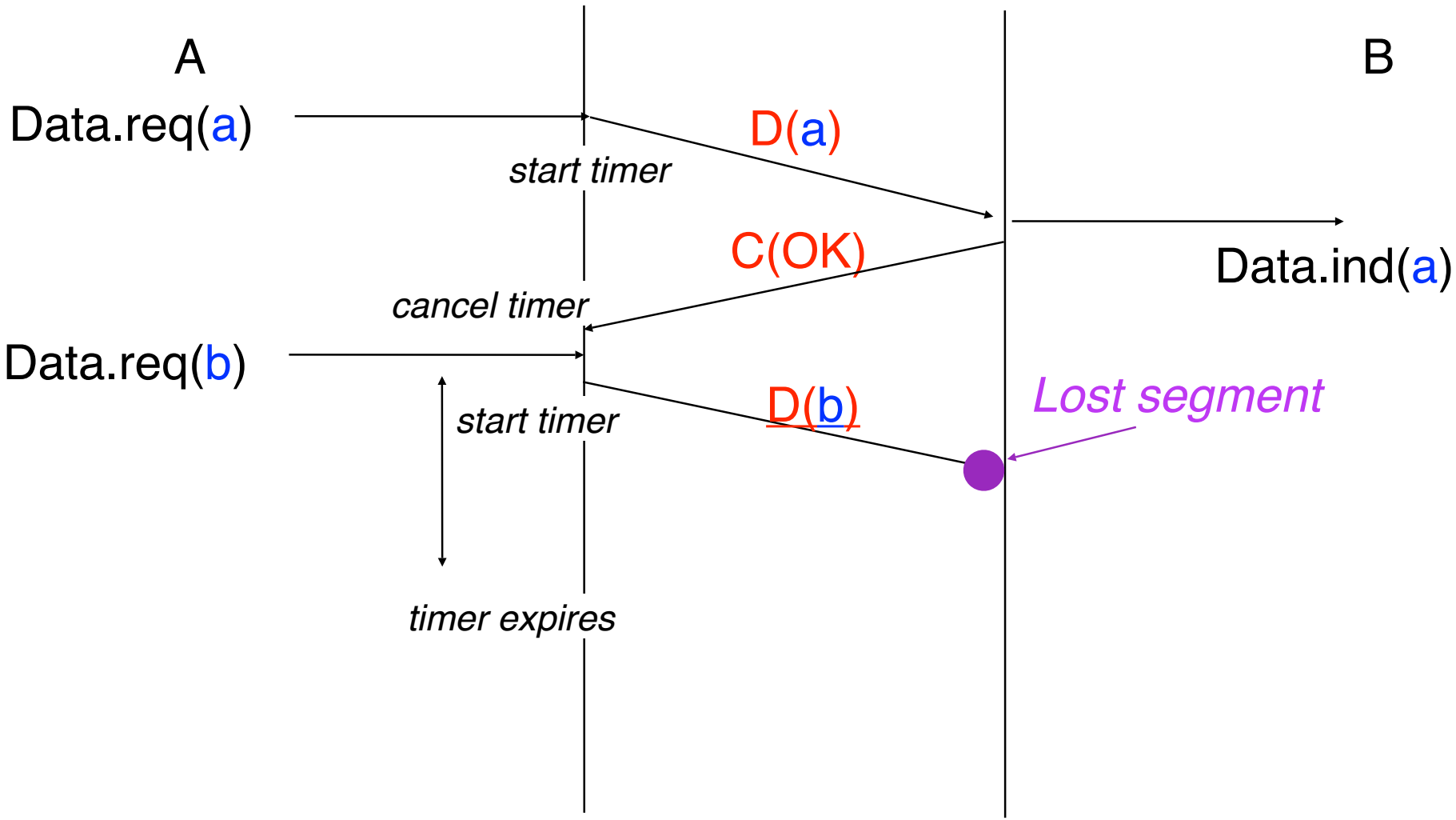
Protocol 3b : Example



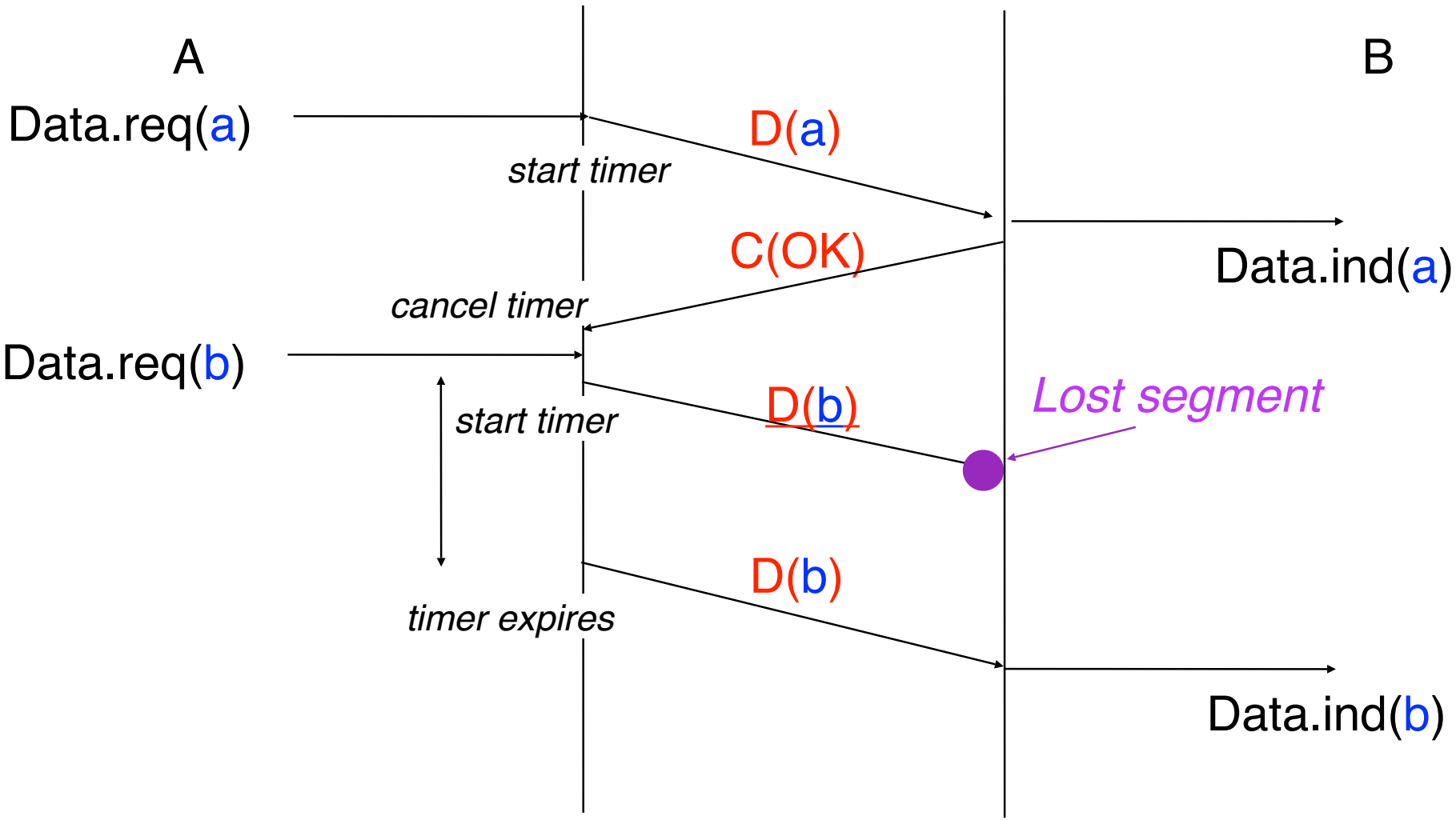
Protocol 3b : Example



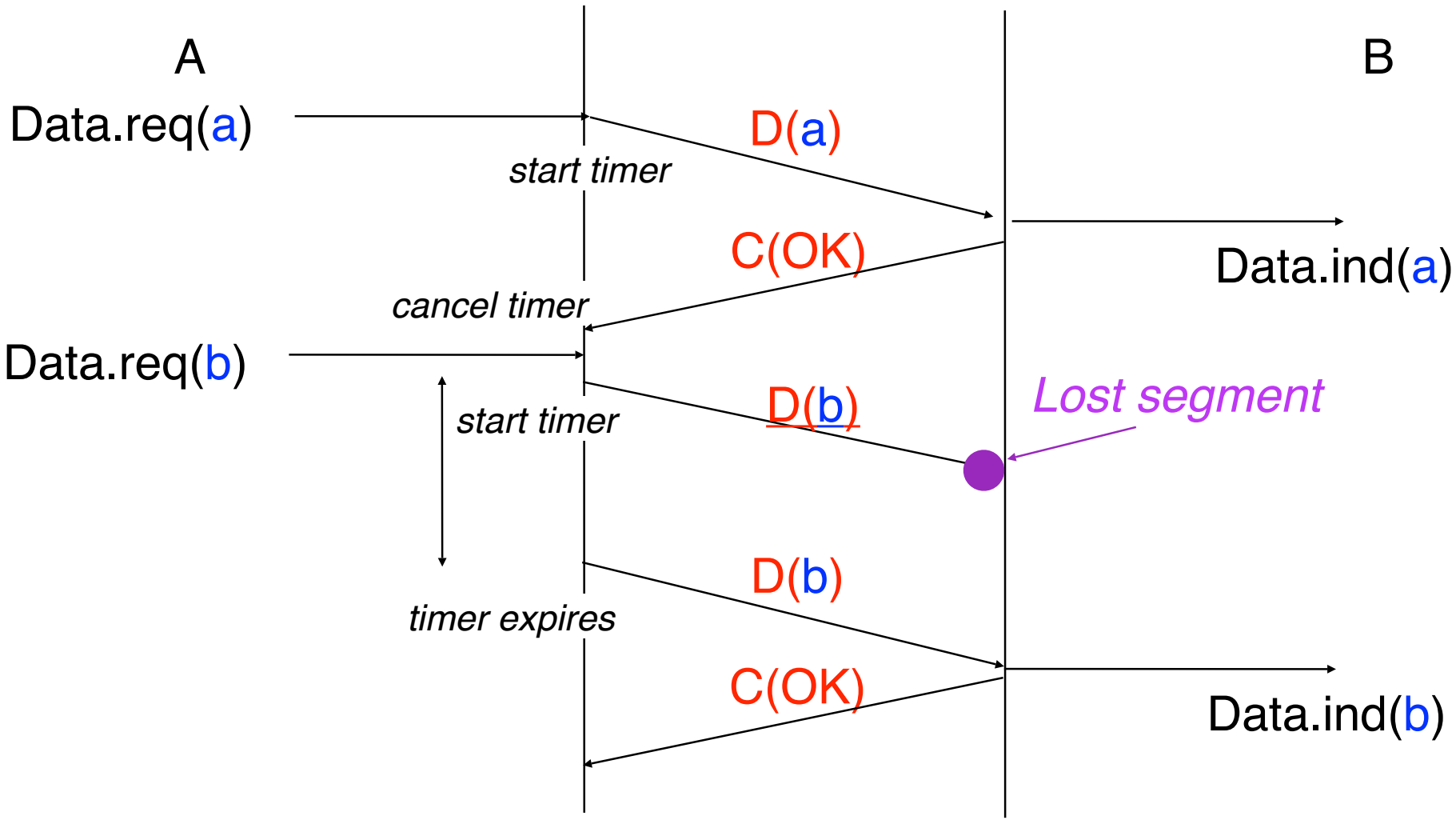
Protocol 3b : Example



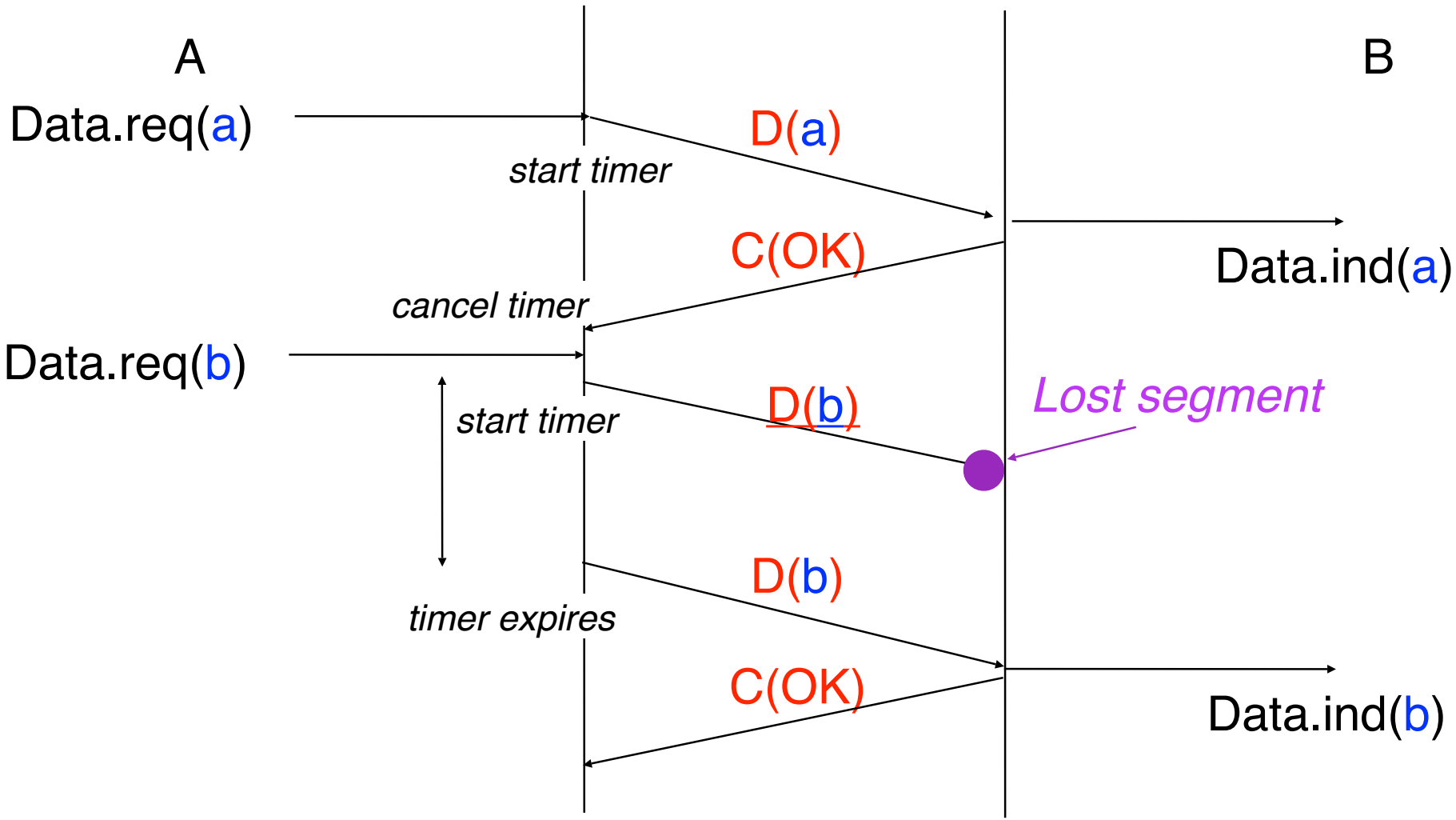
Protocol 3b : Example



Protocol 3b : Example

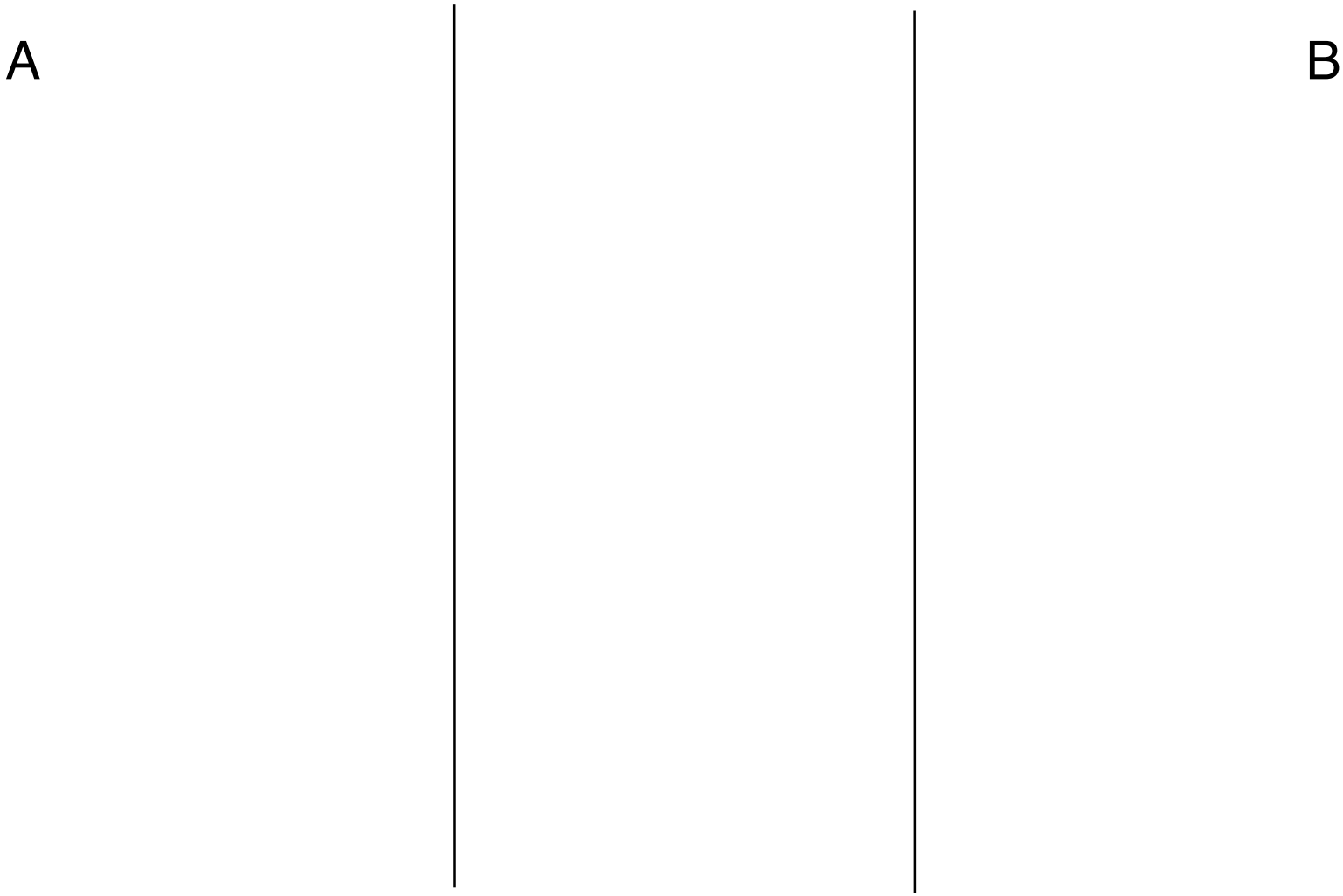


Protocol 3b : Example

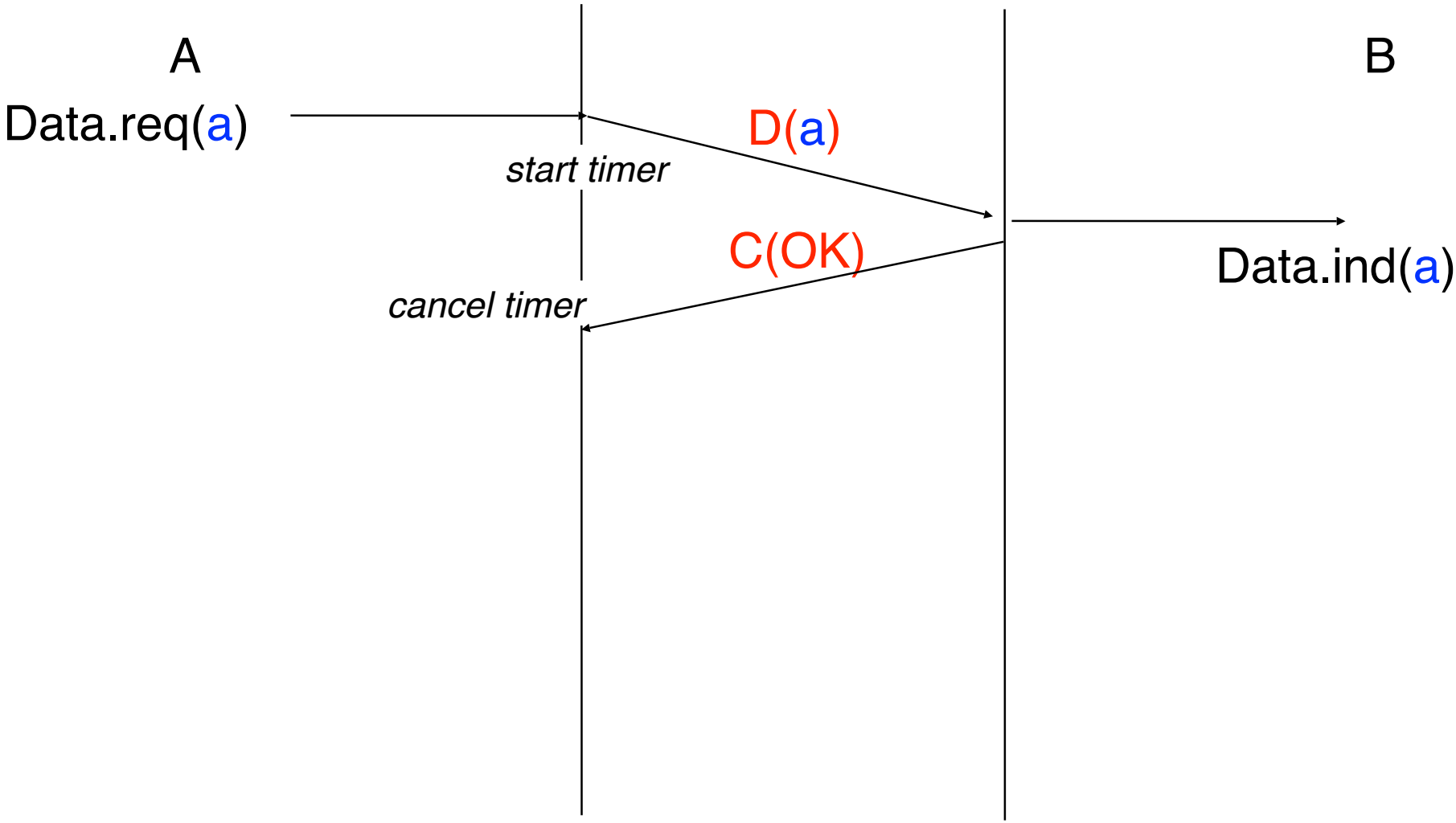


Does this protocol always work ?

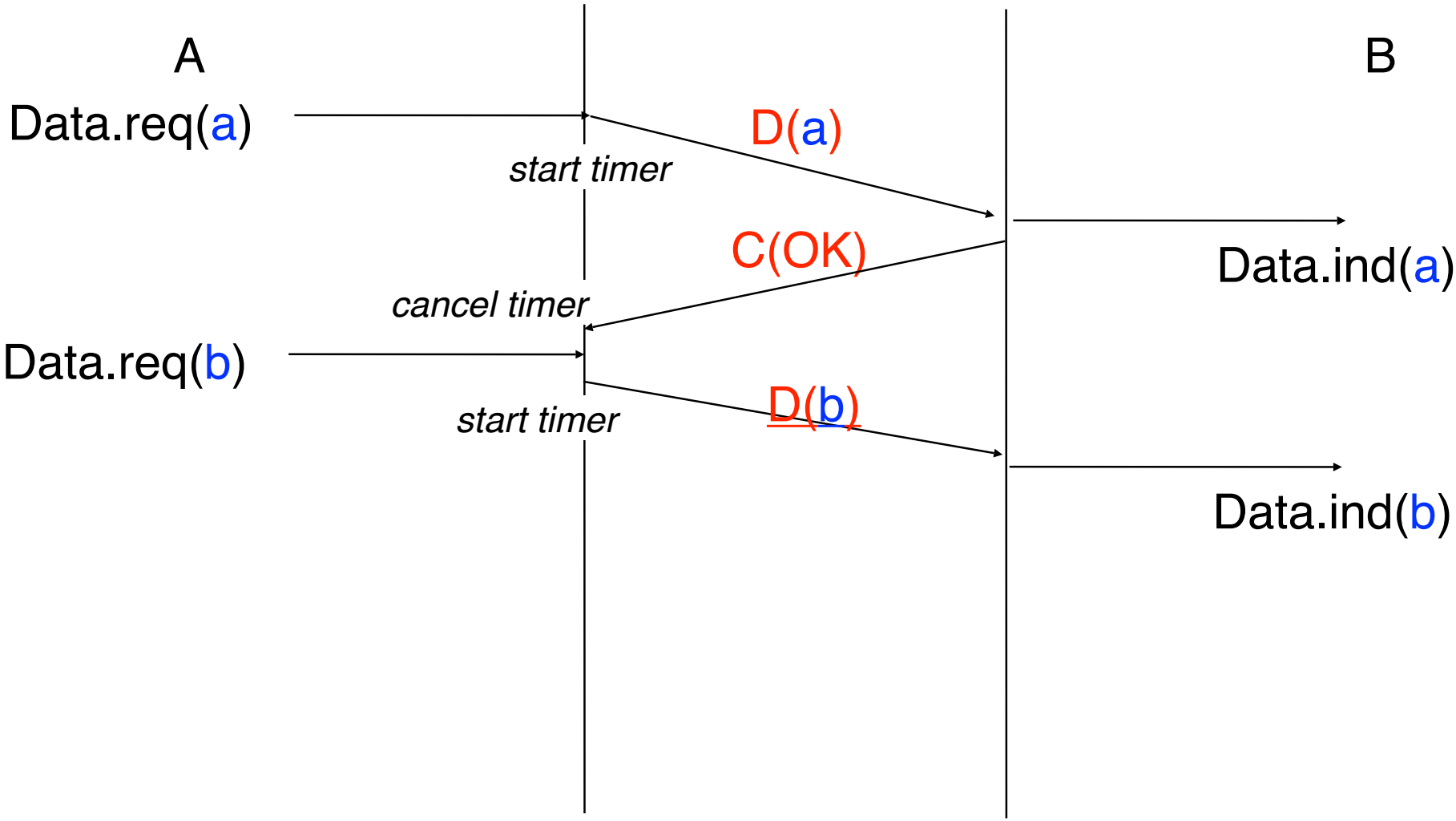
Protocol 3b : Example



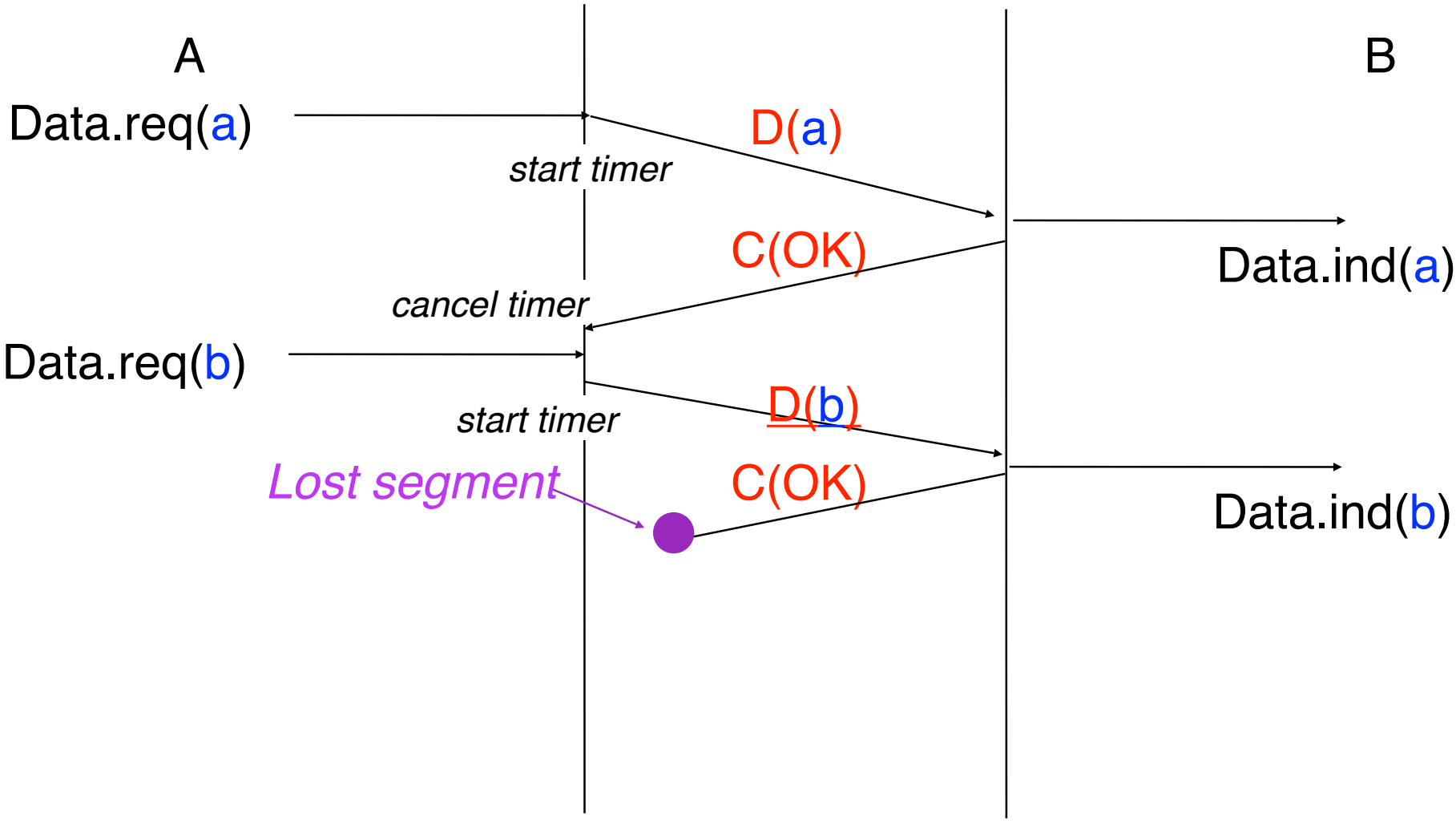
Protocol 3b : Example



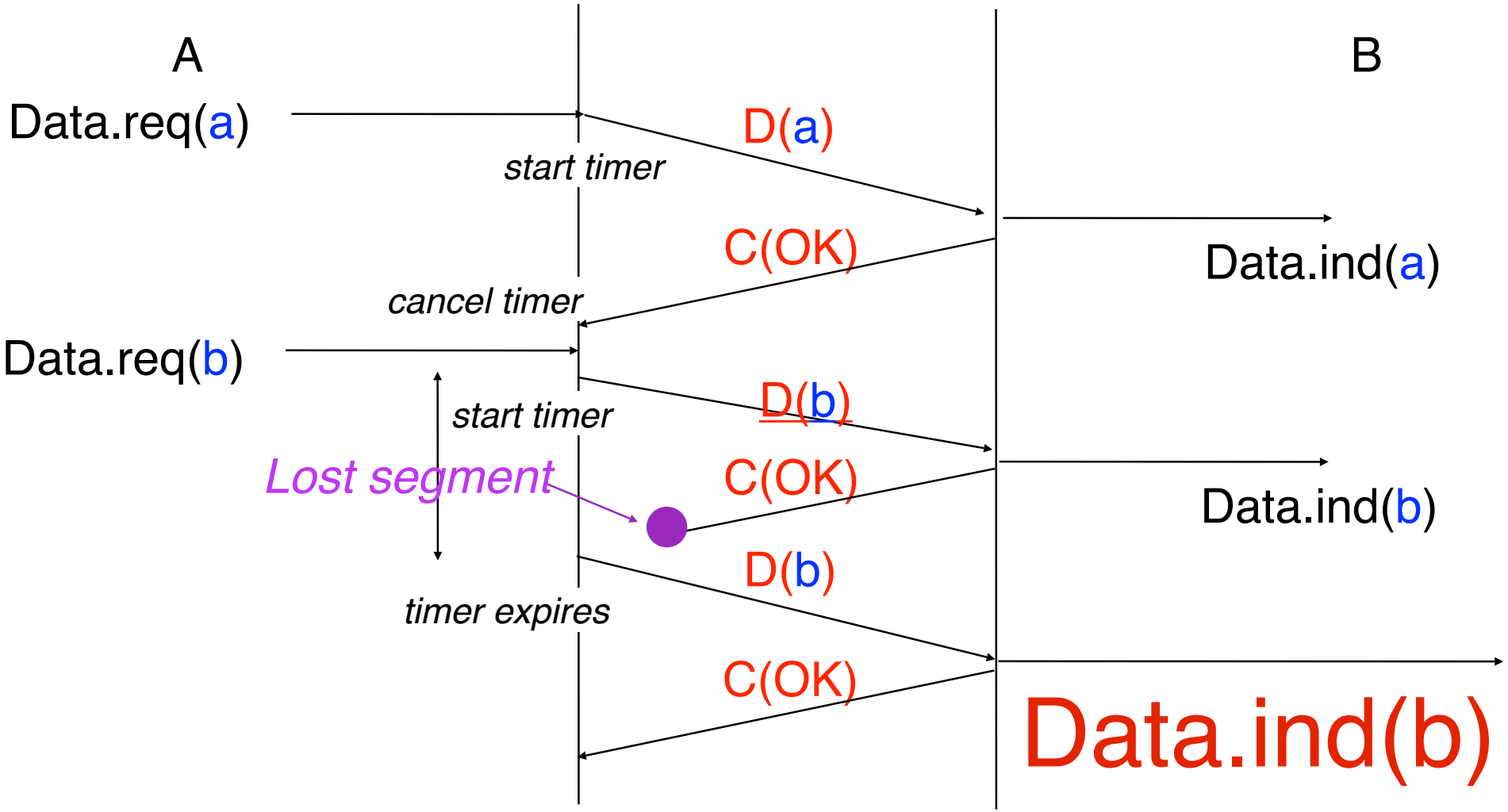
Protocol 3b : Example



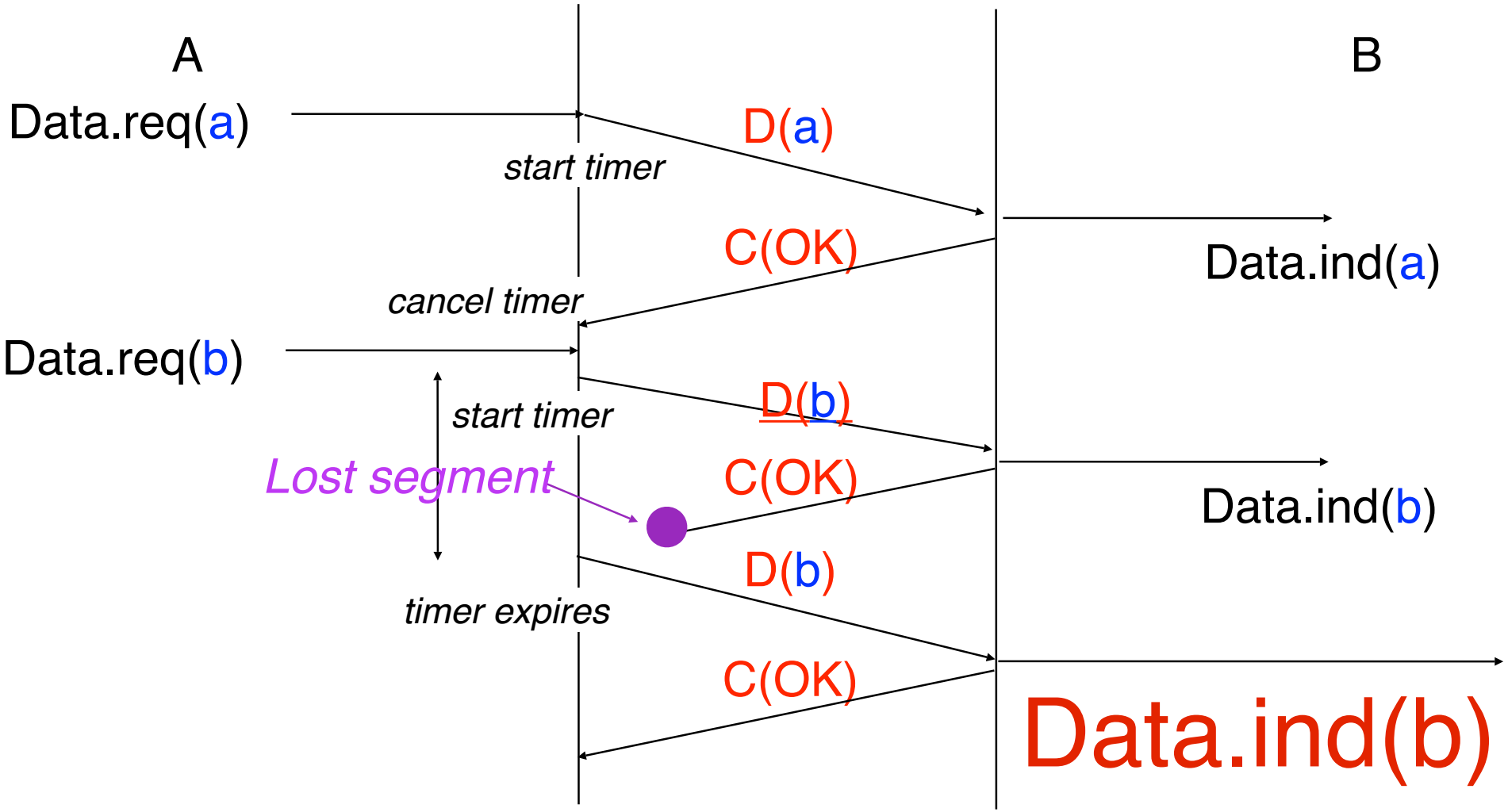
Protocol 3b : Example



Protocol 3b : Example



Protocol 3b : Example



How to solve this problem ?

Alternating bit protocol

Principles of the solution

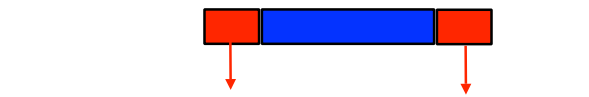
Add sequence numbers to each data segment sent by sender

By looking at the sequence number, the receiver can check whether it has already received this segment

Contents of each segment

Data segments

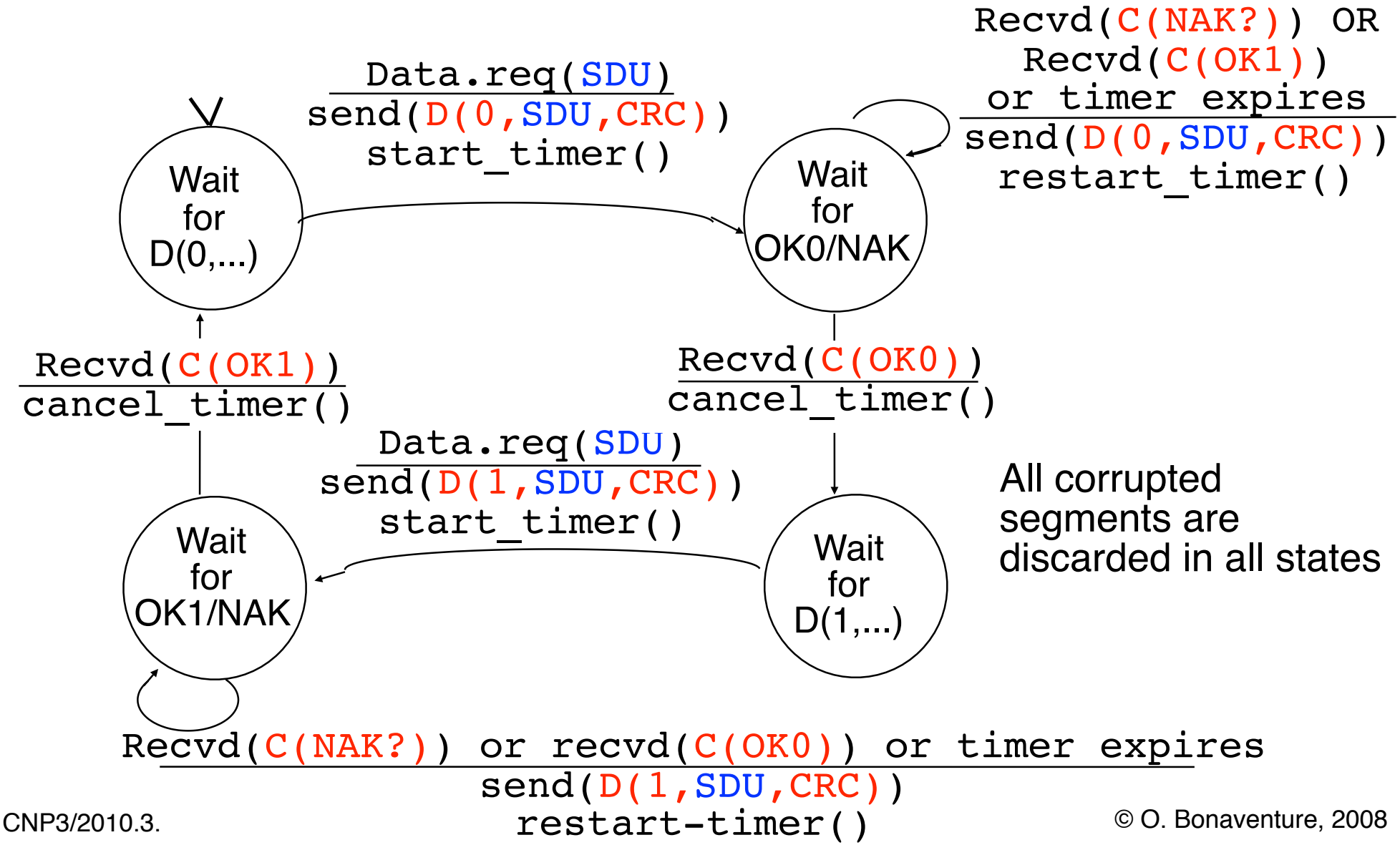
Control segments



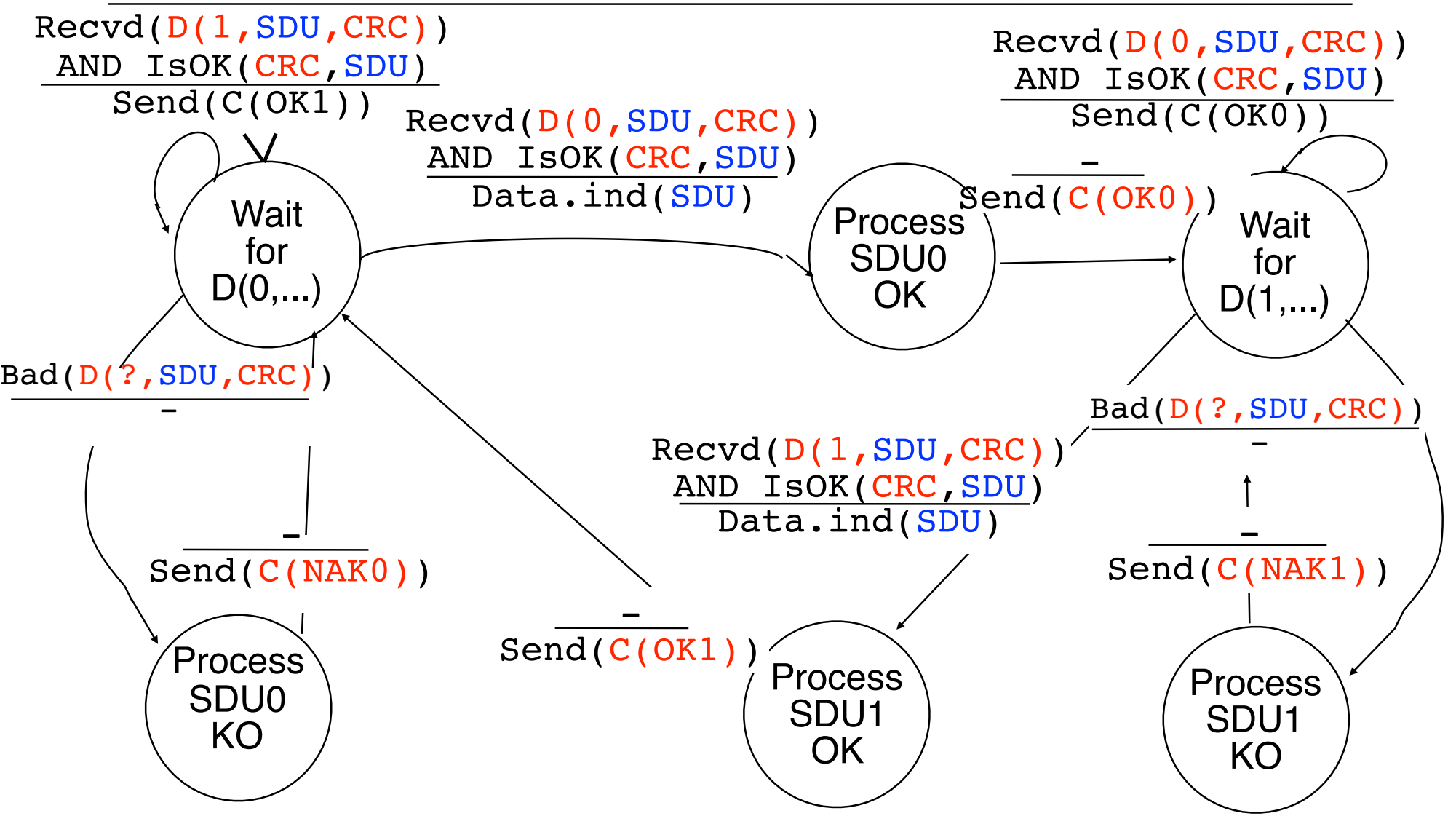
How many bits do we need for the sequence number?

a single bit is enough

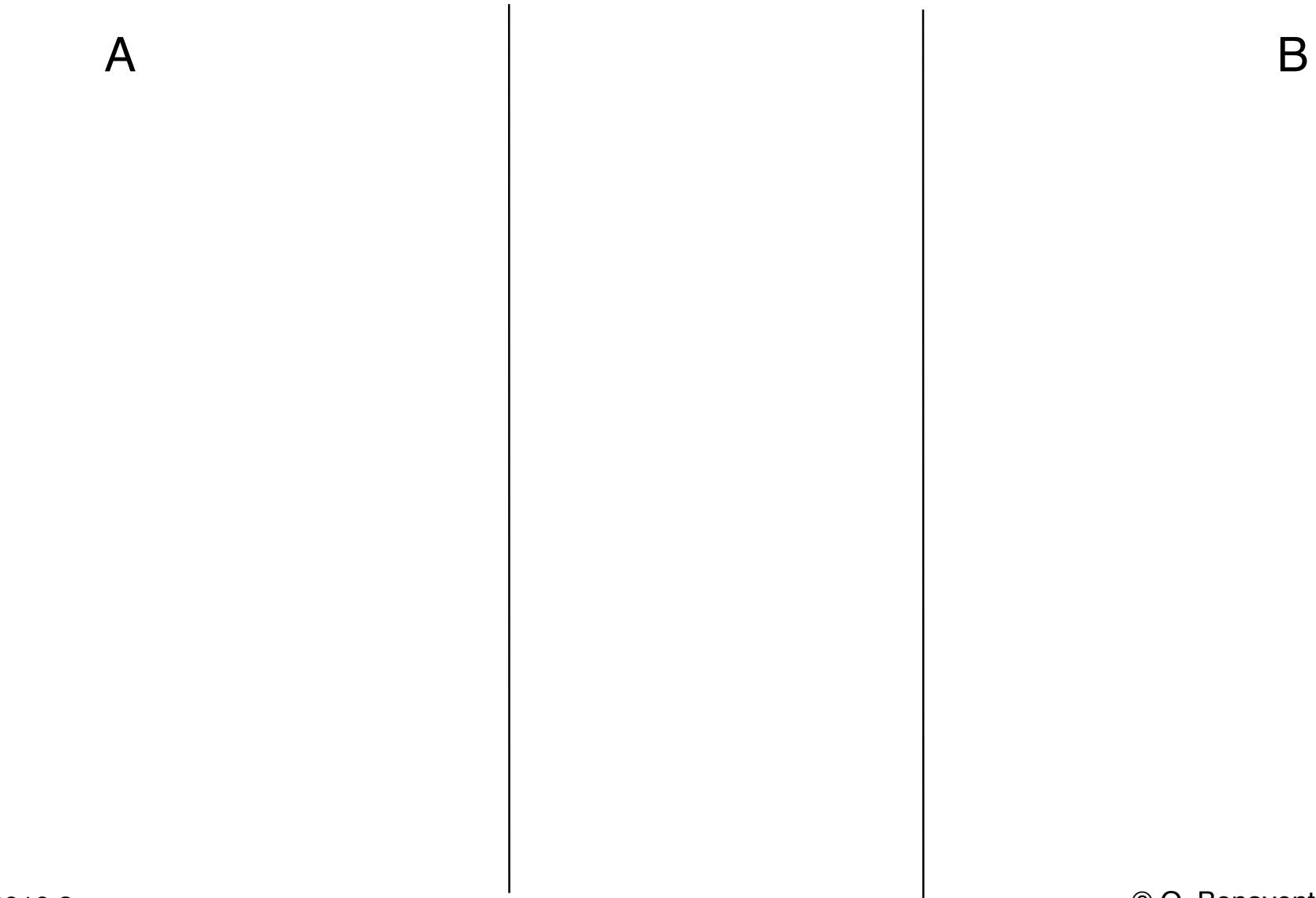
Alternating bit protocol Sender



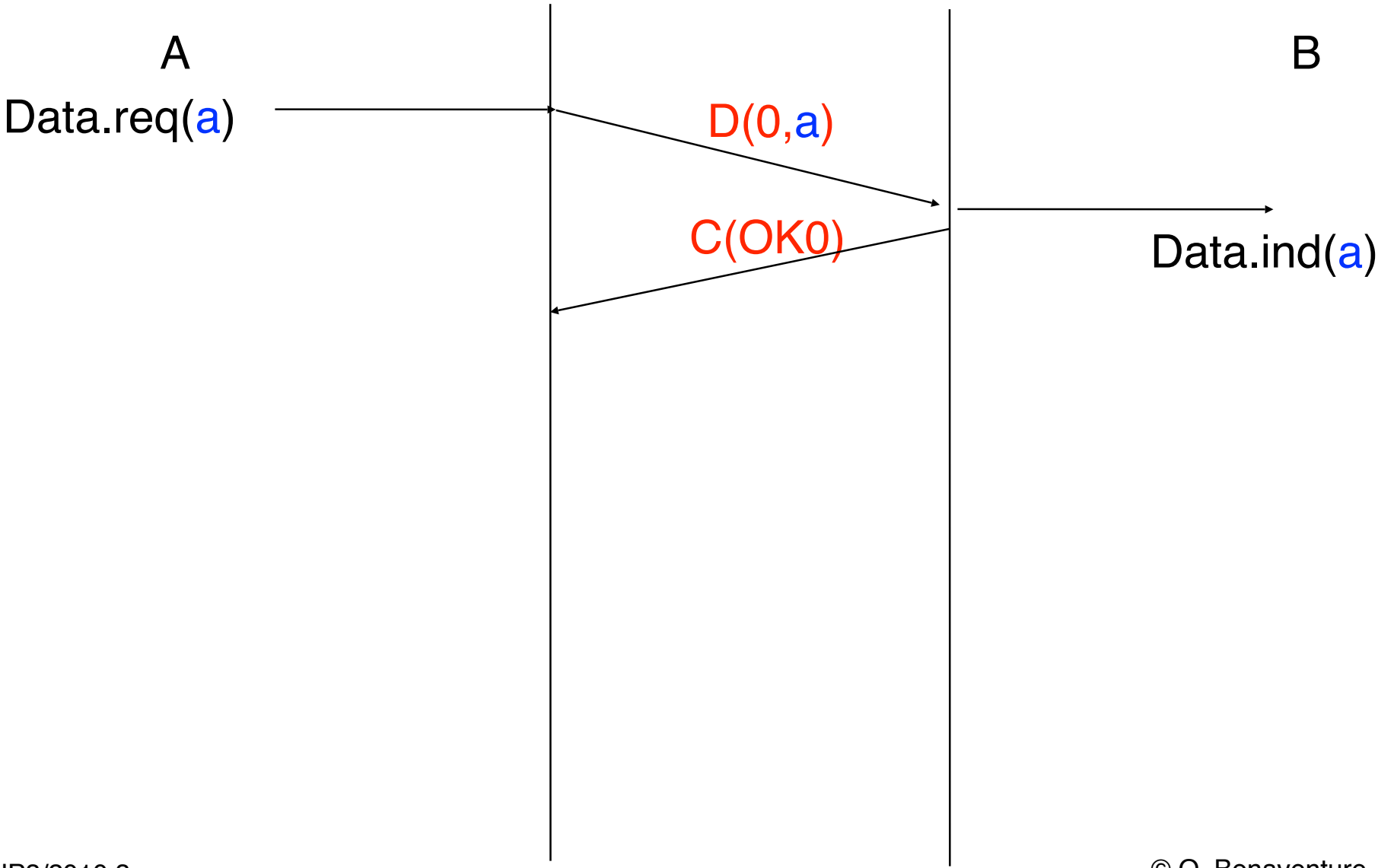
Alternating bit protocol Receiver



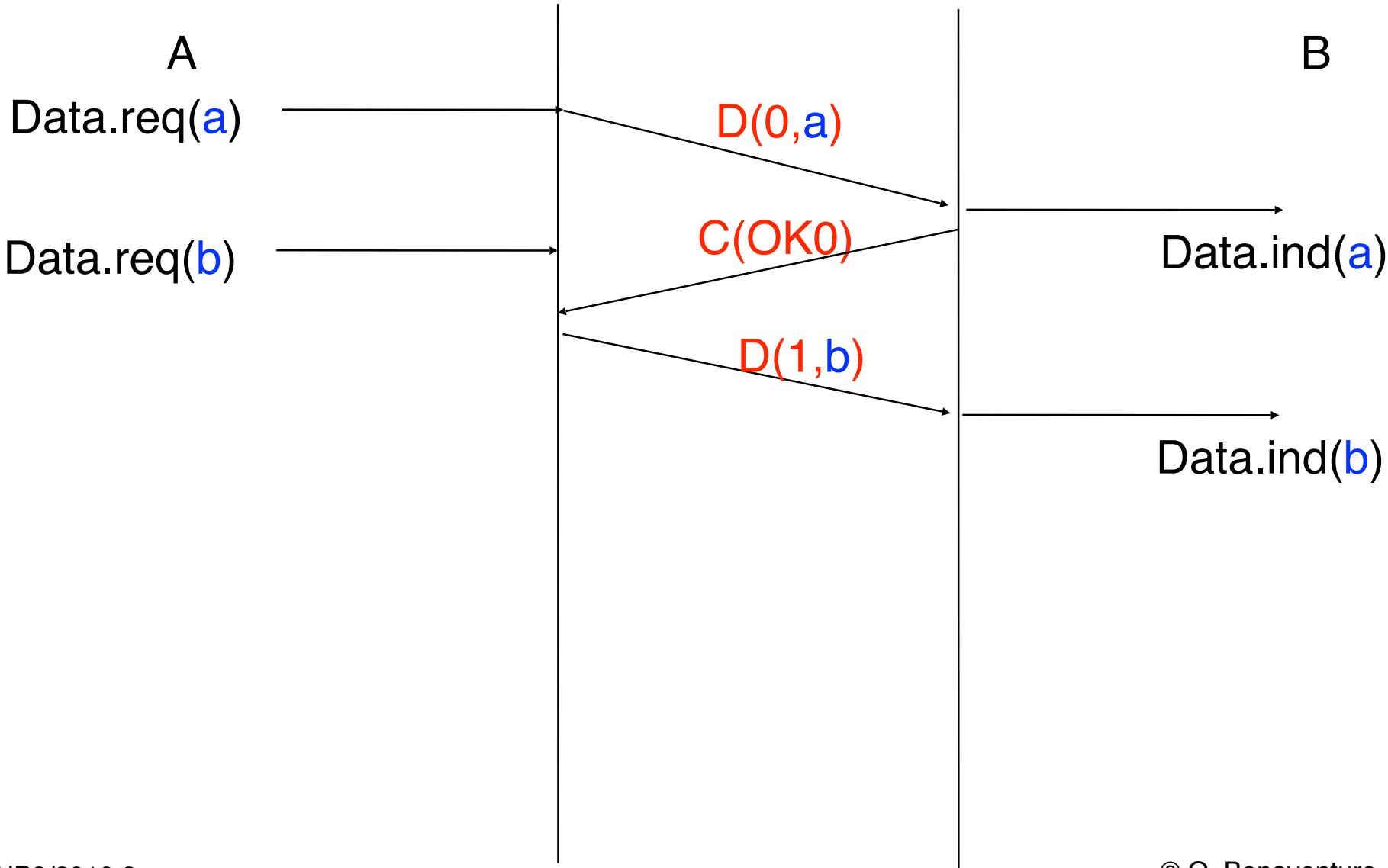
Alternating bit protocol Example



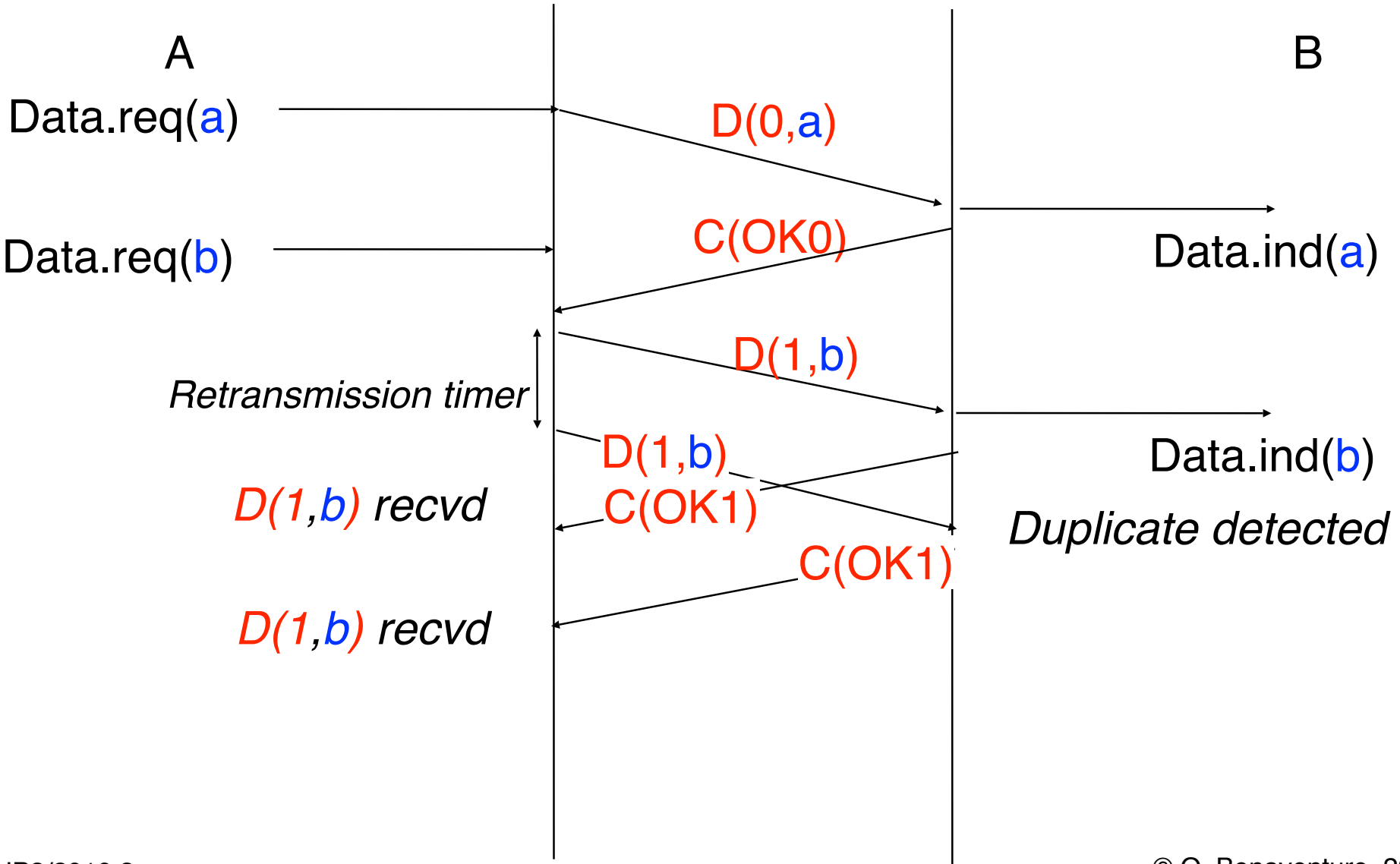
Alternating bit protocol Example



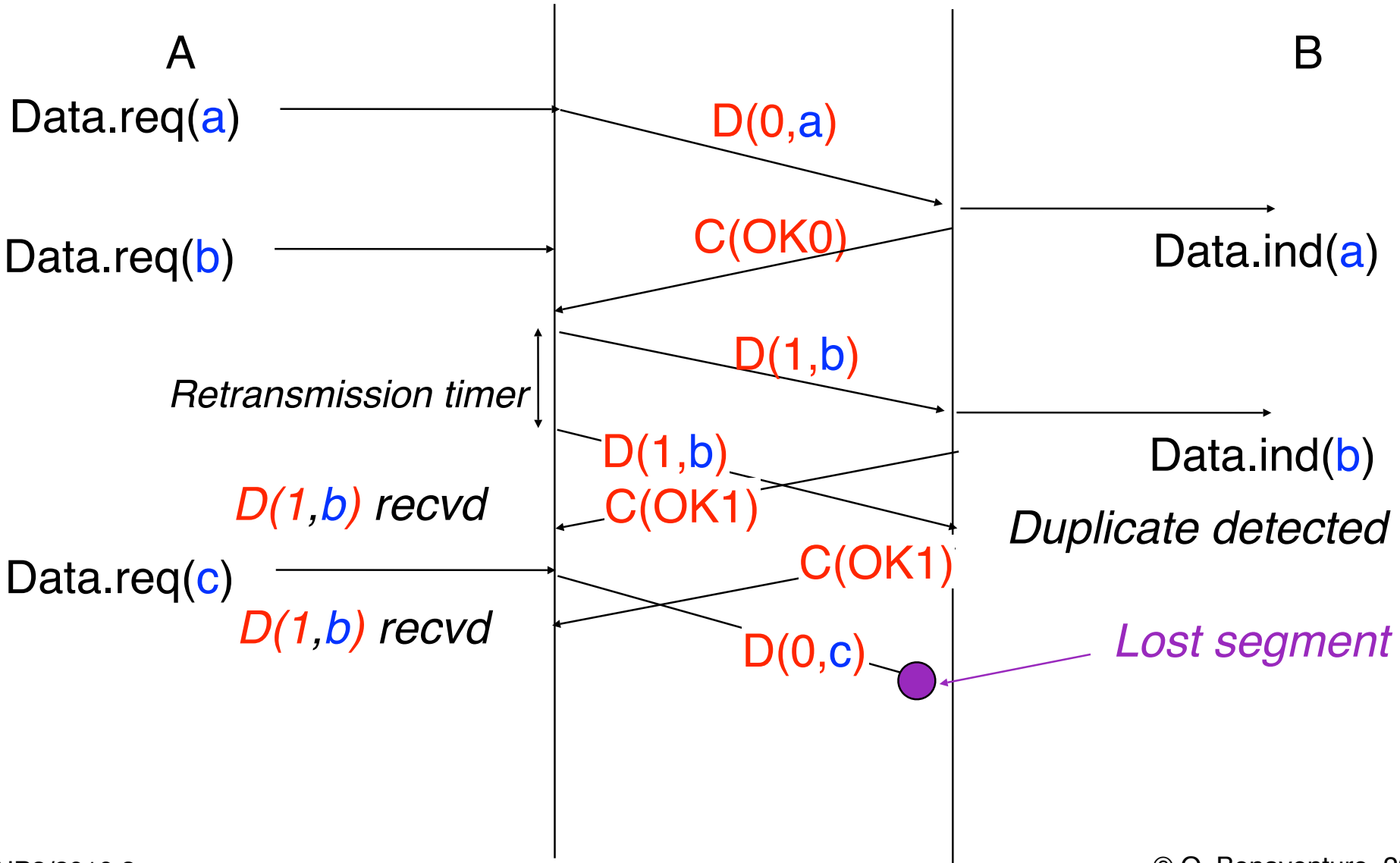
Alternating bit protocol Example



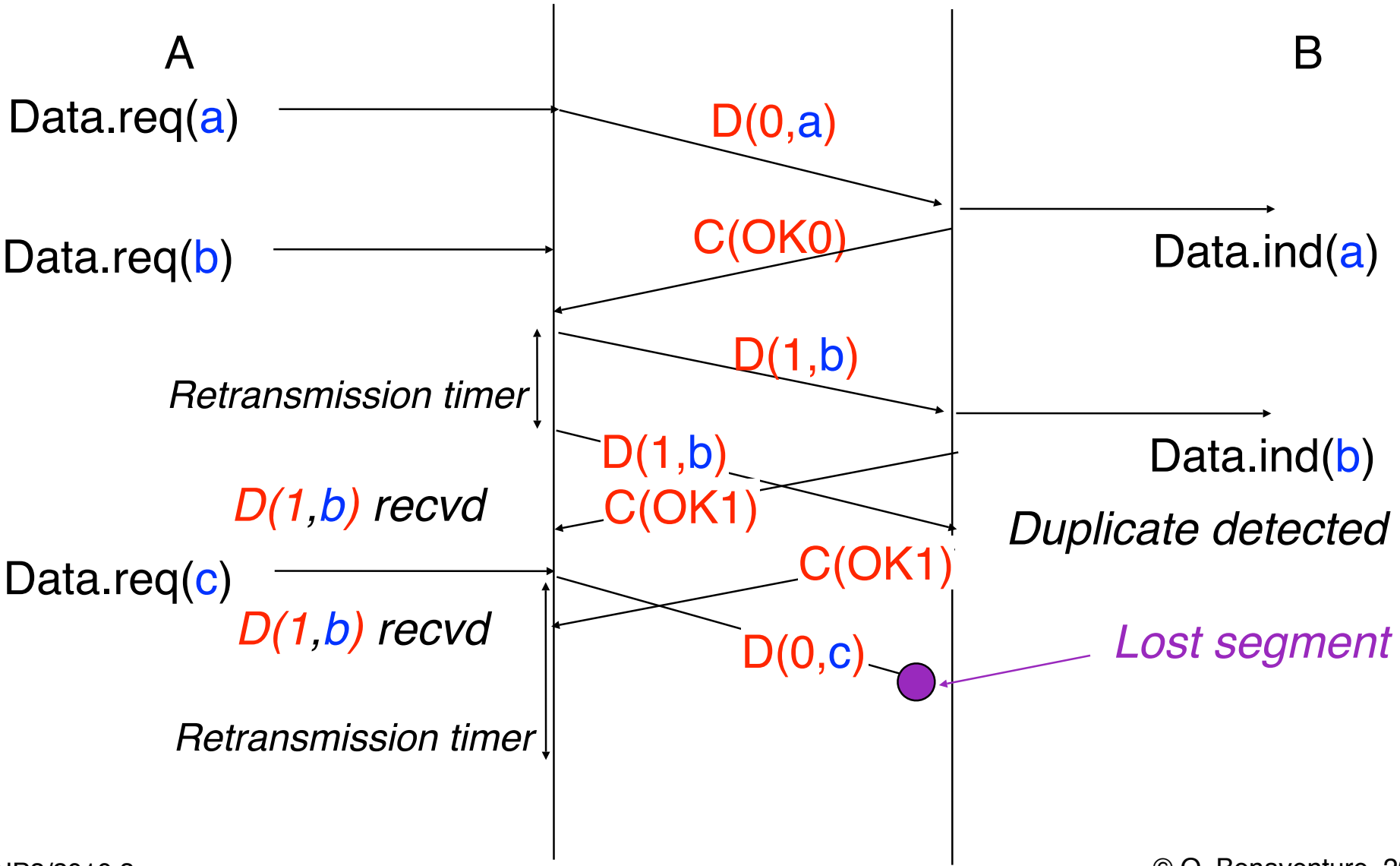
Alternating bit protocol Example



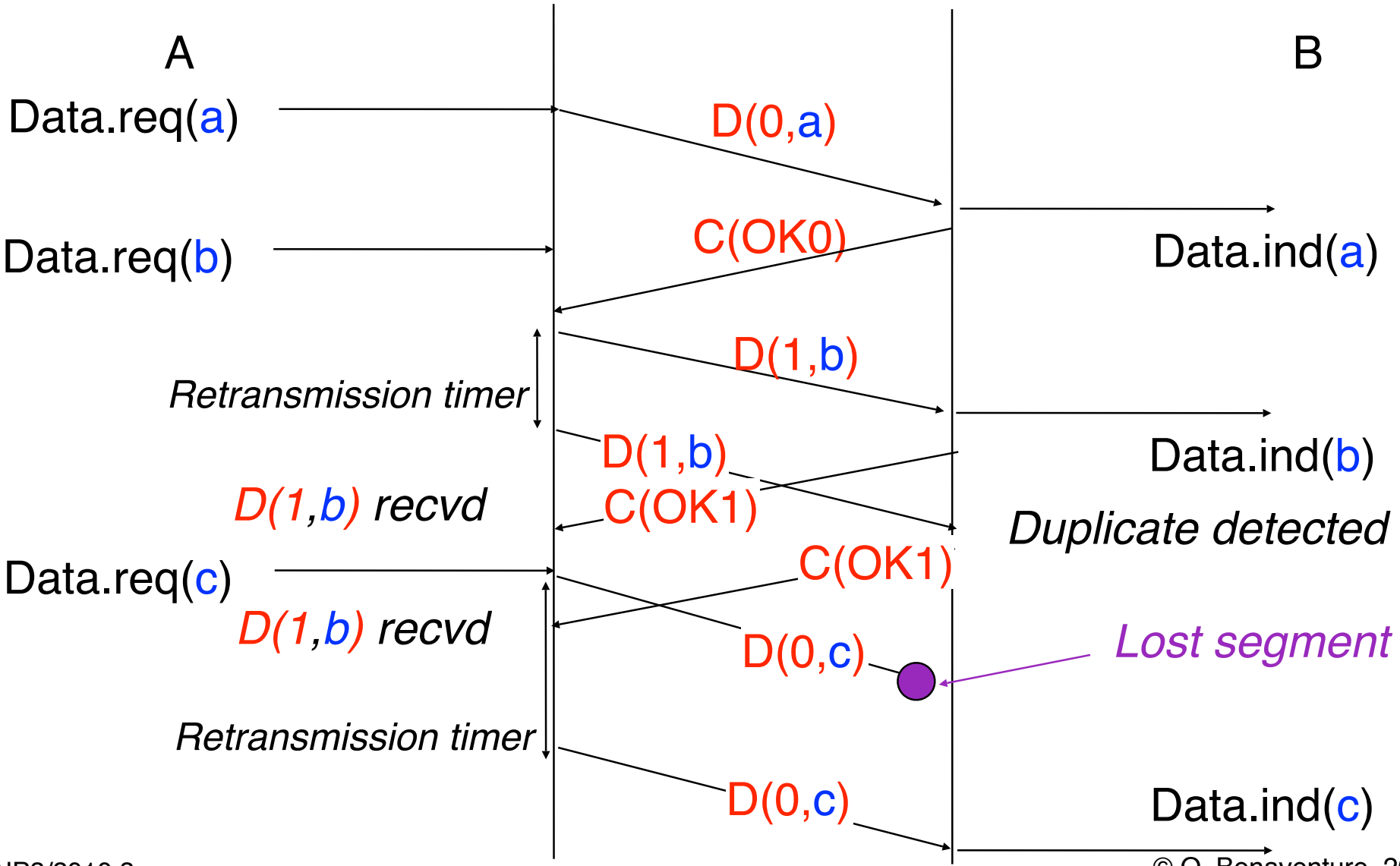
Alternating bit protocol Example



Alternating bit protocol Example



Alternating bit protocol Example



Performance of the alternating bit protocol

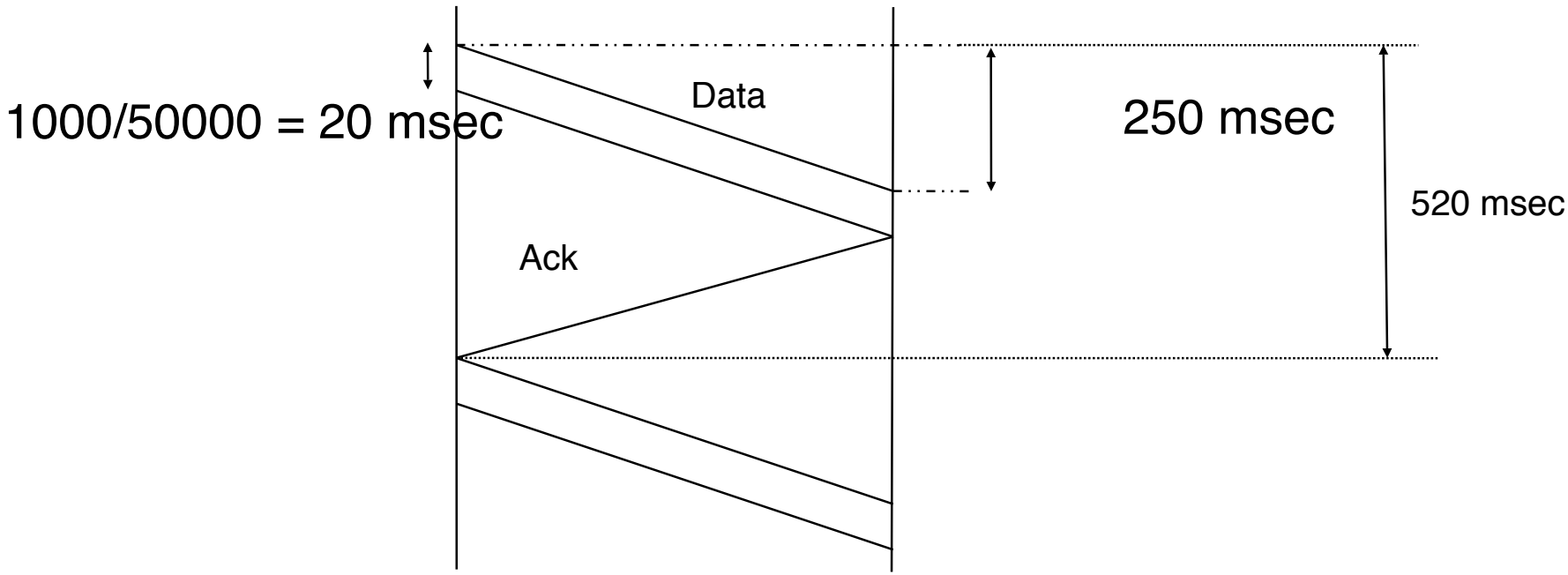
What is the performance of the ABP in this case

- One-way delay : 250 msec
- Physical layer throughput : 50 kbps
- segment size : 1000 bits

Performance of the alternating bit protocol

What is the performance of the ABP in this case

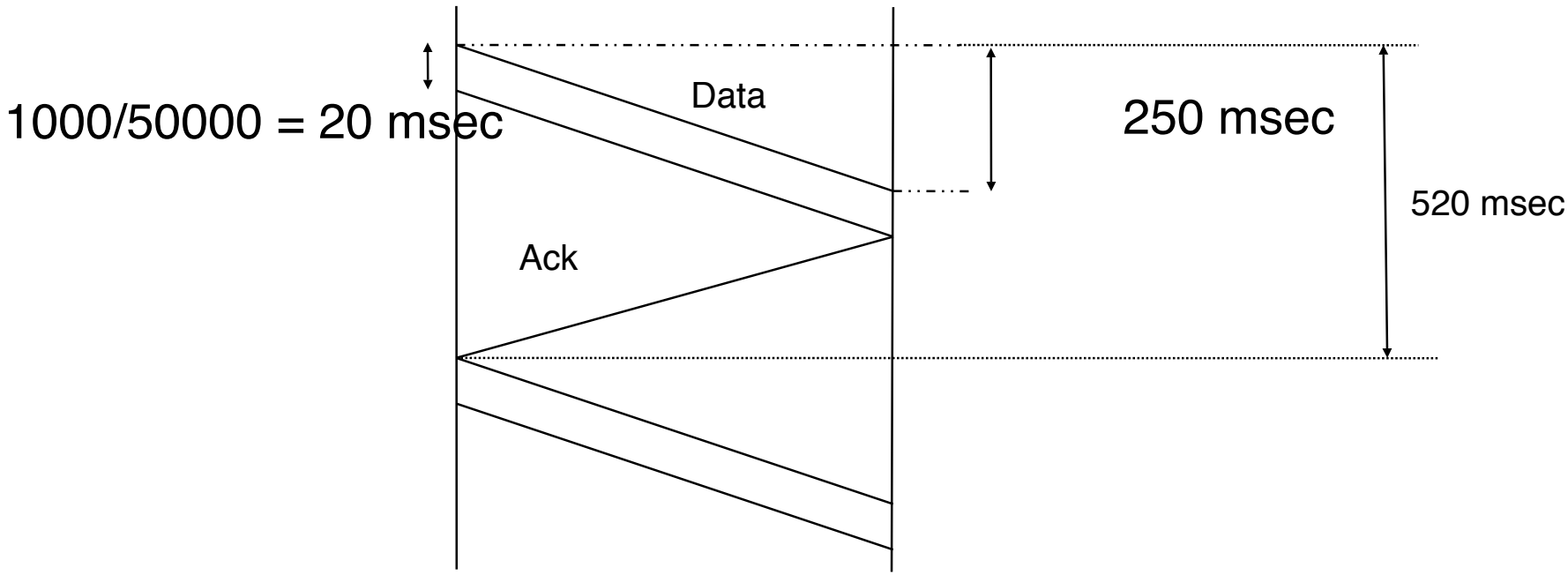
- One-way delay : 250 msec
- Physical layer throughput : 50 kbps
- segment size : 1000 bits



Performance of the alternating bit protocol

What is the performance of the ABP in this case

- One-way delay : 250 msec
- Physical layer throughput : 50 kbps
- segment size : 1000 bits



-> Performance is function of *bandwidth * round-trip-time*

How to improve the alternating bit protocol ?

Use a pipeline

Principle

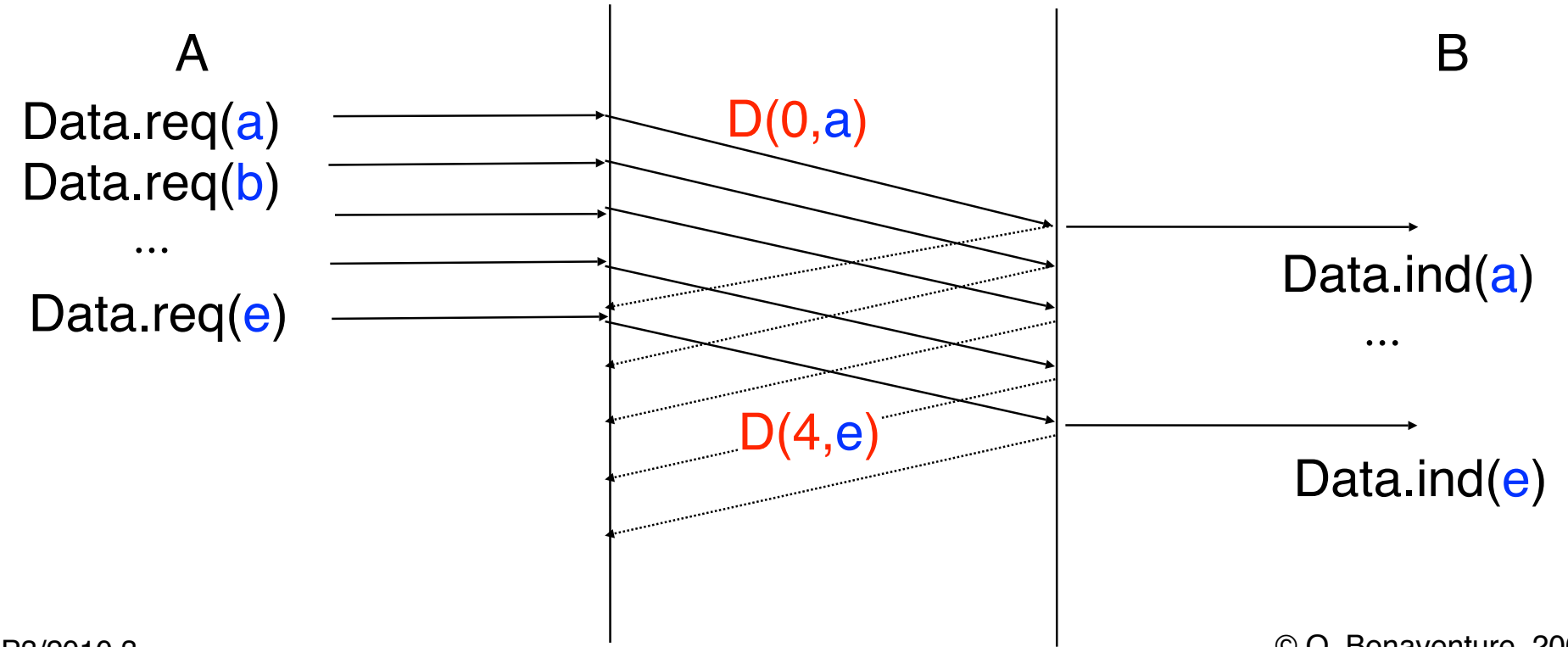
The sender should be allowed to send more than one segment while waiting for an acknowledgement from the receiver

How to improve the alternating bit protocol ?

Use a pipeline

Principle

The sender should be allowed to send more than one segment while waiting for an acknowledgement from the receiver



How to improve the alternating bit protocol ? (2)

Modifications to the alternating bit protocol

Sequence numbers inside each segment

- Each data segment contains its own sequence number
- Each control segment indicates the sequence number of the data segment being acknowledged (OK/NAK)

Sender

- Needs enough buffers to store the data segments that have not yet been acknowledged to be able to retransmit them if required

Receiver

- Needs enough buffers to store the out-of-sequence segments

How to improve the alternating bit protocol ? (2)

Modifications to the alternating bit protocol

Sequence numbers inside each segment

Each data segment contains its own sequence number
Each control segment indicates the sequence number of the data segment being acknowledged (OK/NAK)

Sender

Needs enough buffers to store the data segments that have not yet been acknowledged to be able to retransmit them if required

Receiver

Needs enough buffers to store the out-of-sequence segments

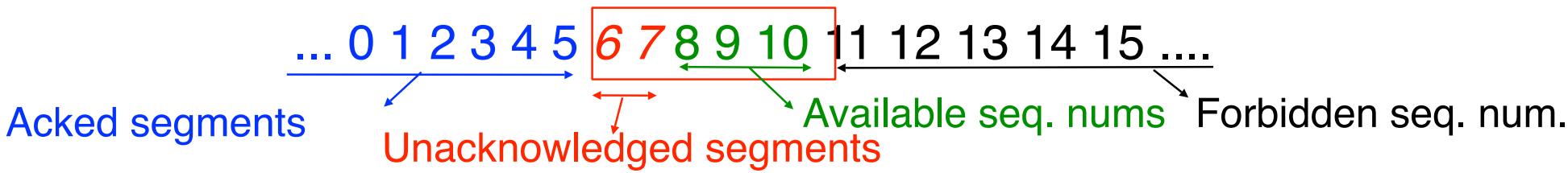
How to avoid an overflow of the receiver's buffers ?

Sliding window

Principle

Sender keeps a list of all the segments that it is allowed to send

`sending_window`



Receiver also maintains a receiving window with the list of acceptable sequence number

receiving_window

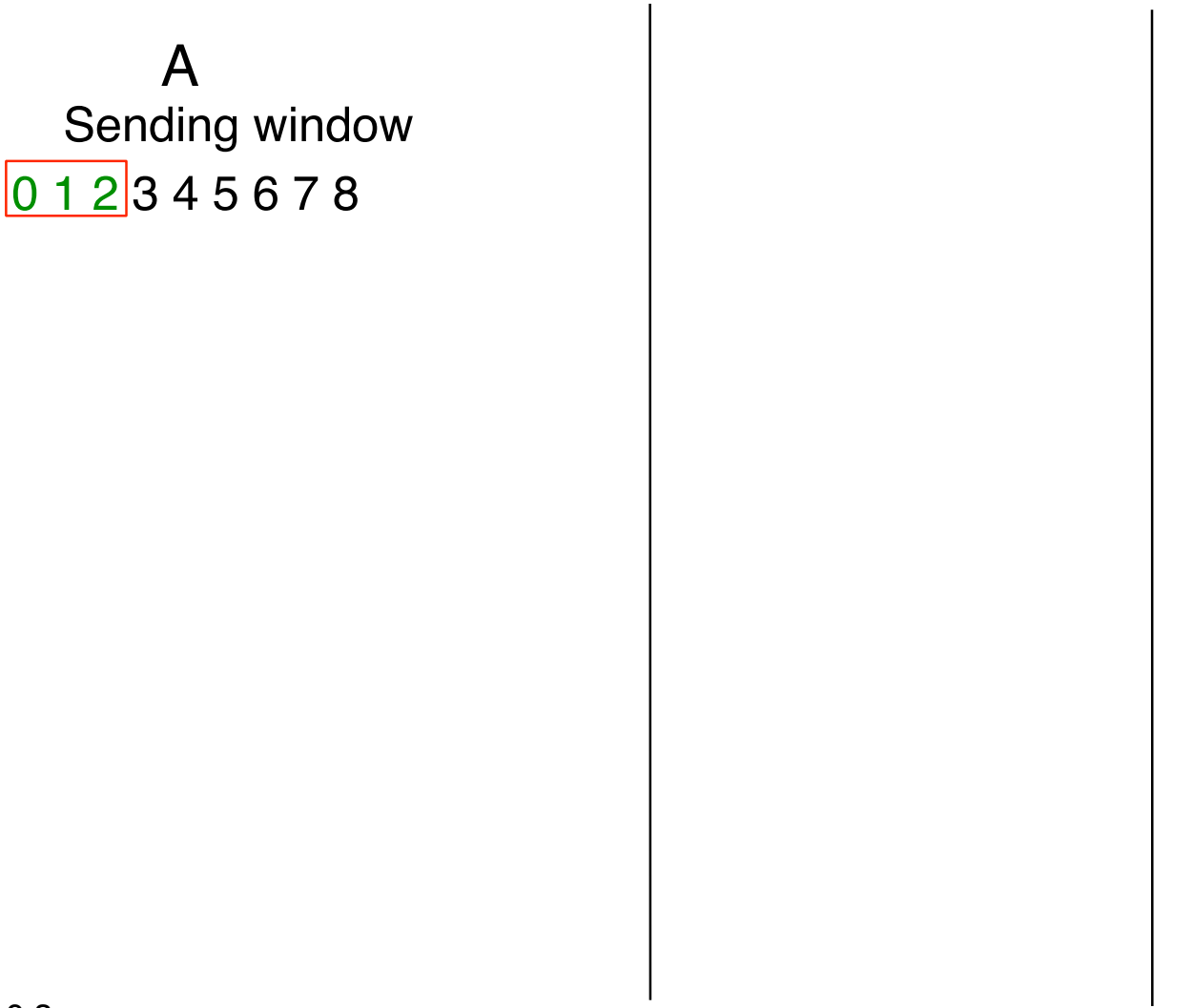
Sender and receiver must use compatible windows

sending_window ≤ receiving window

For example, window size is a constant for a given protocol or negotiated during connection establishment phase

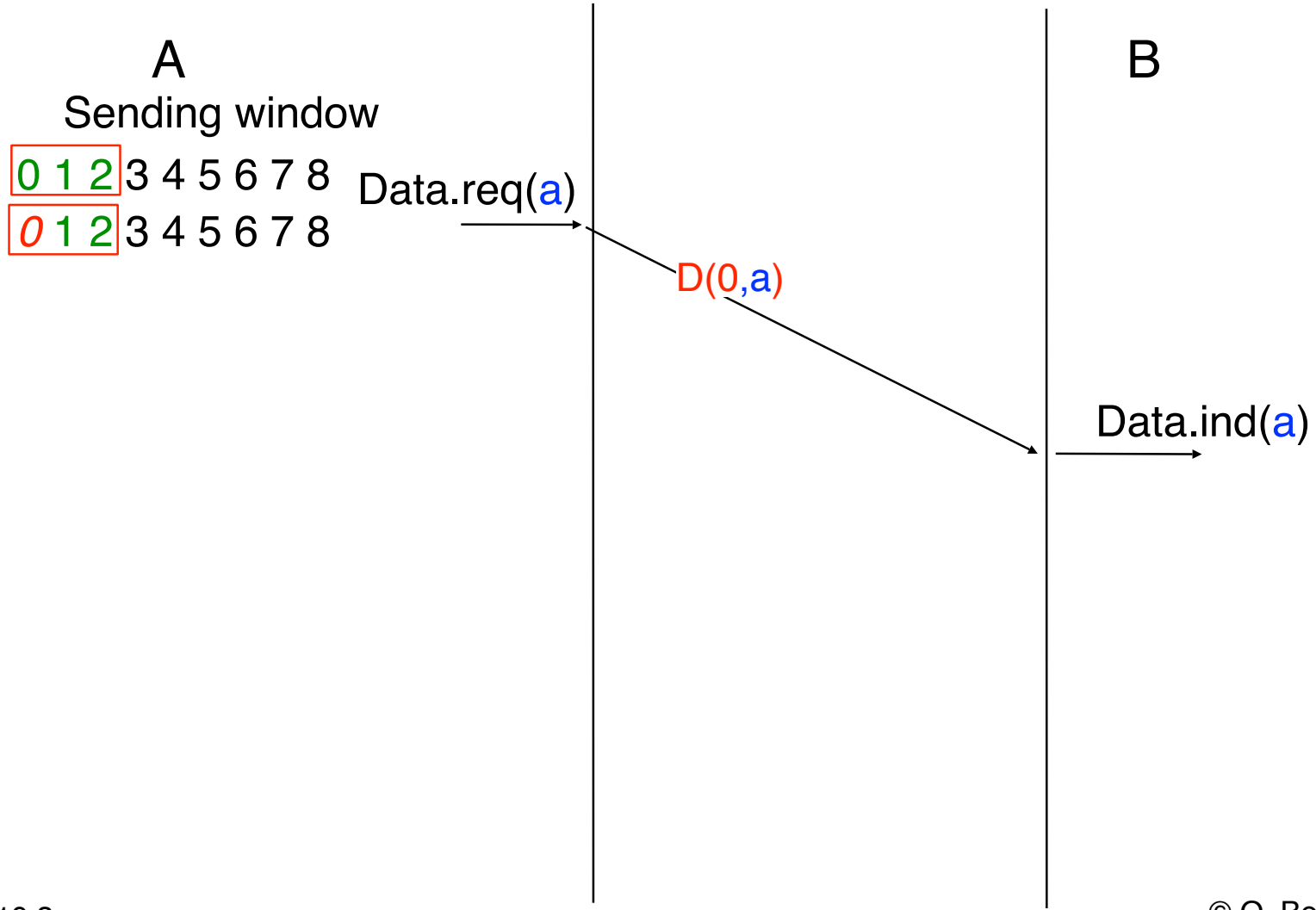
Sliding windows : example

Sending and receiving window : 3 segments



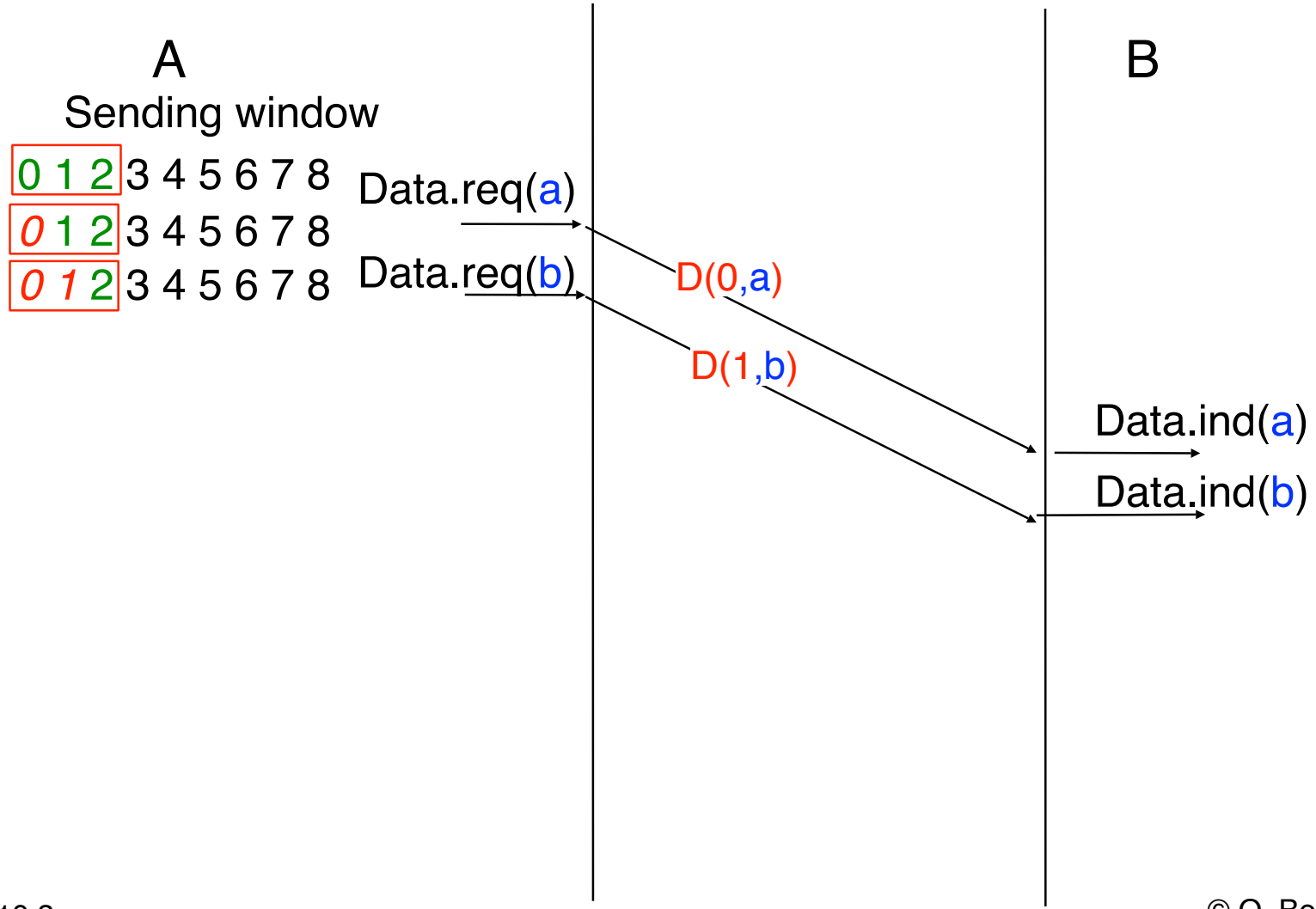
Sliding windows : example

Sending and receiving window : 3 segments



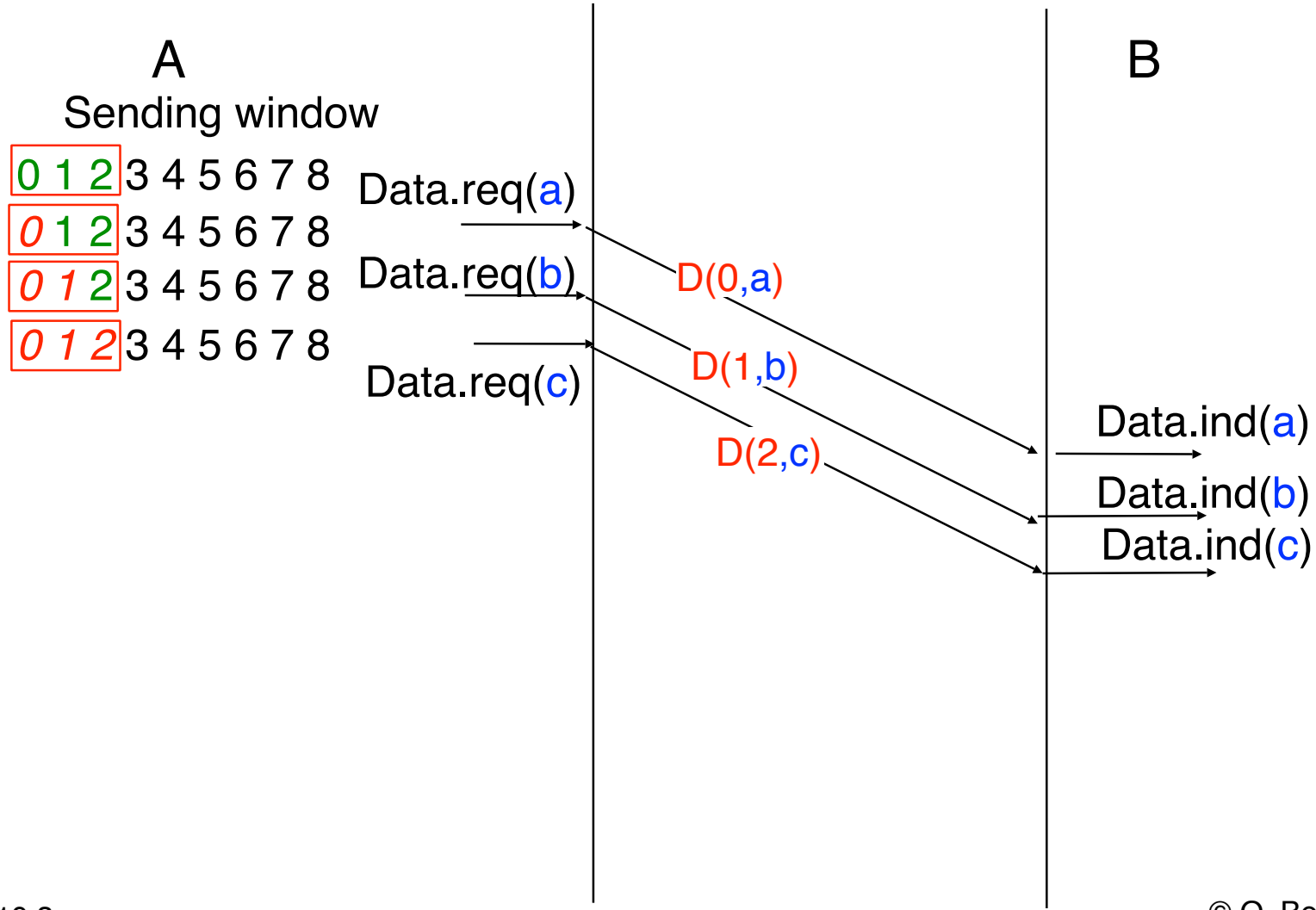
Sliding windows : example

Sending and receiving window : 3 segments



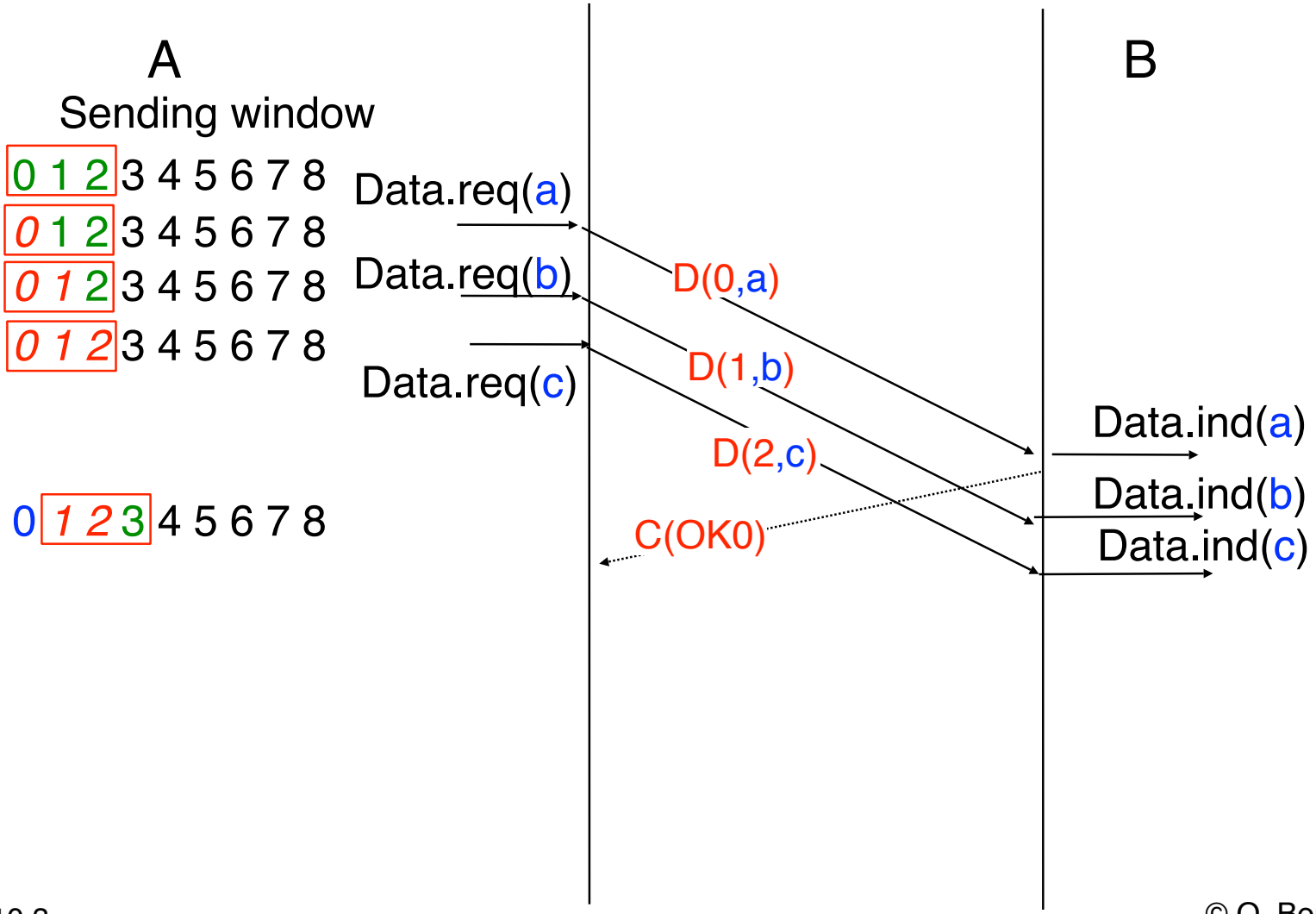
Sliding windows : example

Sending and receiving window : 3 segments



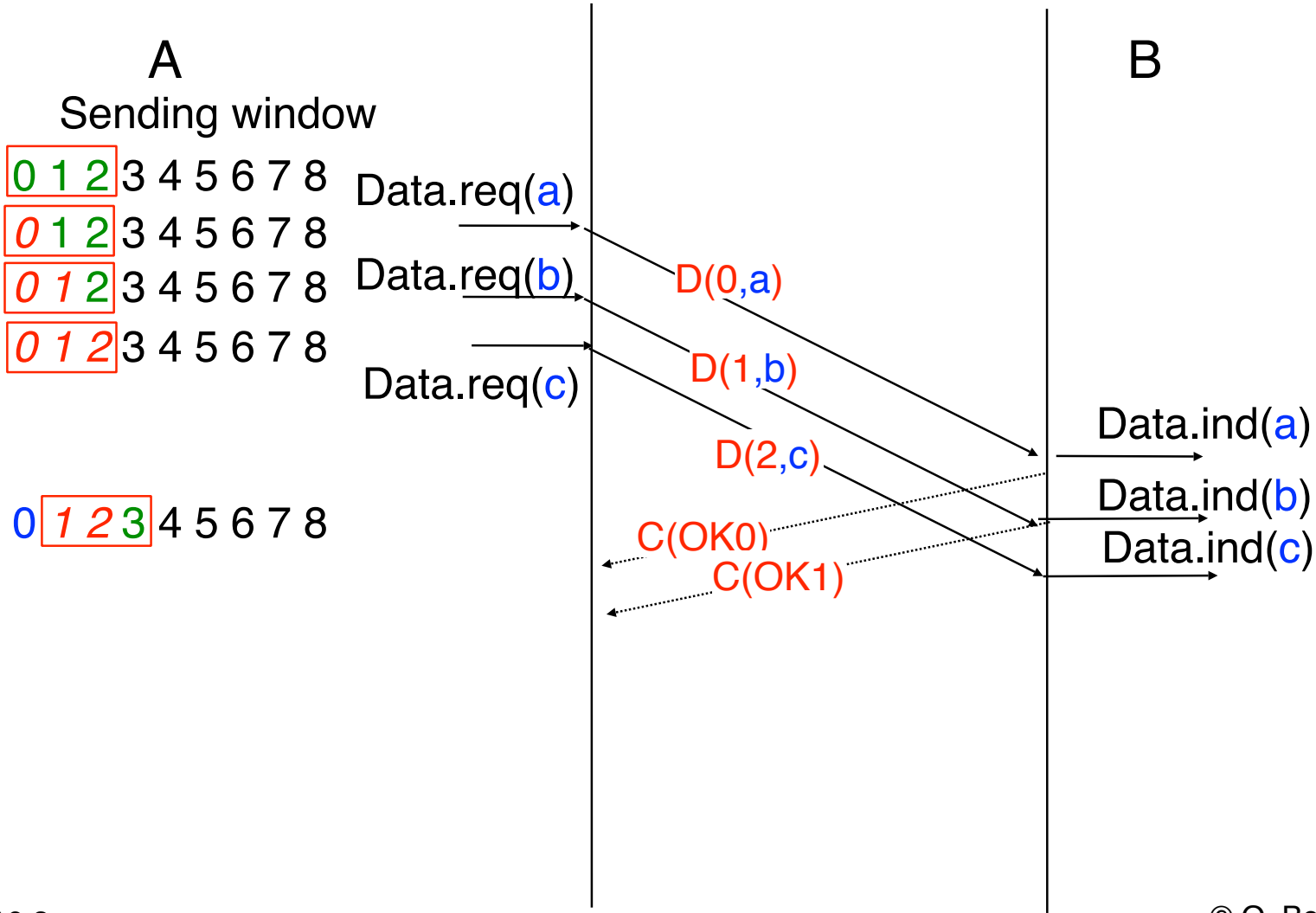
Sliding windows : example

Sending and receiving window : 3 segments



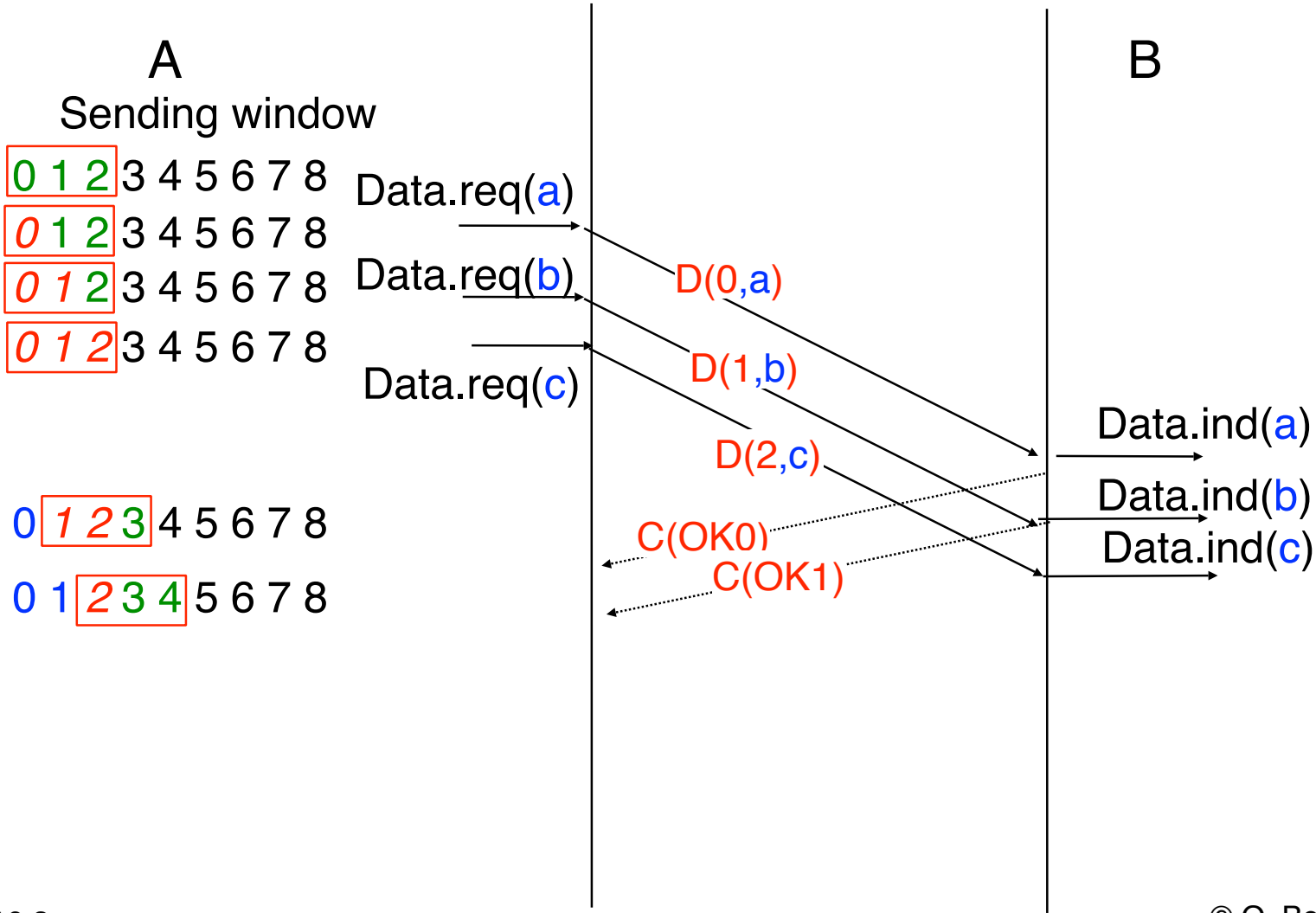
Sliding windows : example

Sending and receiving window : 3 segments



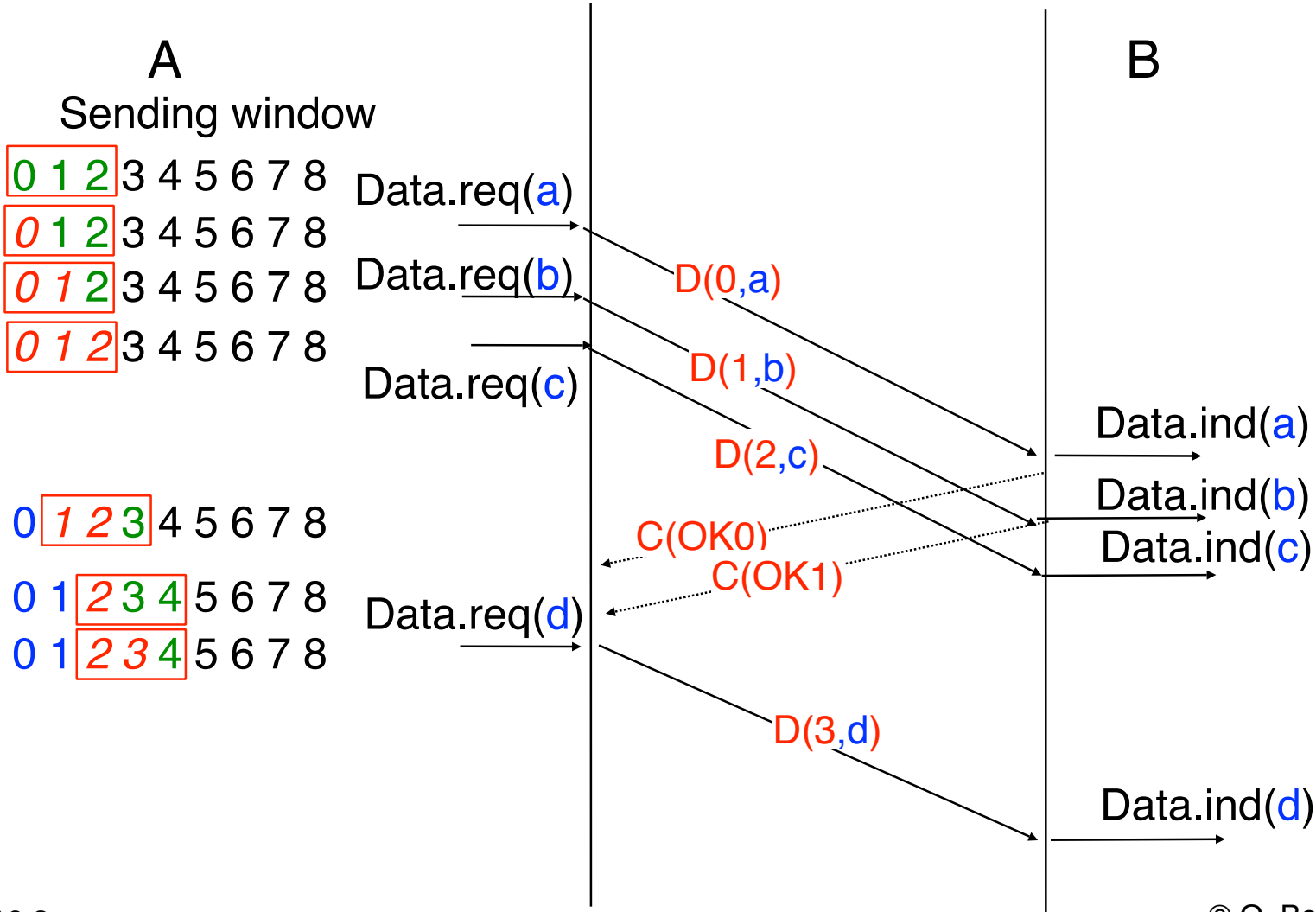
Sliding windows : example

Sending and receiving window : 3 segments



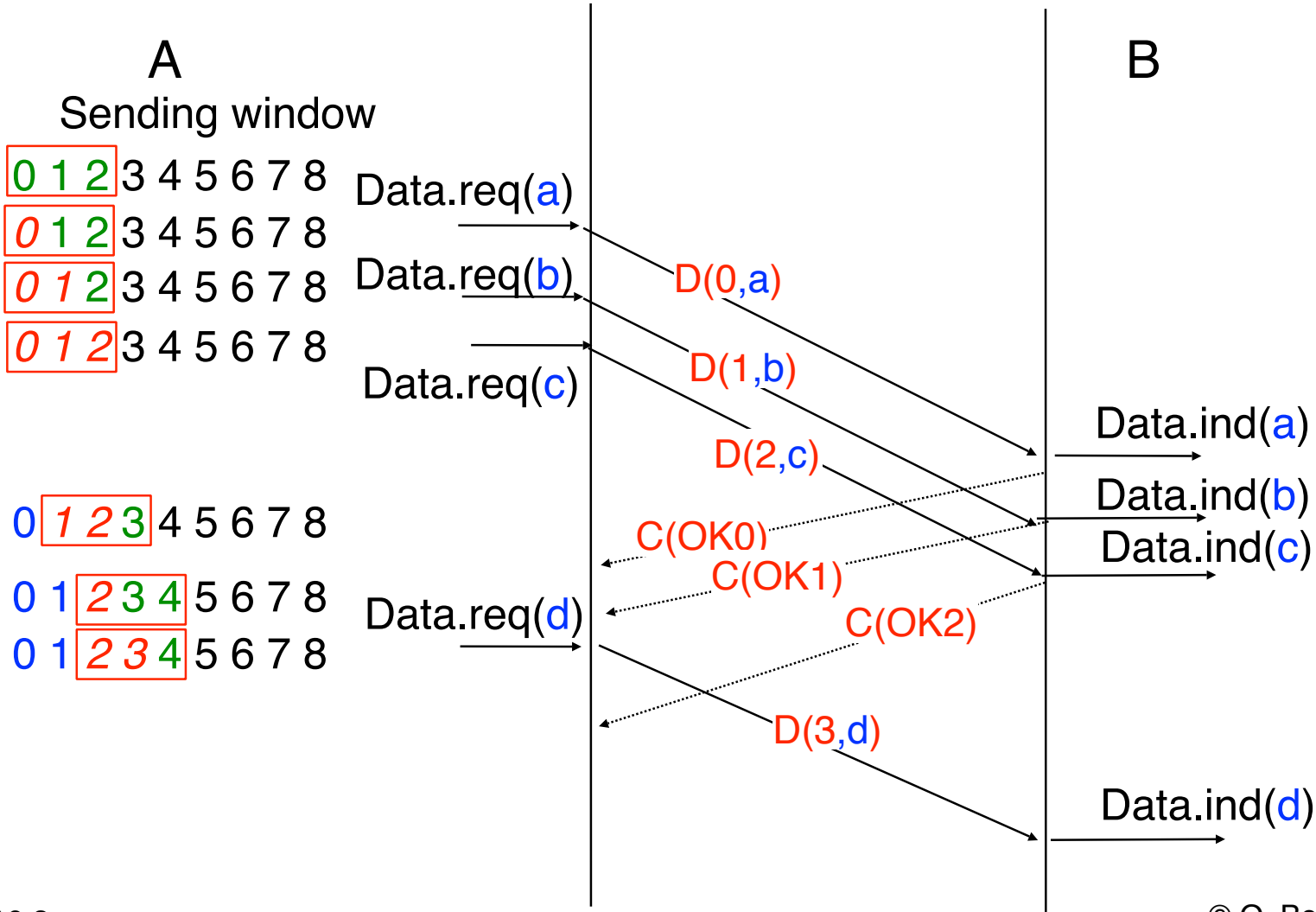
Sliding windows : example

Sending and receiving window : 3 segments



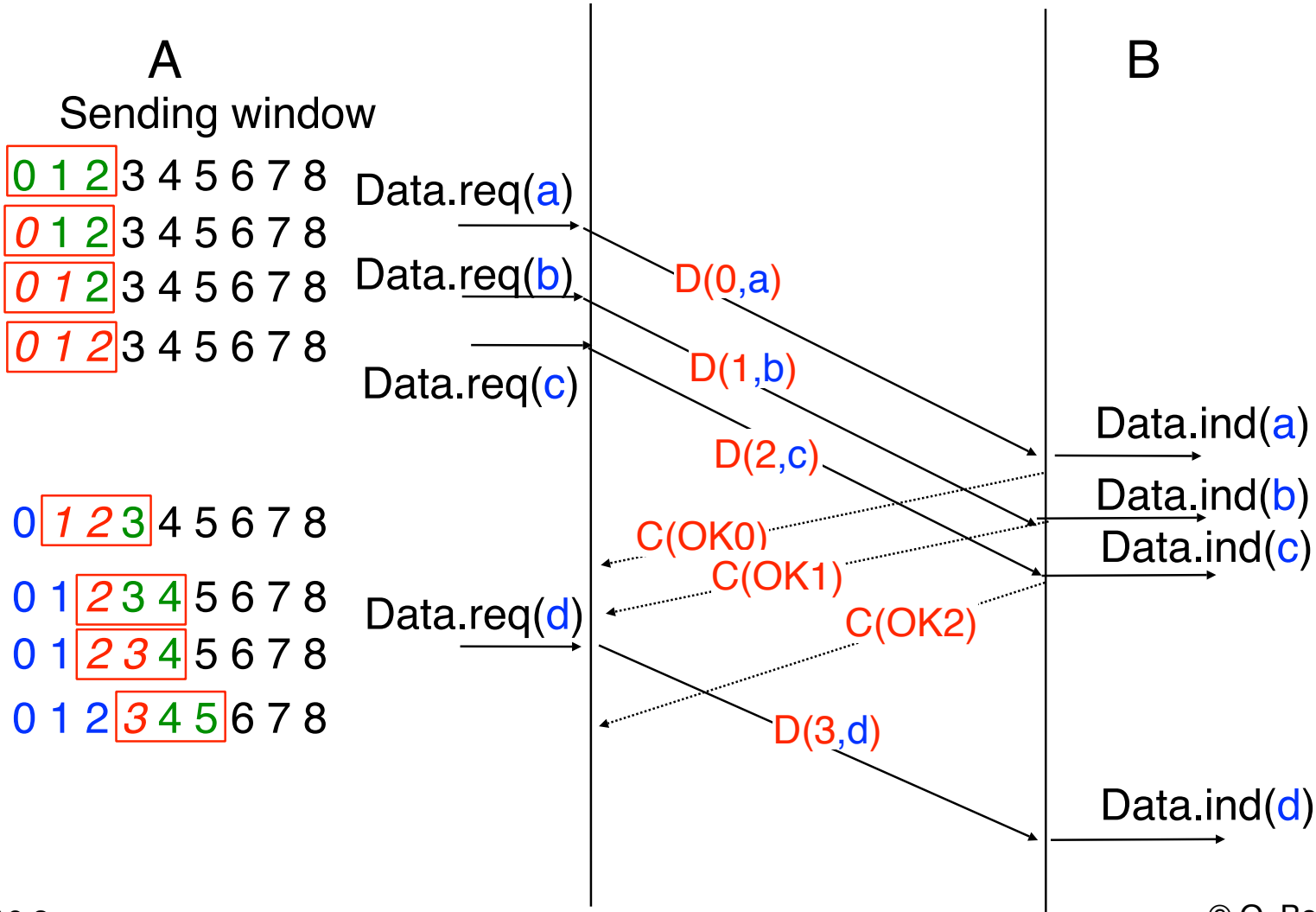
Sliding windows : example

Sending and receiving window : 3 segments



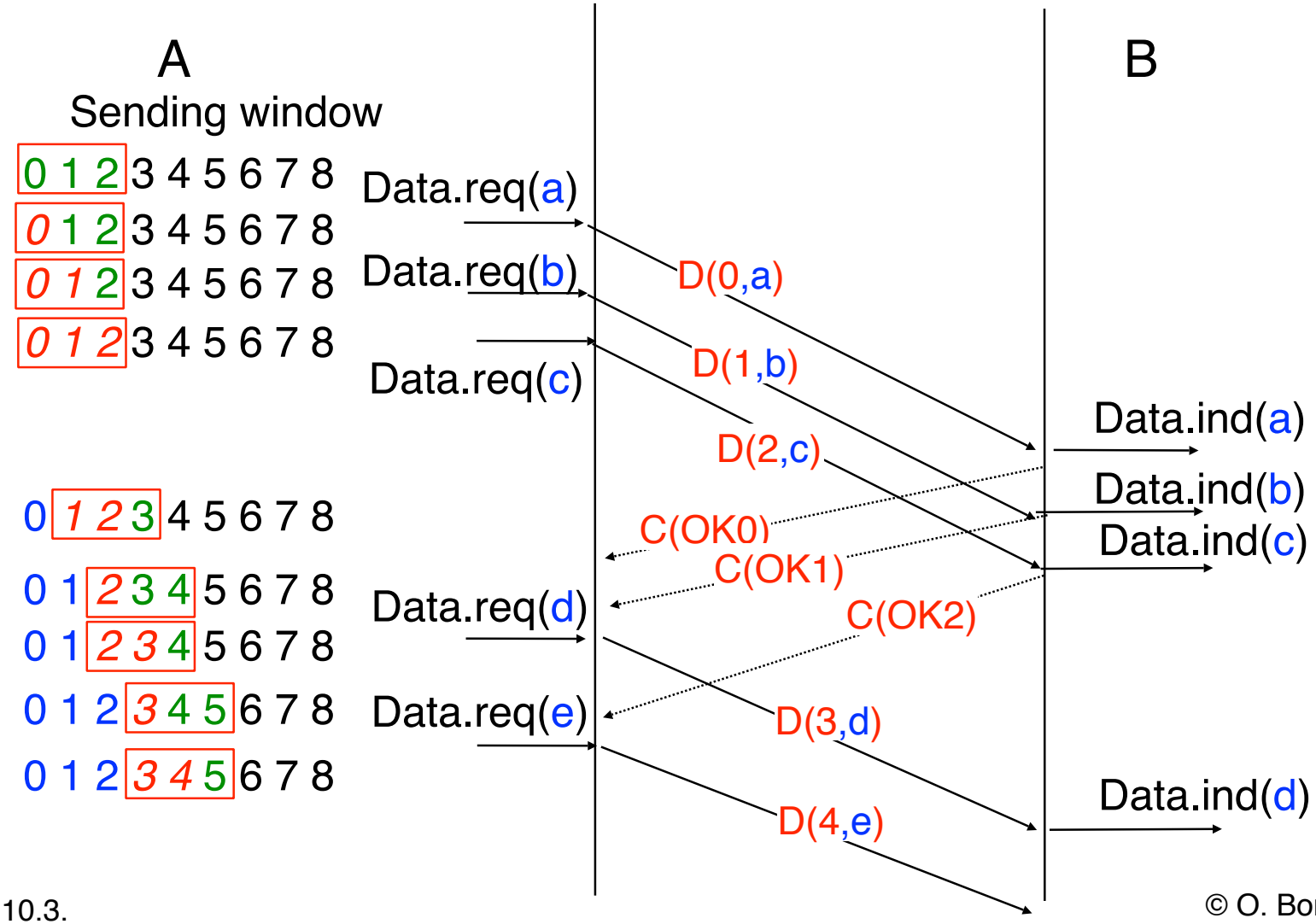
Sliding windows : example

Sending and receiving window : 3 segments



Sliding windows : example

Sending and receiving window : 3 segments



Encoding sequence numbers

Problem

- How many bits do we have in the segment header to encode the sequence number
  - N bits means 2^N different sequence numbers

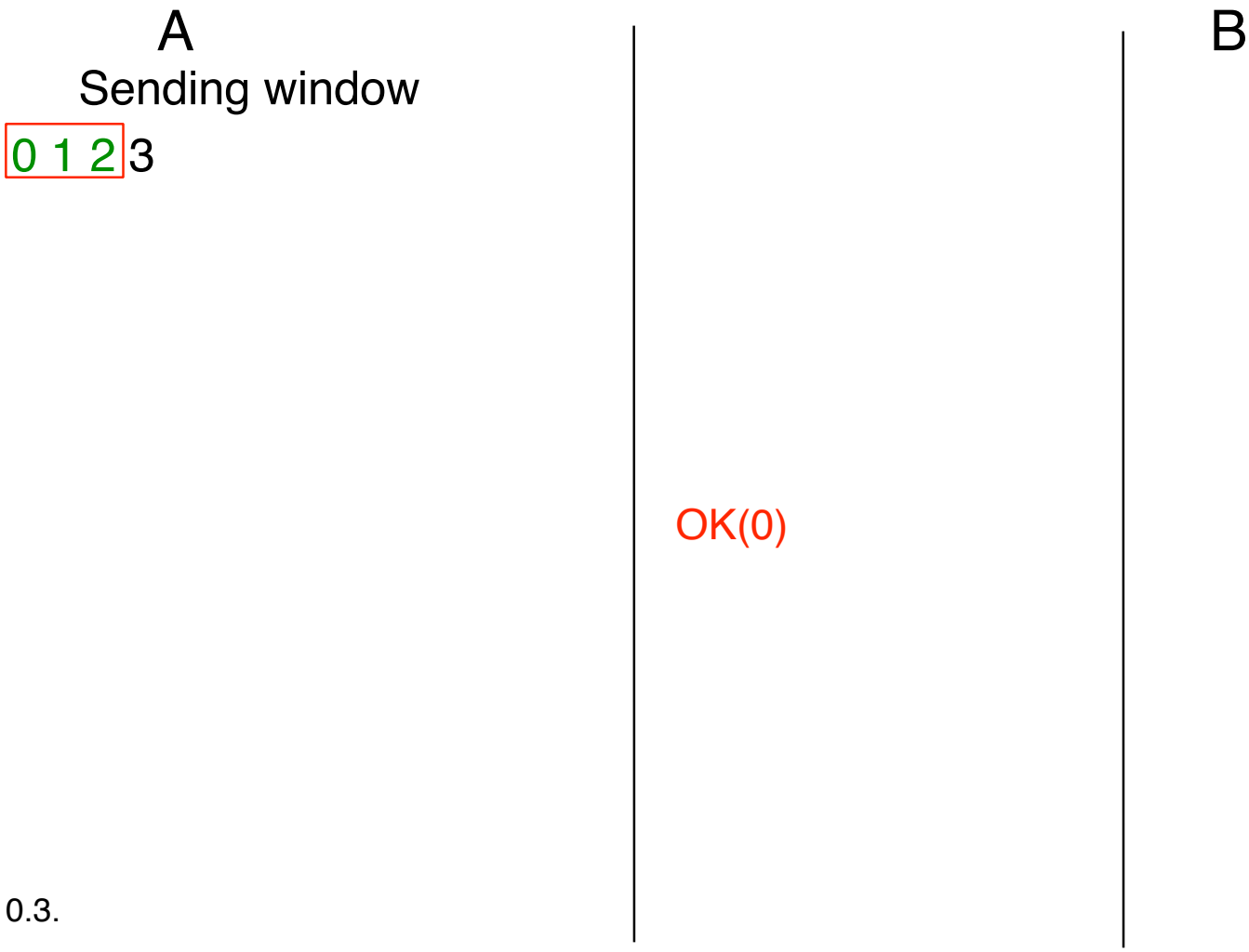
Solution

- place inside each transmitted segment its sequence number modulo 2^N
- The same sequence number will be used for several different segments
  - be careful, this could cause problems...

- Sliding window
  - List of consecutive sequence numbers (modulo 2^N) that the sender is allowed to transmit

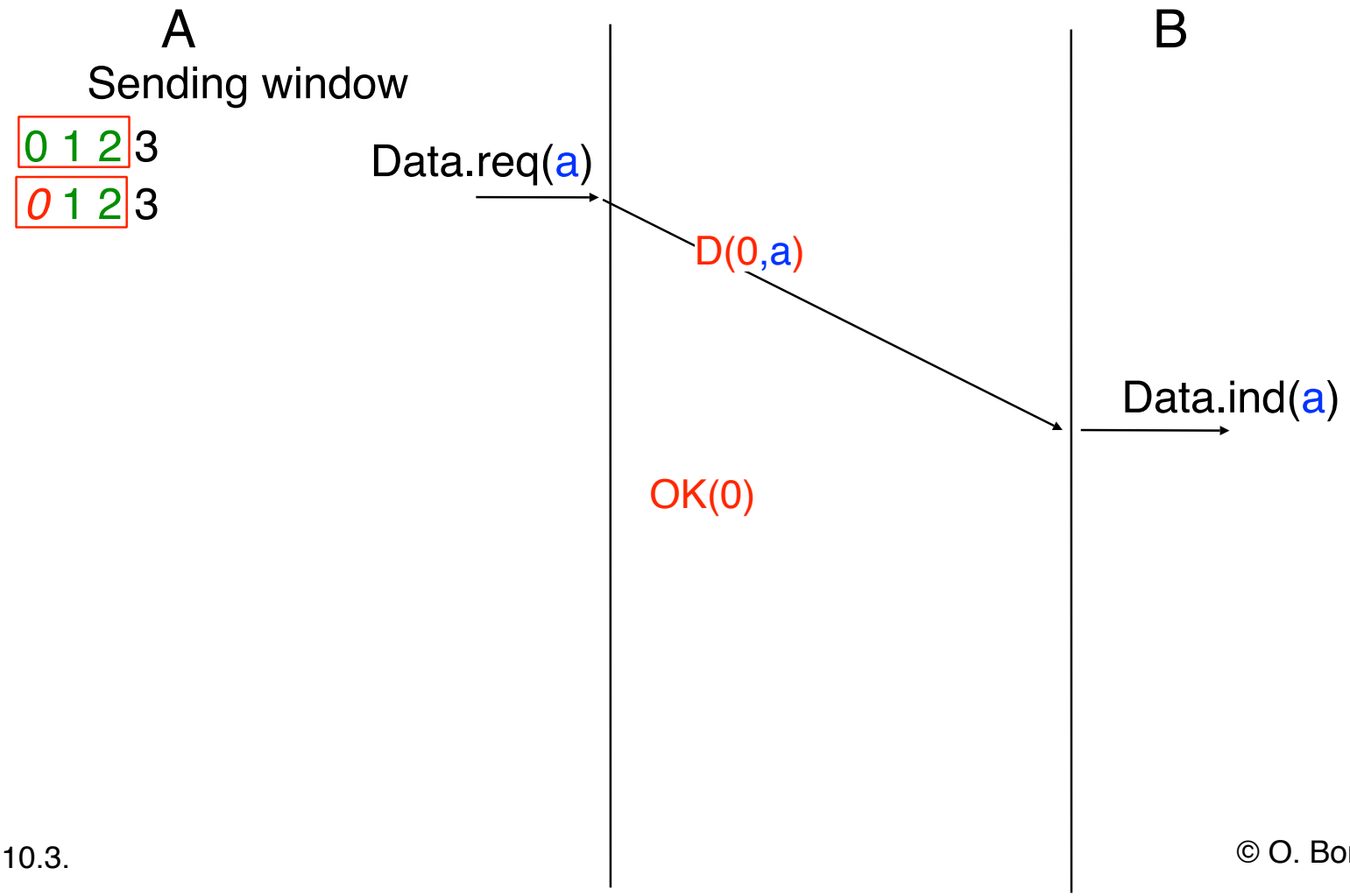
Sliding window : second example

3 segments sending and receiving window
Sequence number encoded as 2 bits field



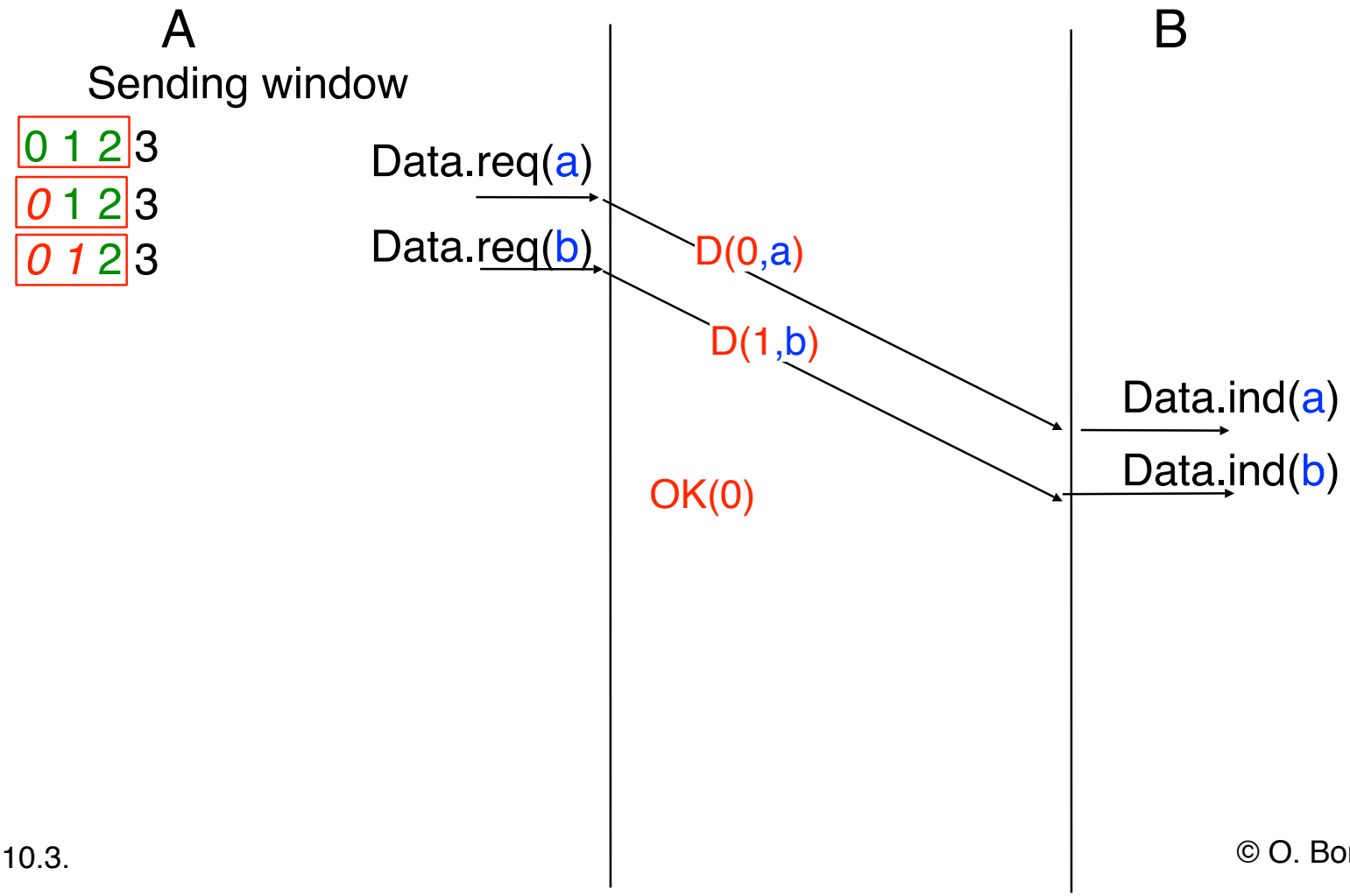
Sliding window : second example

3 segments sending and receiving window
Sequence number encoded as 2 bits field



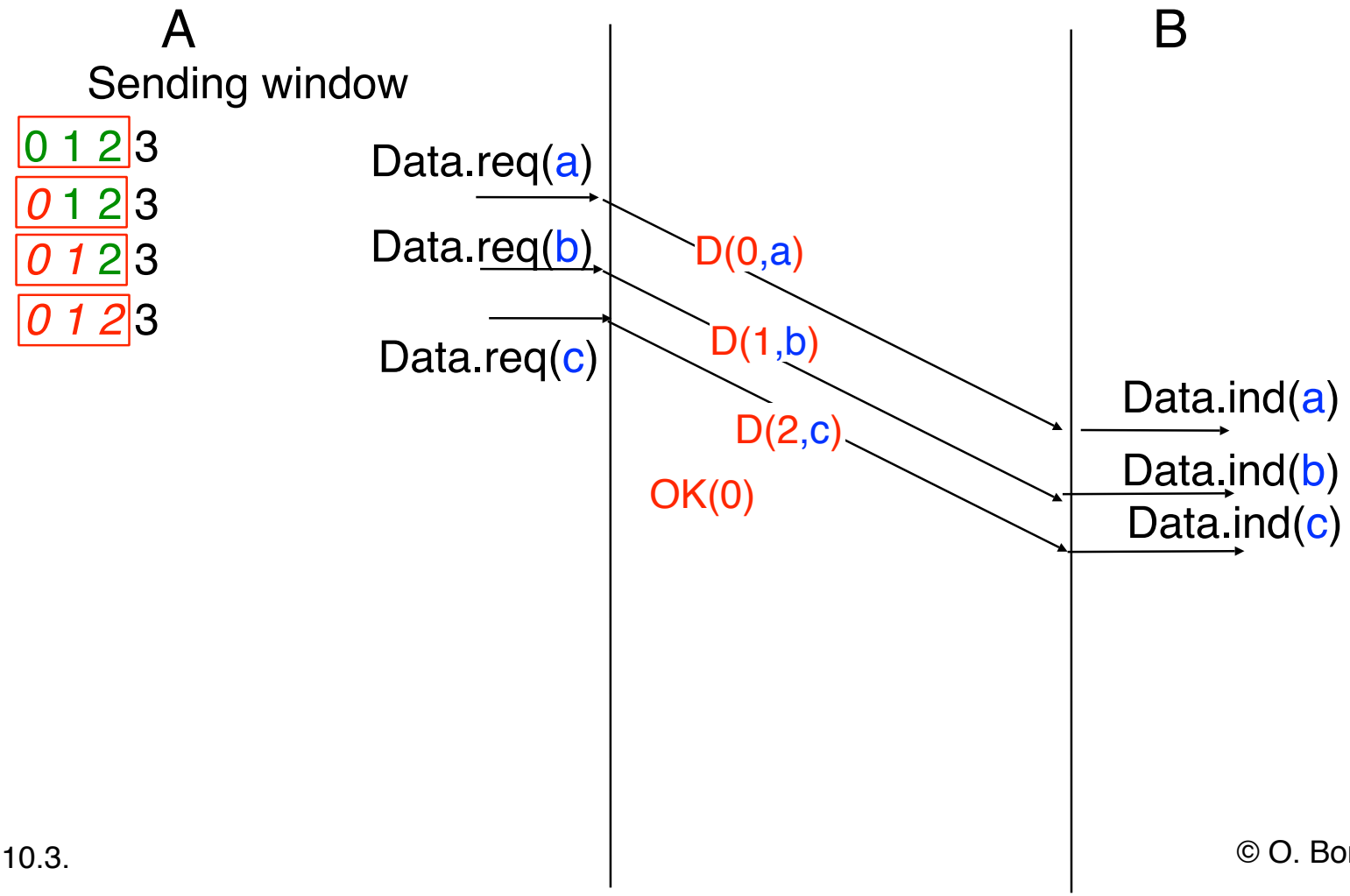
Sliding window : second example

3 segments sending and receiving window
Sequence number encoded as 2 bits field



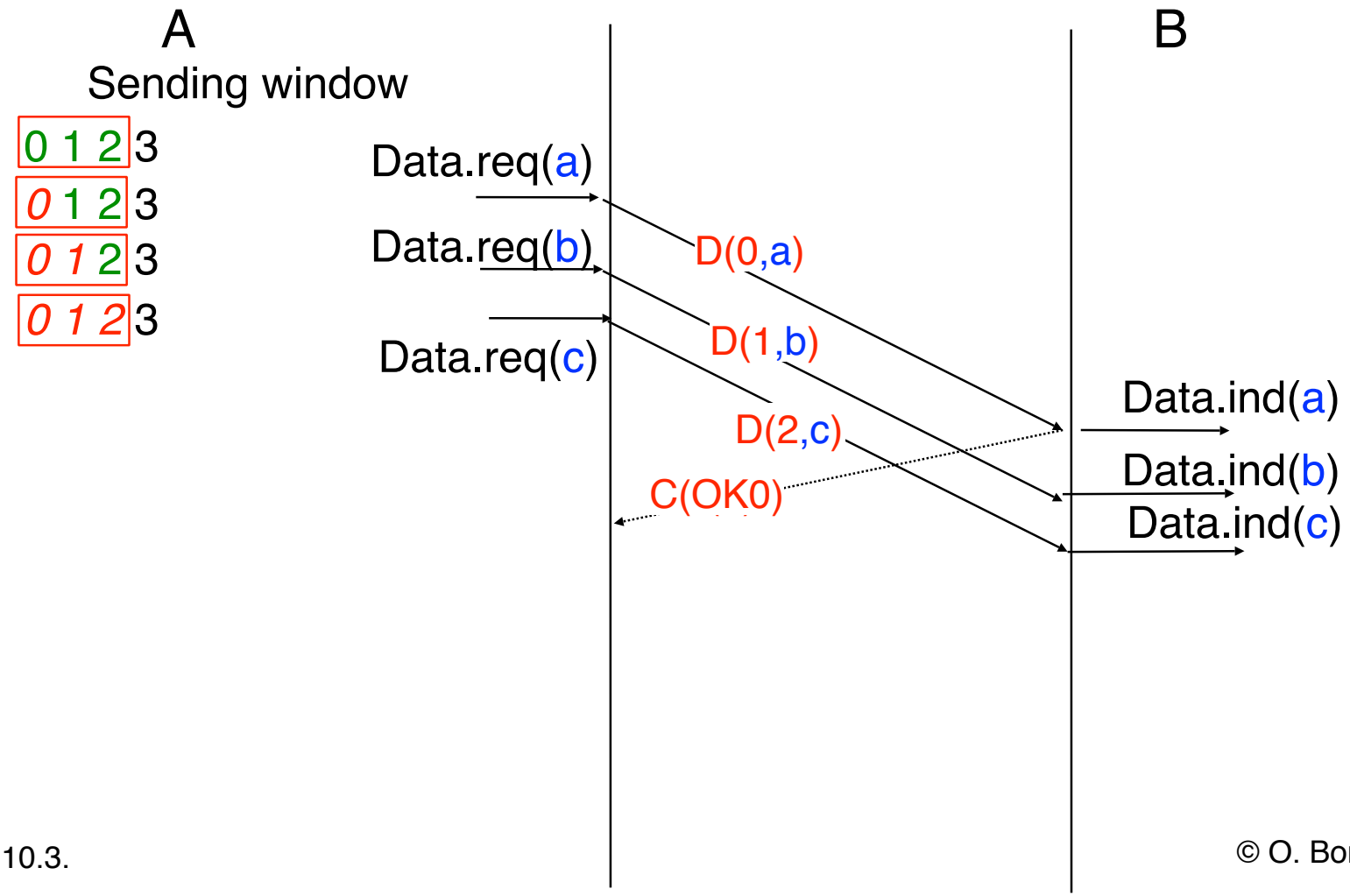
Sliding window : second example

3 segments sending and receiving window
Sequence number encoded as 2 bits field



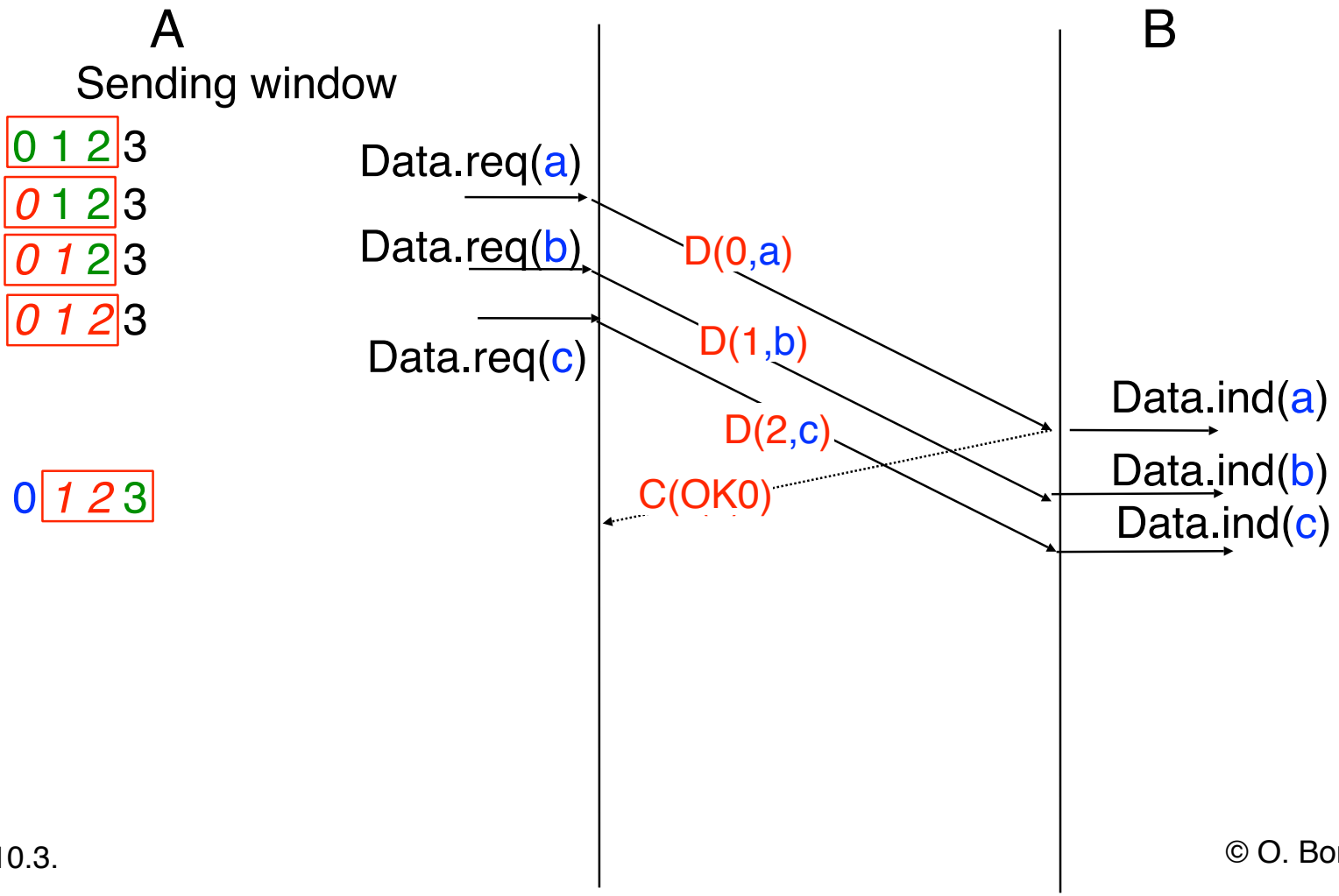
Sliding window : second example

3 segments sending and receiving window
Sequence number encoded as 2 bits field



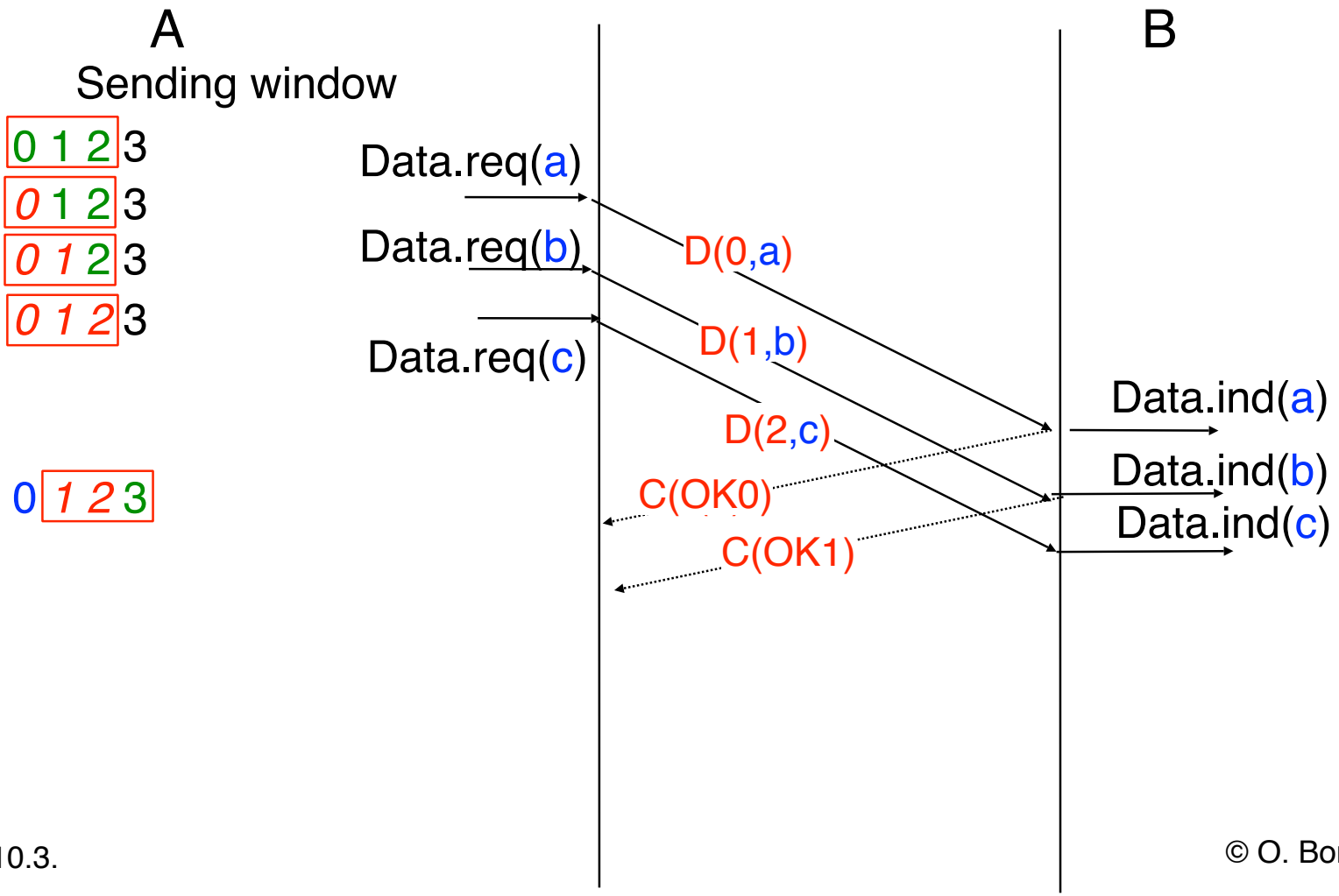
Sliding window : second example

3 segments sending and receiving window
Sequence number encoded as 2 bits field



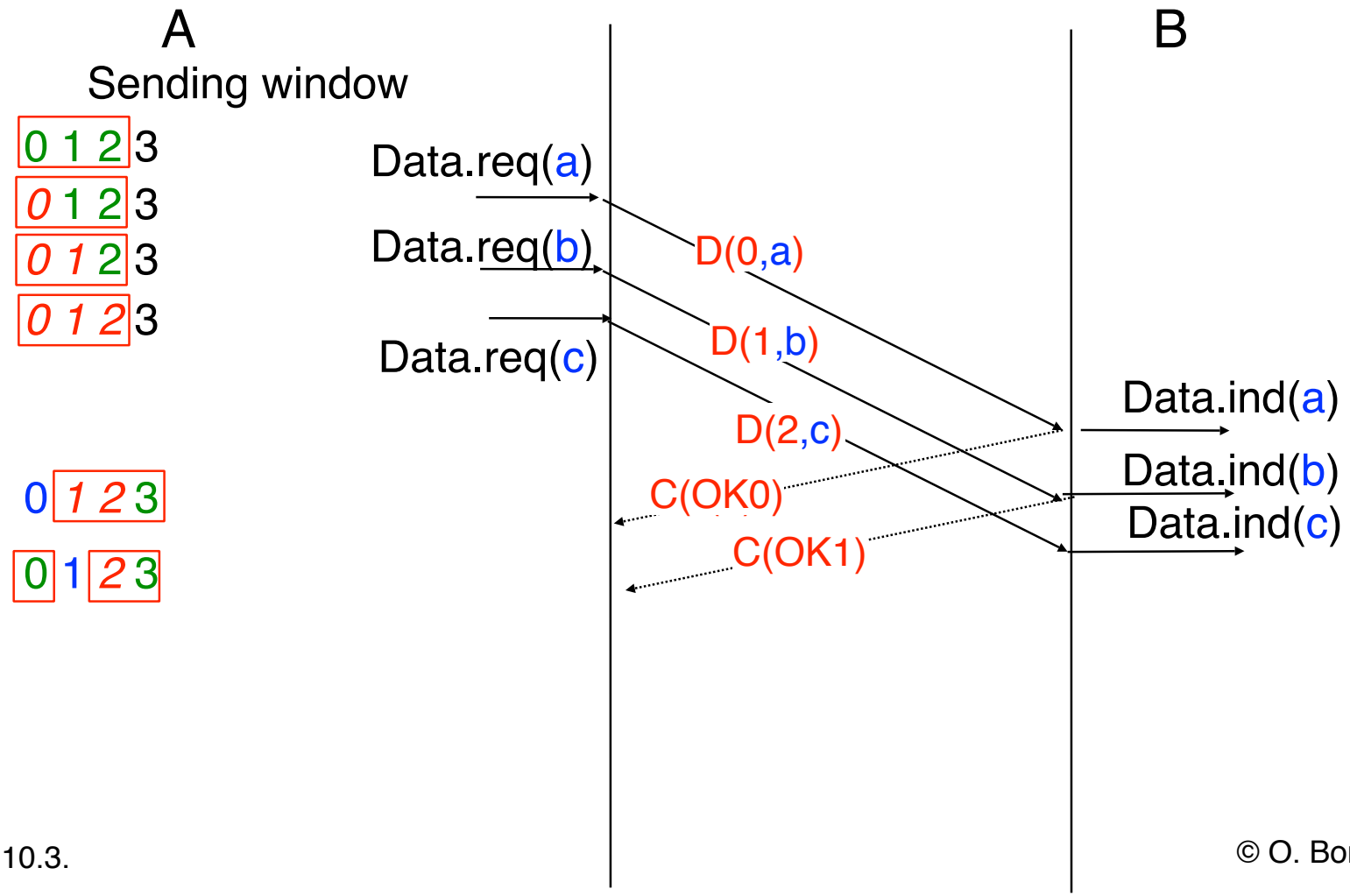
Sliding window : second example

3 segments sending and receiving window
Sequence number encoded as 2 bits field



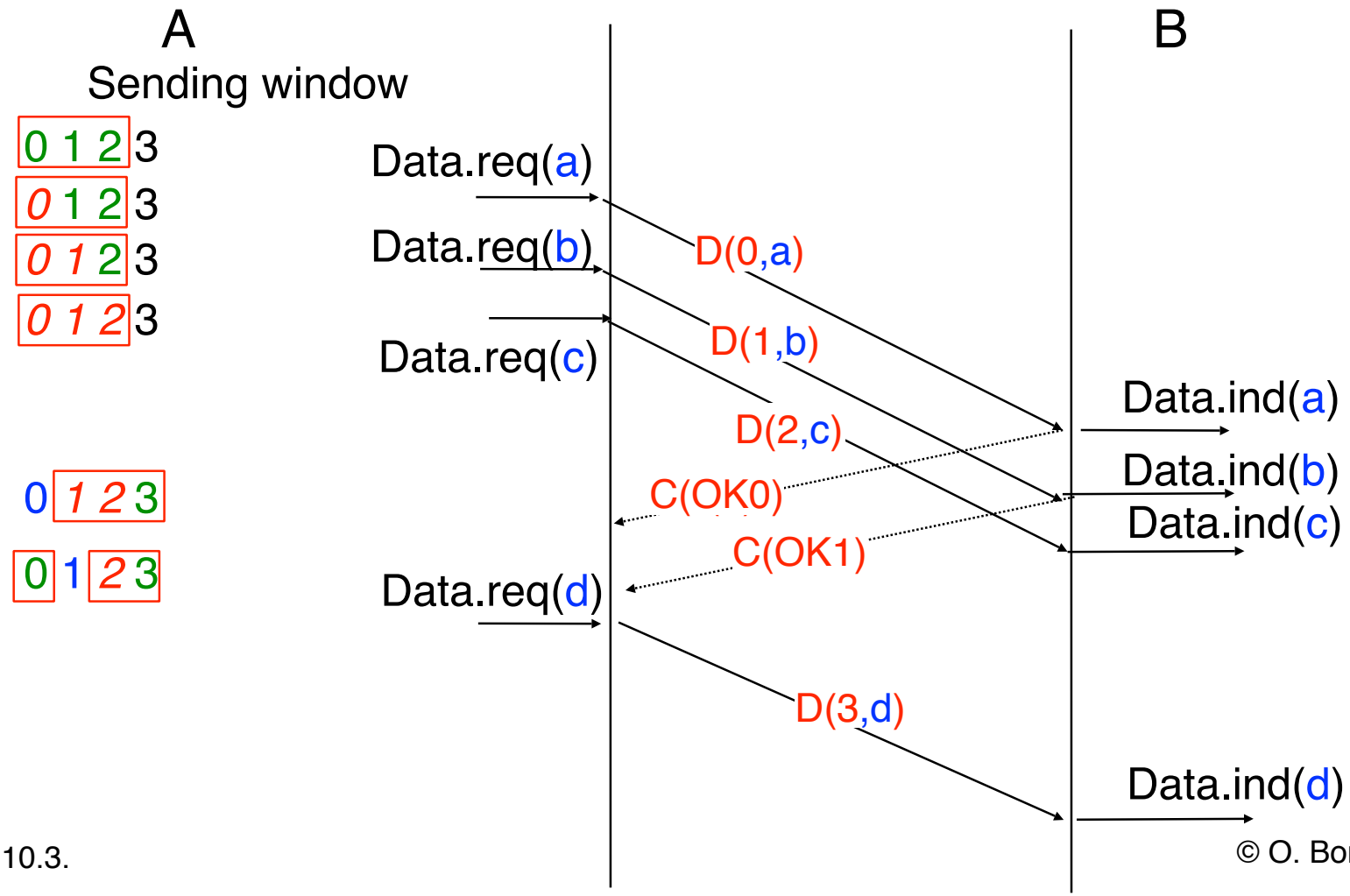
Sliding window : second example

3 segments sending and receiving window
Sequence number encoded as 2 bits field



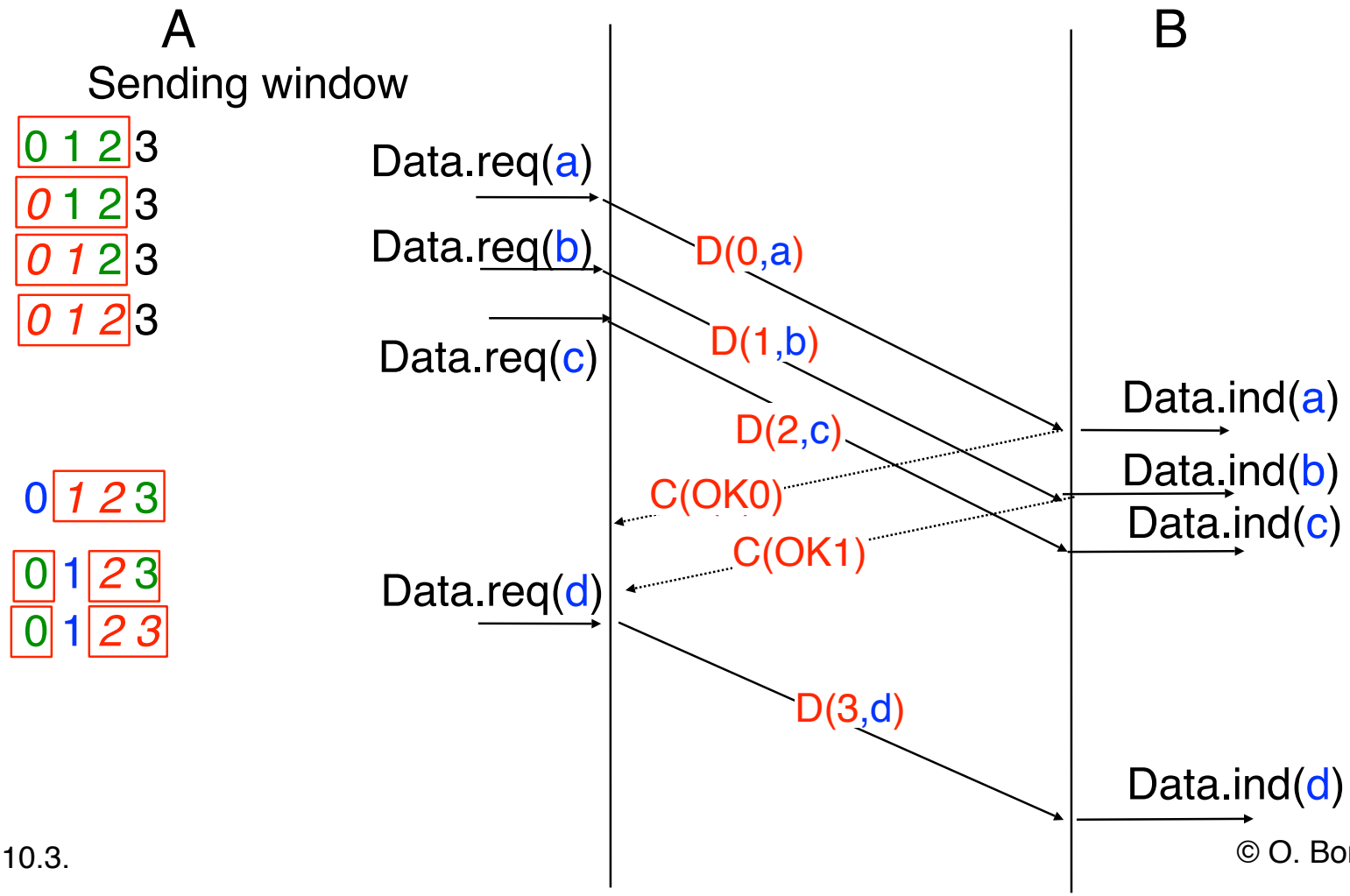
Sliding window : second example

3 segments sending and receiving window
Sequence number encoded as 2 bits field



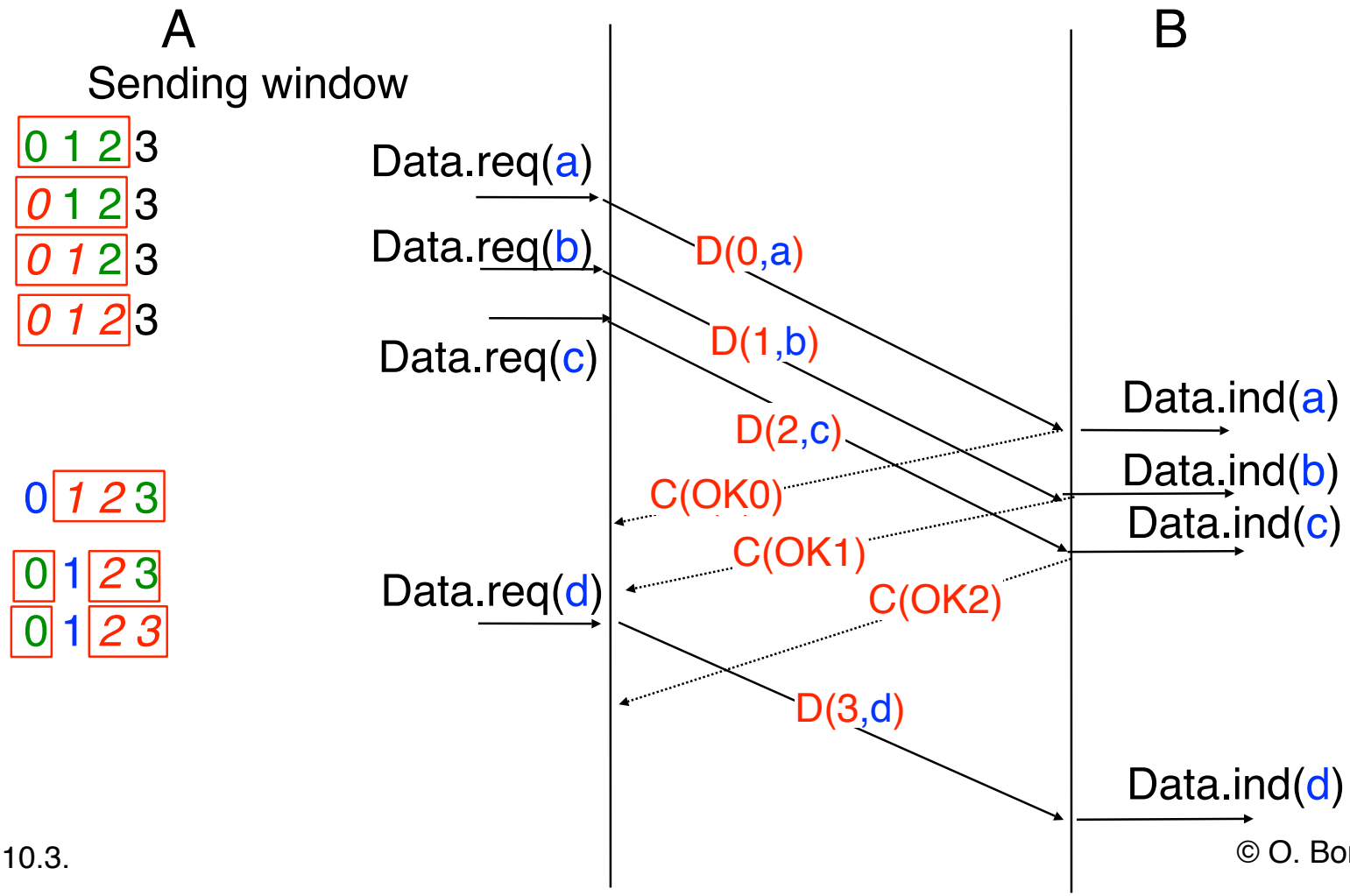
Sliding window : second example

3 segments sending and receiving window
Sequence number encoded as 2 bits field



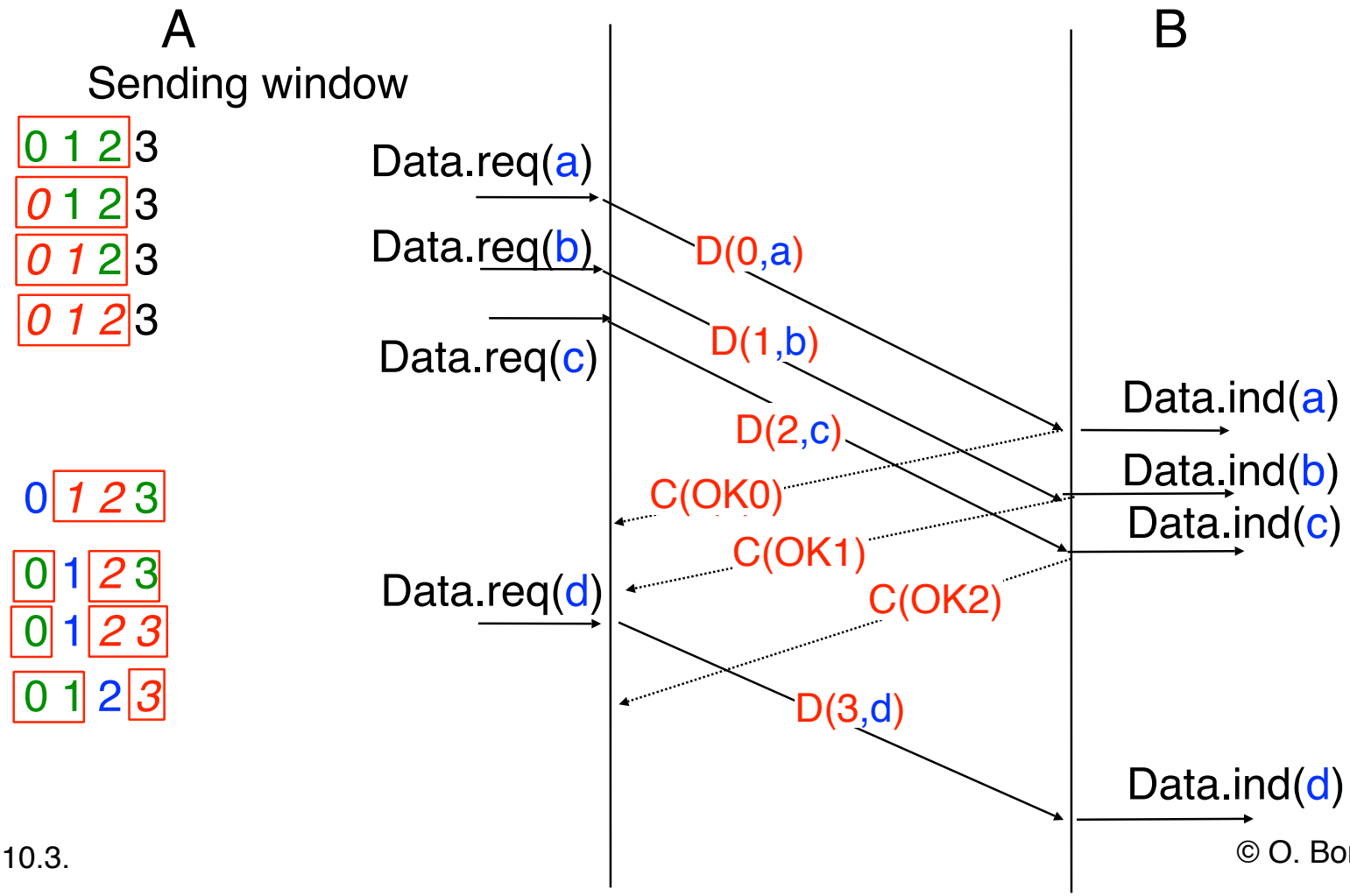
Sliding window : second example

3 segments sending and receiving window
Sequence number encoded as 2 bits field



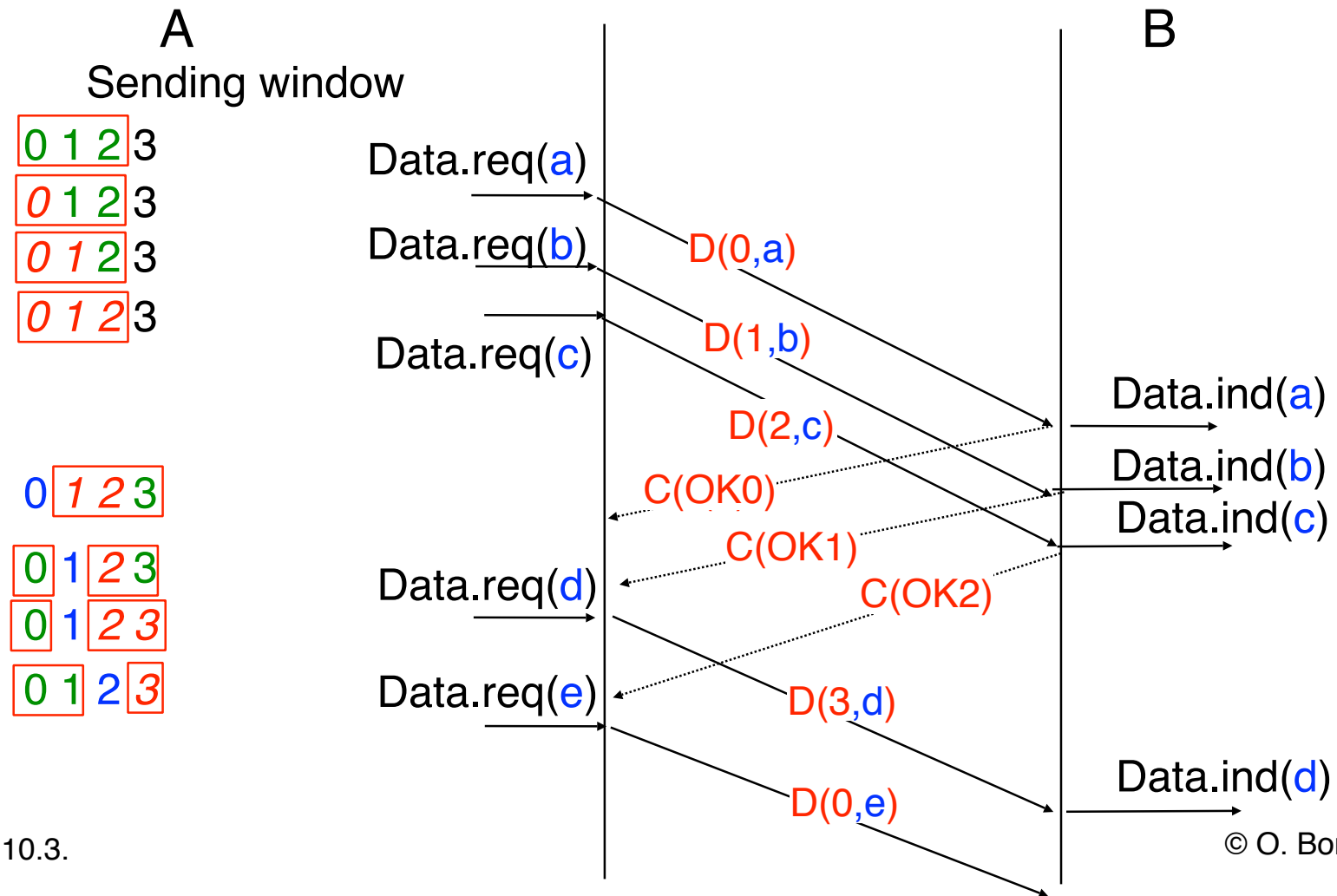
Sliding window : second example

3 segments sending and receiving window
Sequence number encoded as 2 bits field



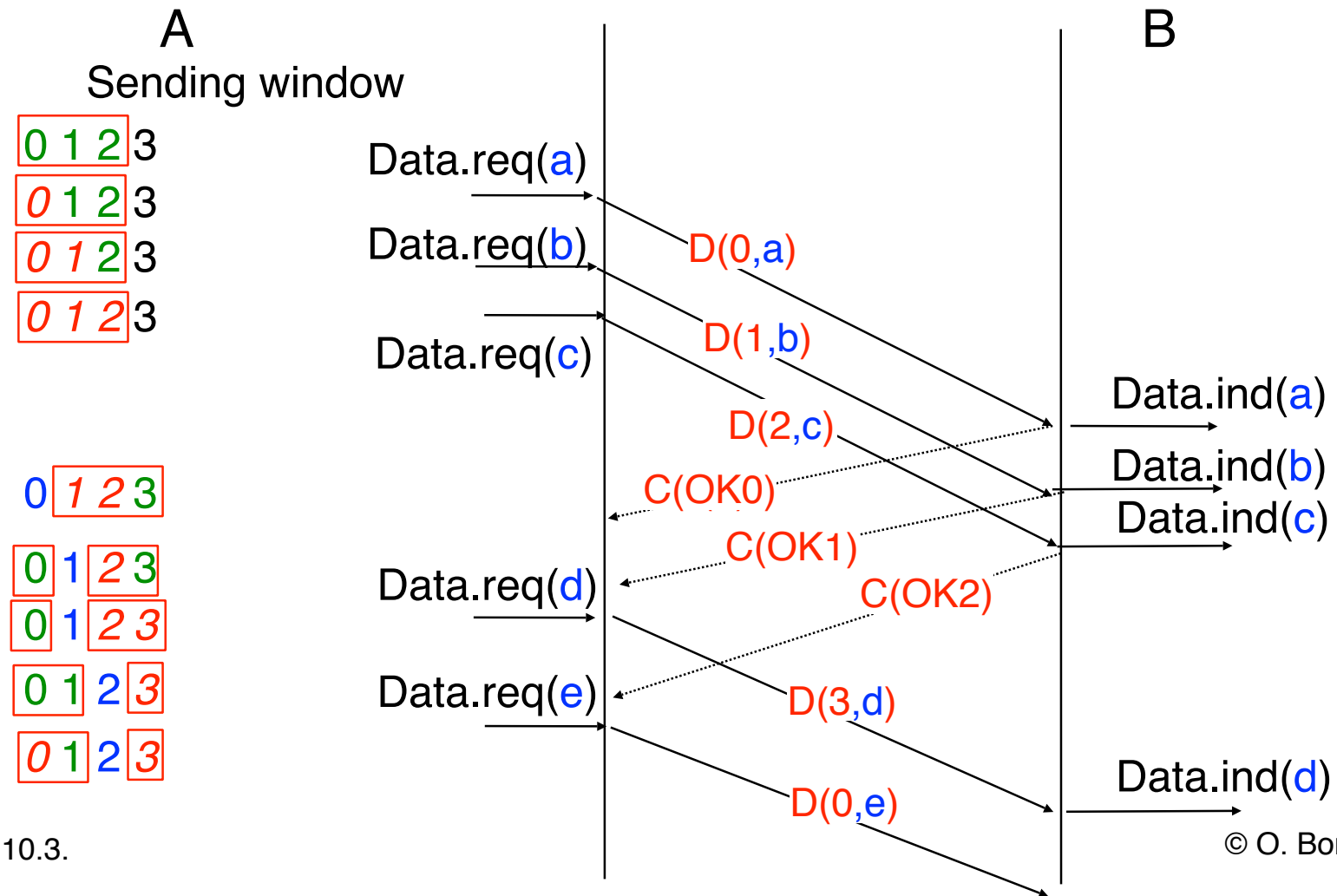
Sliding window : second example

3 segments sending and receiving window
Sequence number encoded as 2 bits field



Sliding window : second example

3 segments sending and receiving window
Sequence number encoded as 2 bits field



Reliable transfer with a sliding window

How to provide a reliable data transfer with a sliding window

- How to react upon reception of a control segment ?
- Sender's and receiver's behaviours

Basic solutions

- Go-Back-N
  - simple implementation, in particular on receiving side
  - throughput will be limited when losses occur
- Selective Repeat
  - more difficult from an implementation viewpoint
  - throughput can remain high when limited losses occur

GO-BACK-N

Principle

- Receiver must be as simple as possible

Receiver

- Only accepts consecutive in-sequence data segments
- Meaning of control segments
  - Upon reception of data segment
    - OKX means that all data segments, up to and including X have been received correctly
    - NAKX means that the data segment whose sequence number is X contained an error or was lost

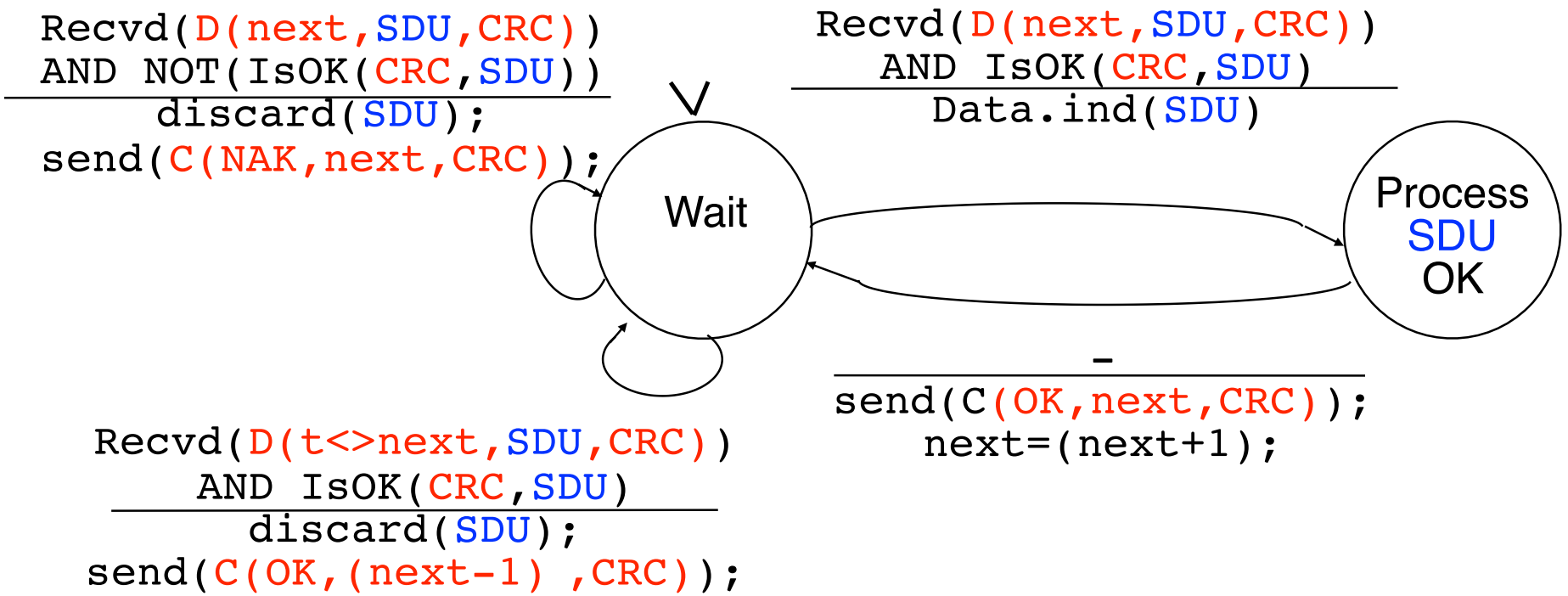
Sender

- Relies on a retransmission timer to detect segment losses
- Upon expiration of retransmission time or arrival of a NAK segment : retransmit all the unacknowledged data segments
  - the sender may thus retransmit a segment that was already received correctly but out-of-sequence at destination

Go-Back-N : Receiver

State variable

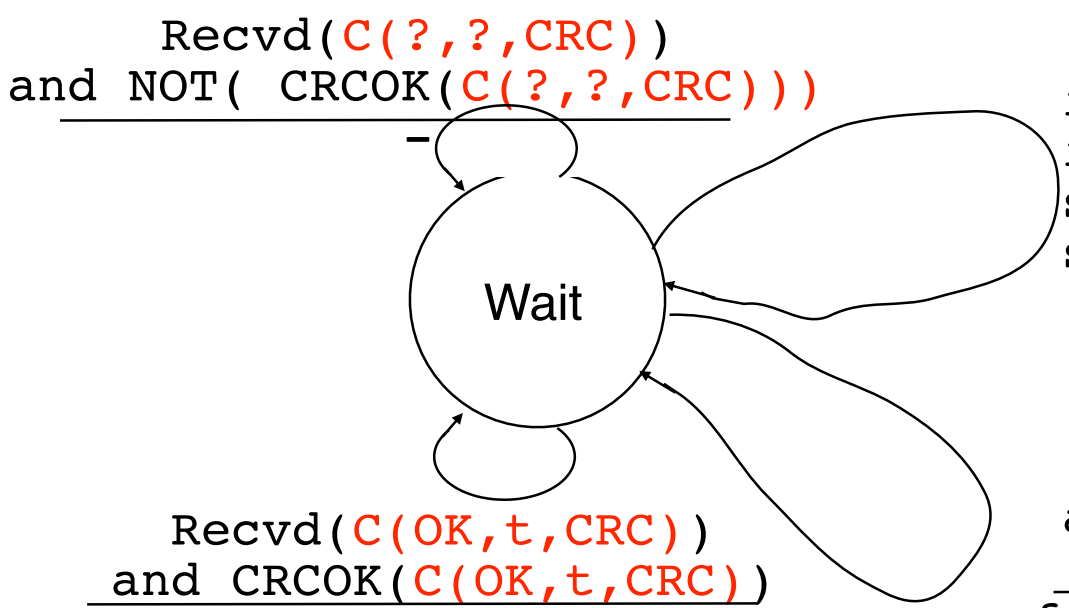
`next` : sequence number of expected data segment



Go-Back-N : Sender

State variables

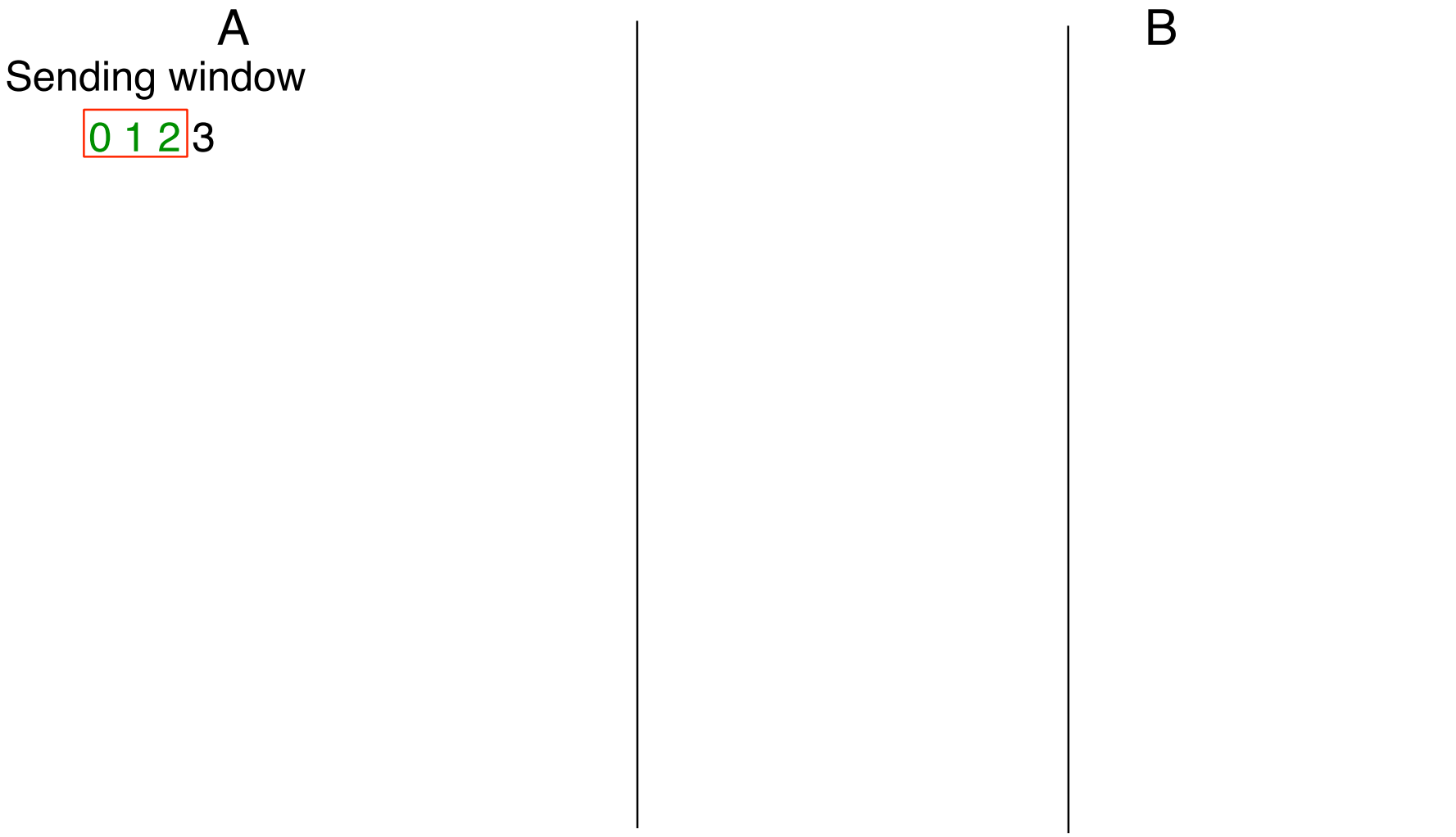
`base` : sequence number of oldest data segment
`seq` : first available sequence number
`W` : size of sending window



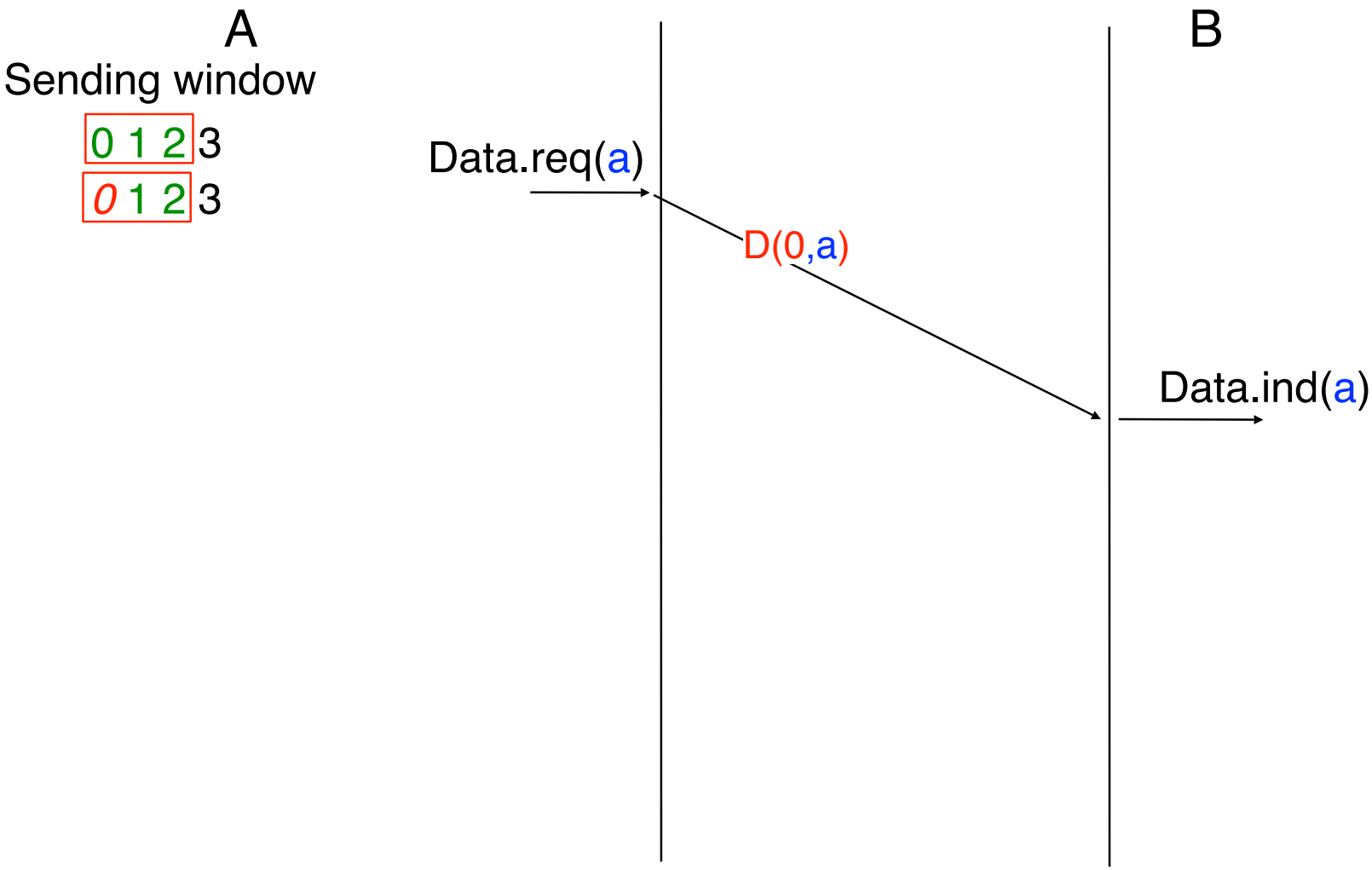

```
      Data.req(SDU)
    AND ( seq < (base+w) )
if (seq==base) { start_timer;}
insert_in_buffer(SDU);
send(D(seq,SDU,CRC));
seq=seq+1 ;
```

```
  [ 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();
```

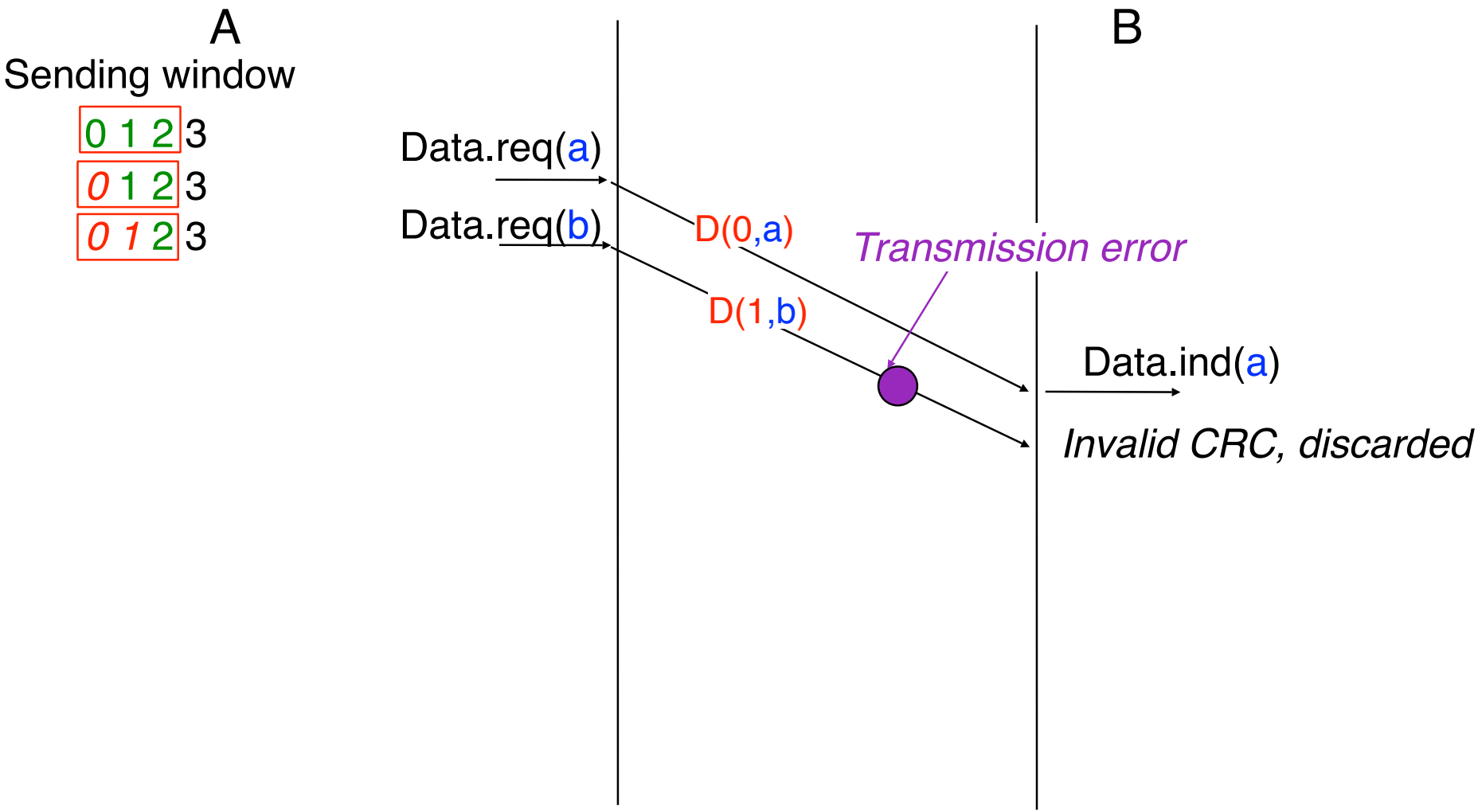
Go-Back-N : Example



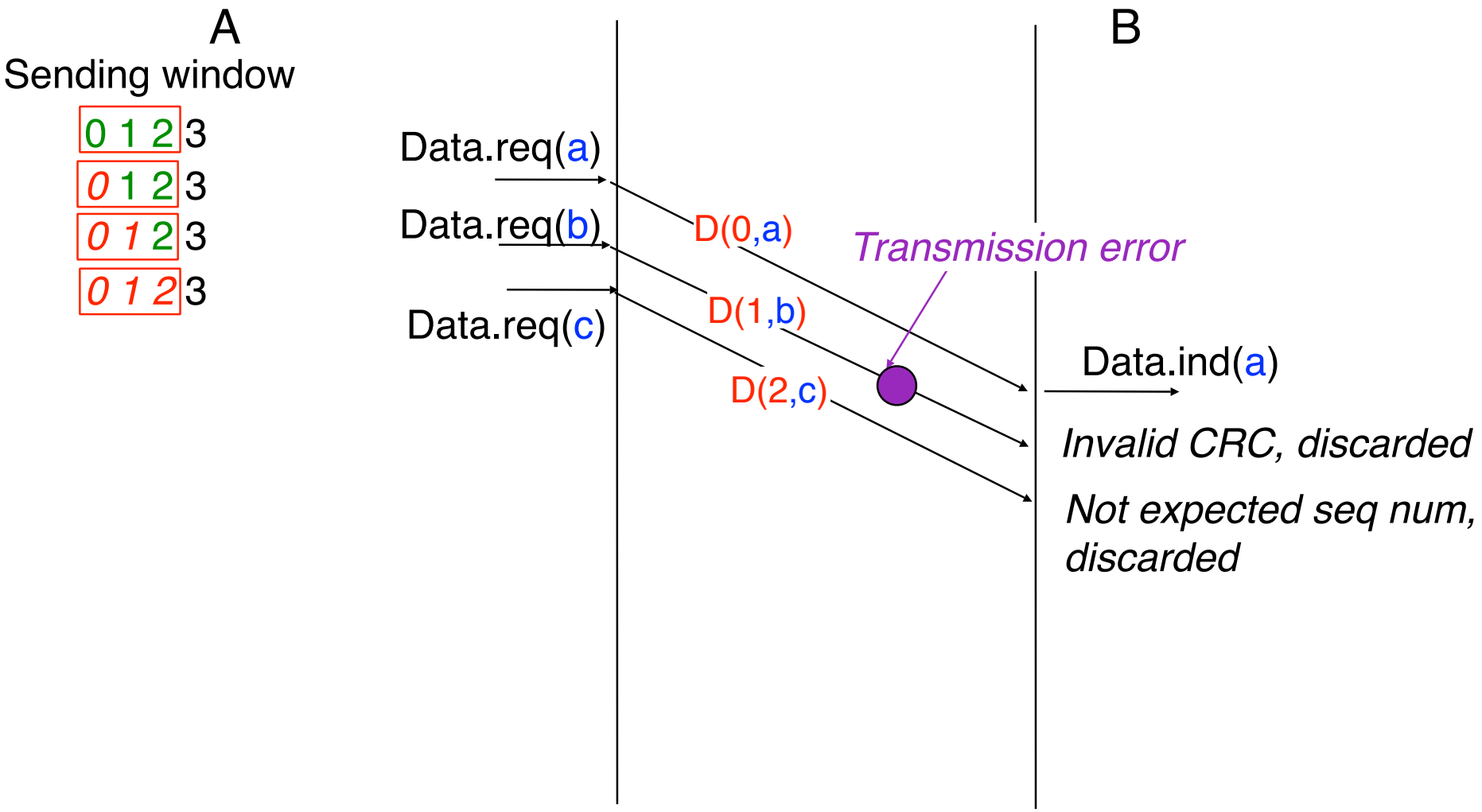
Go-Back-N : Example



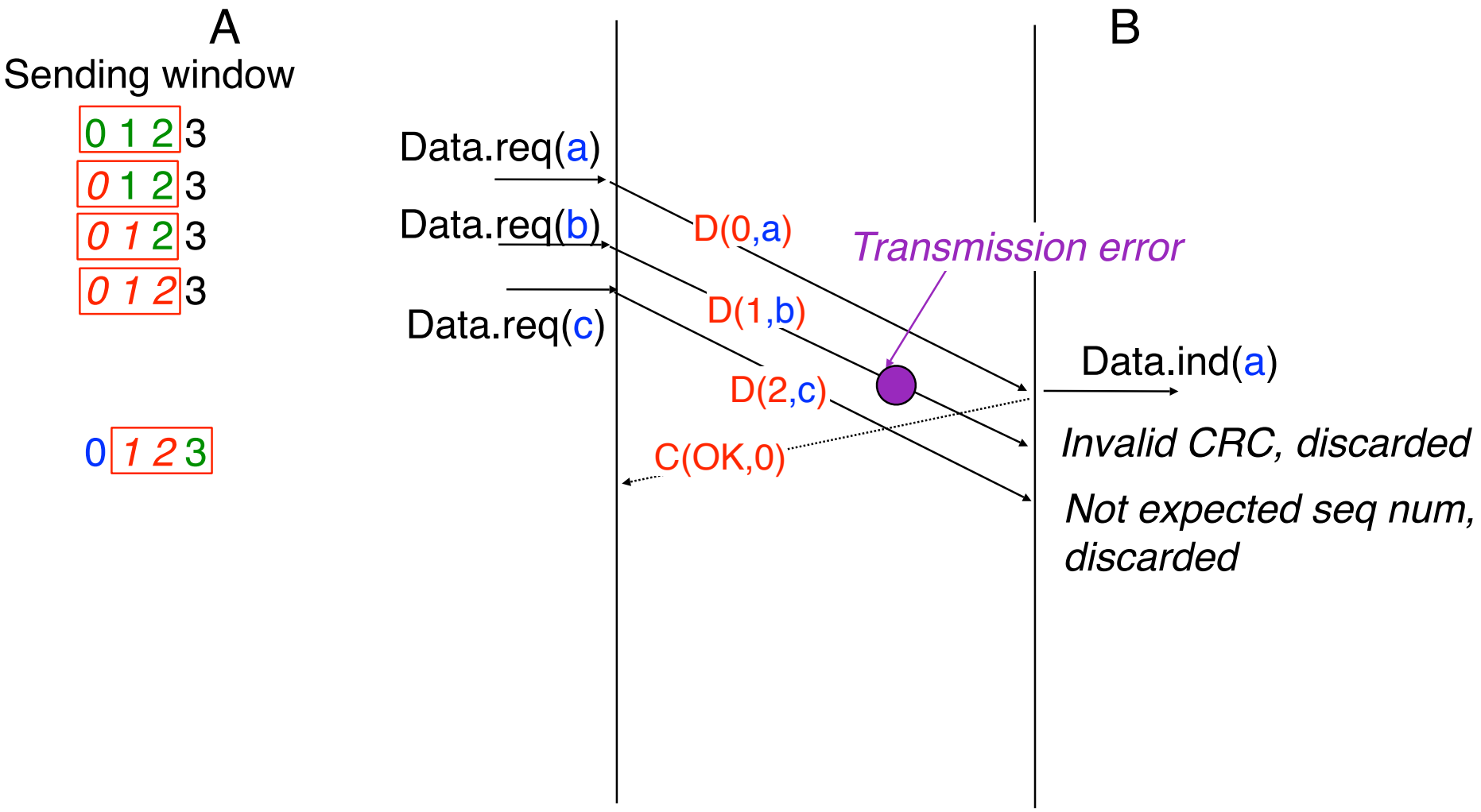
Go-Back-N : Example



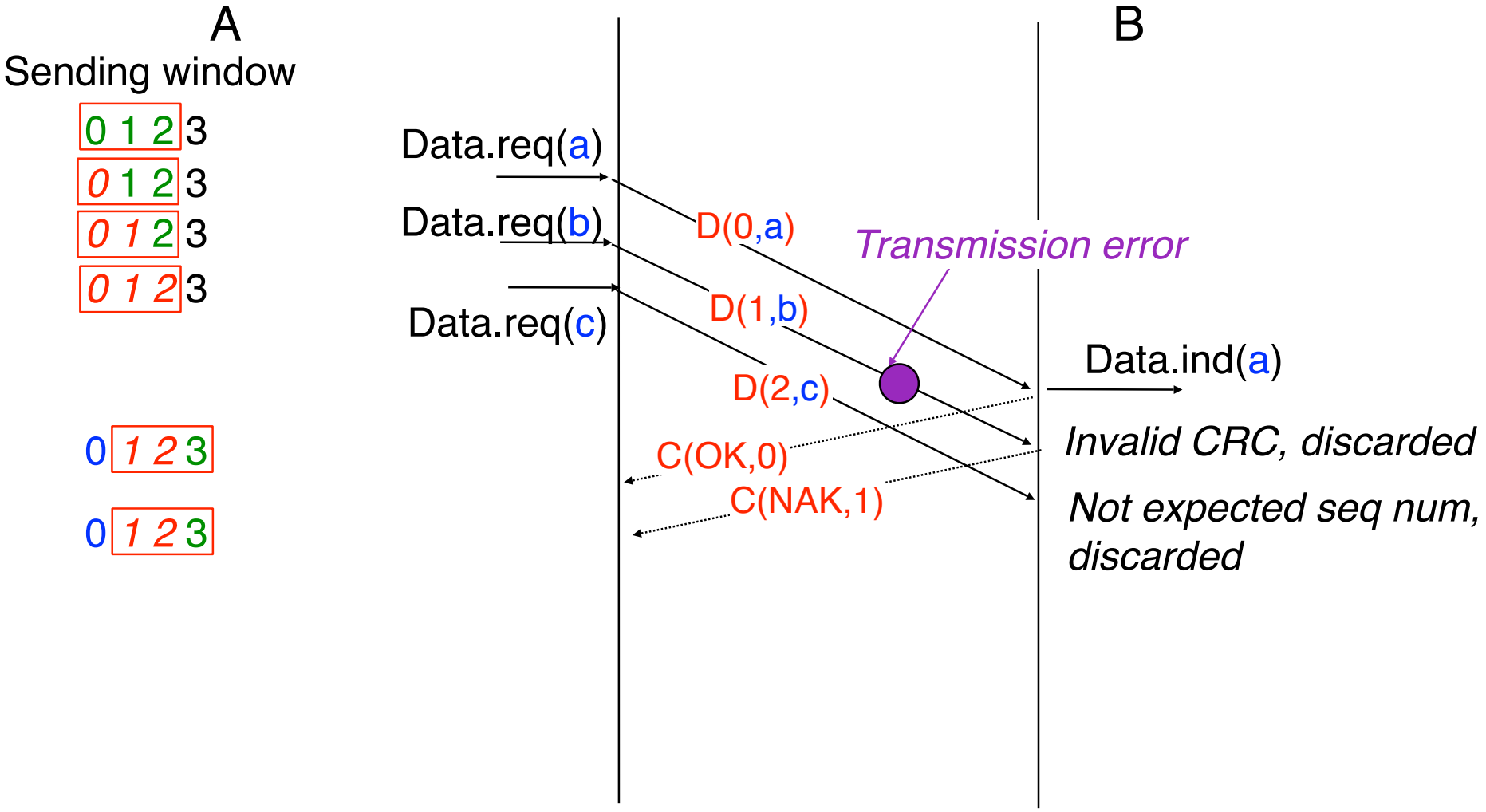
Go-Back-N : Example



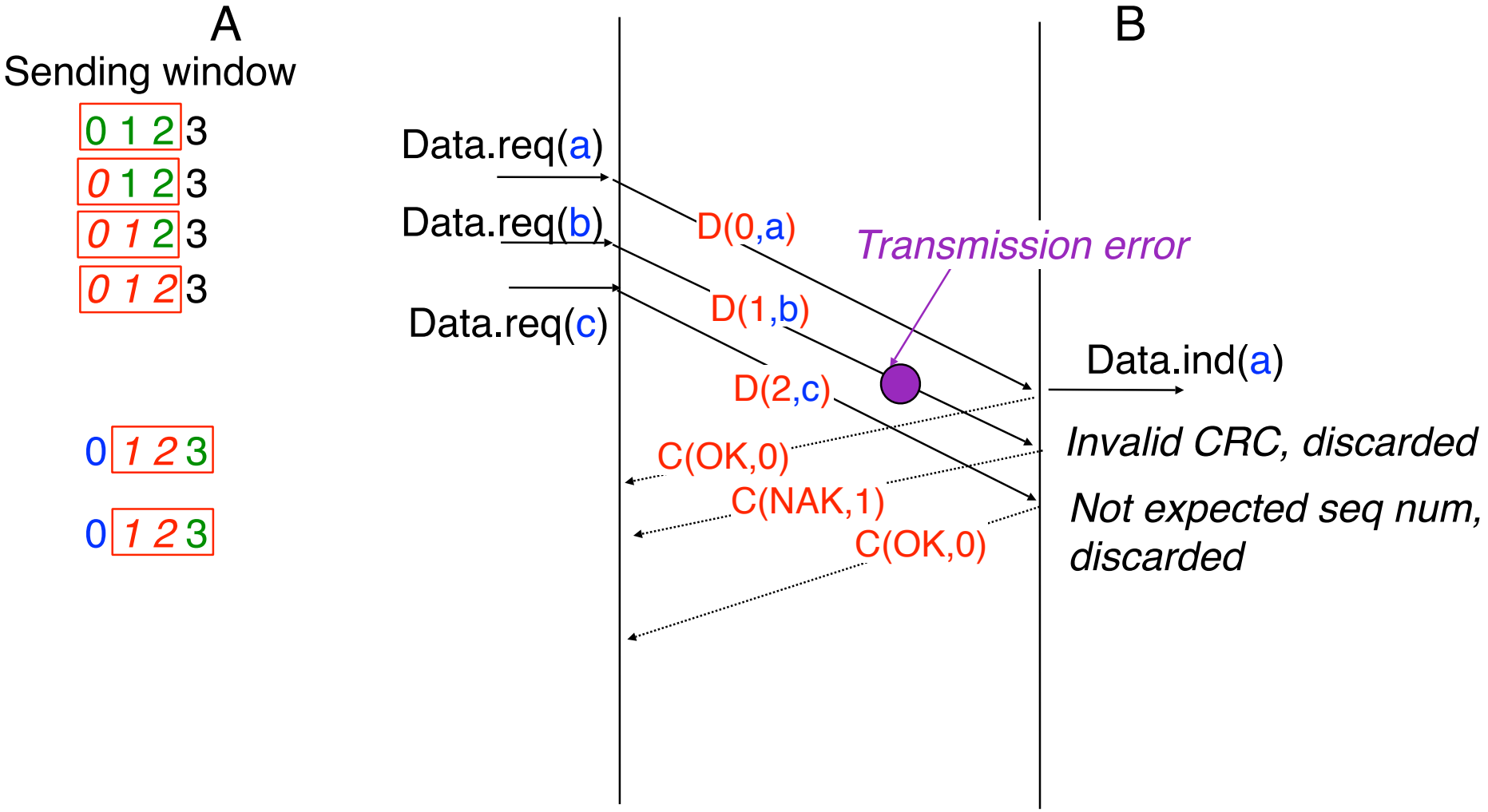
Go-Back-N : Example



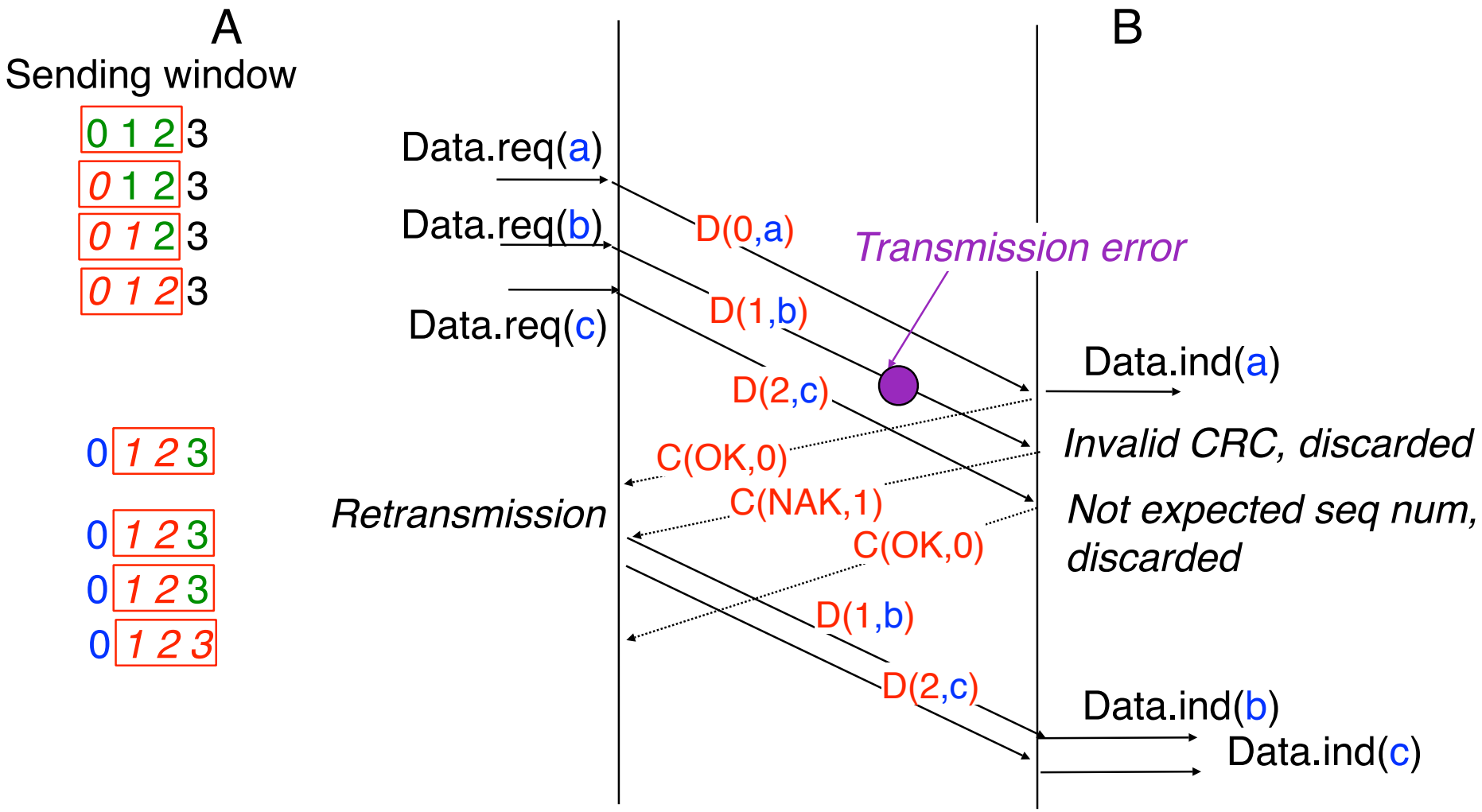
Go-Back-N : Example



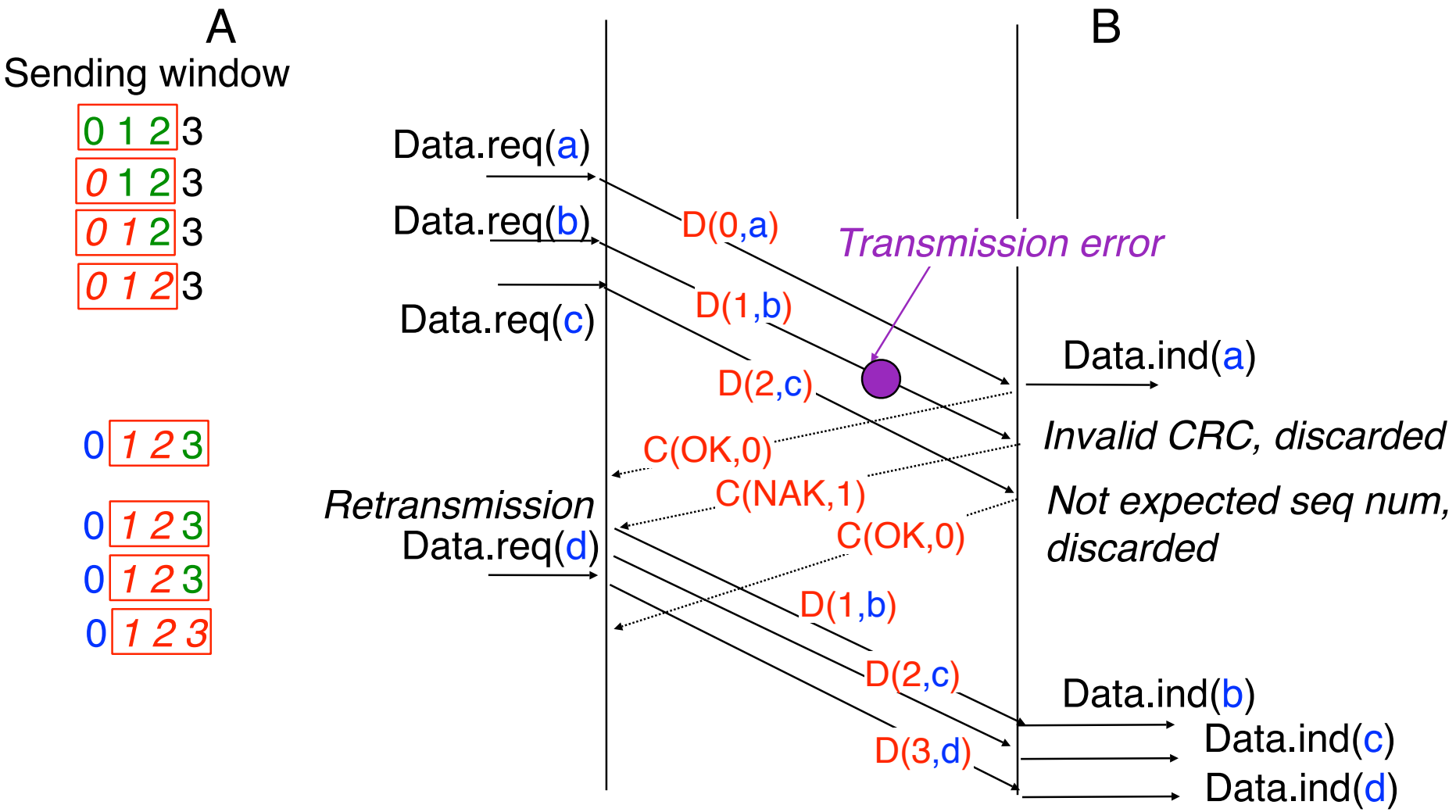
Go-Back-N : Example



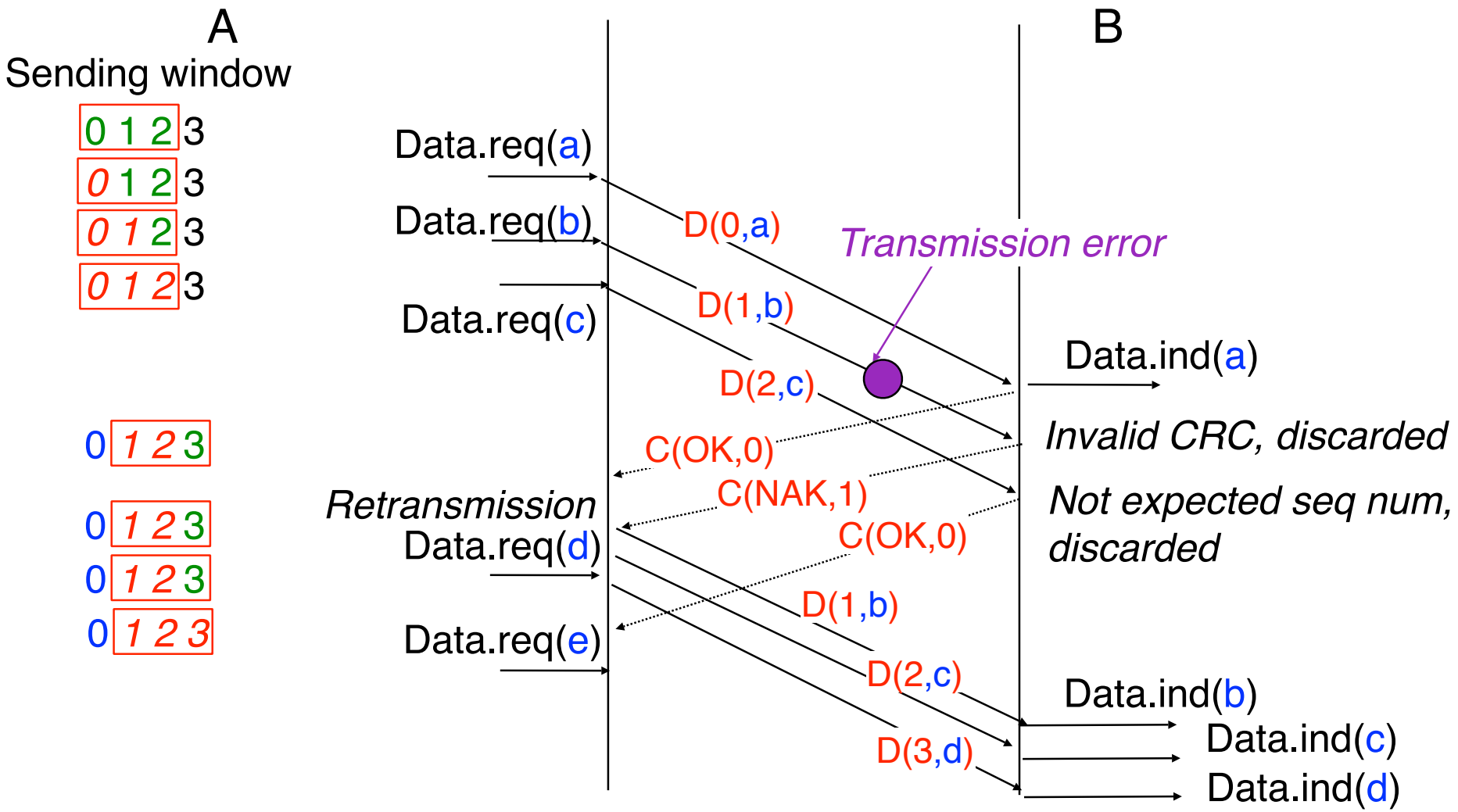
Go-Back-N : Example



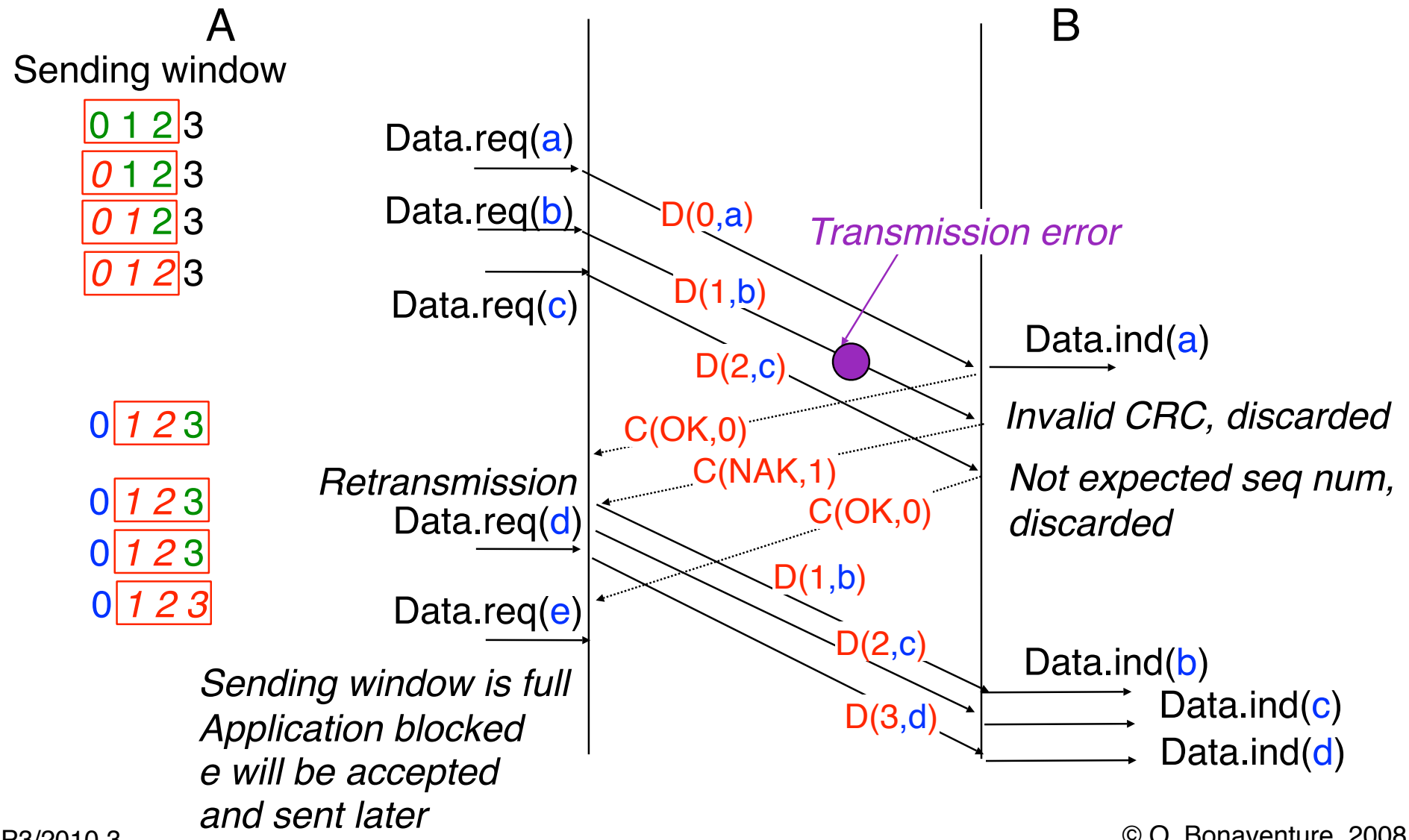
Go-Back-N : Example



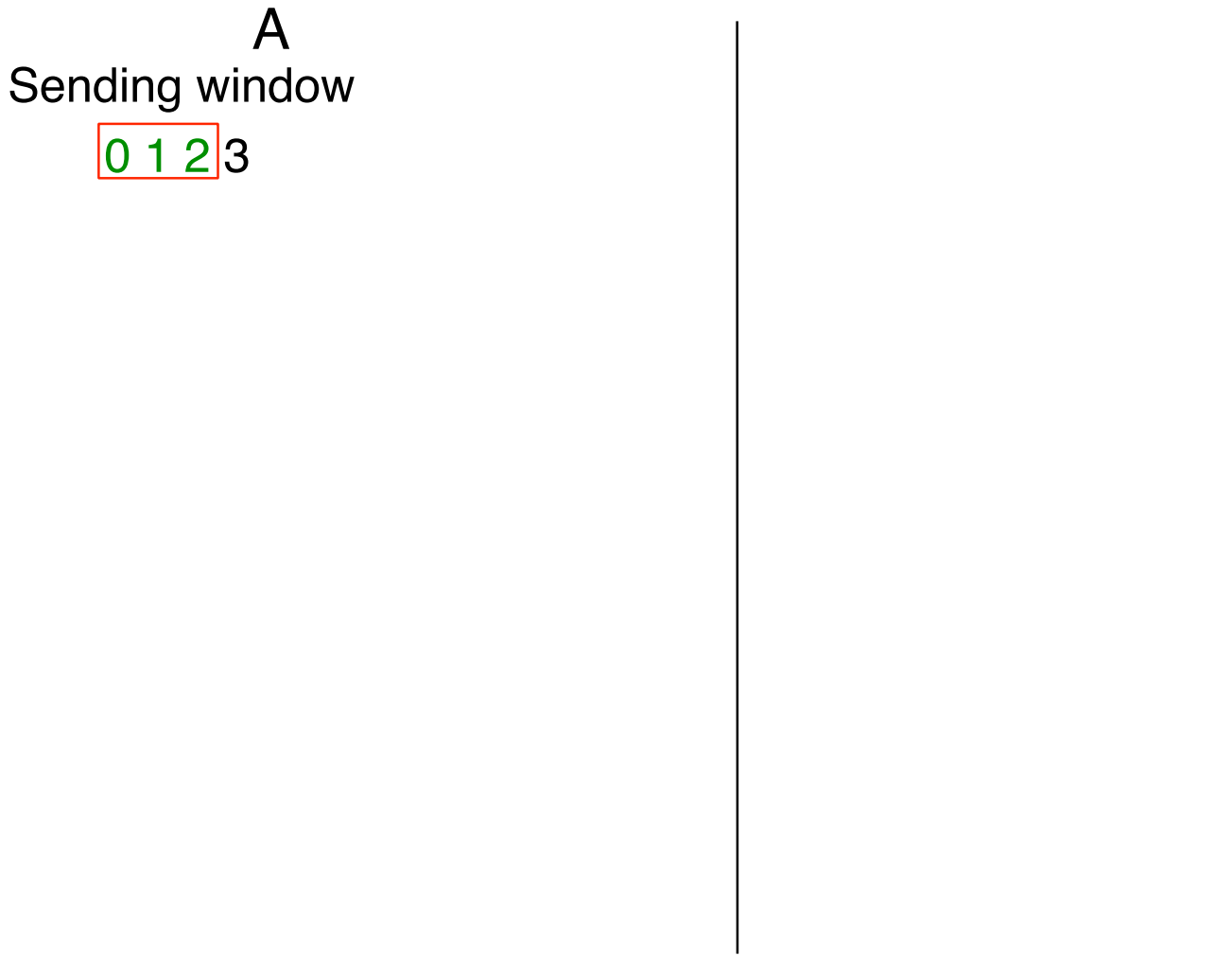
Go-Back-N : Example



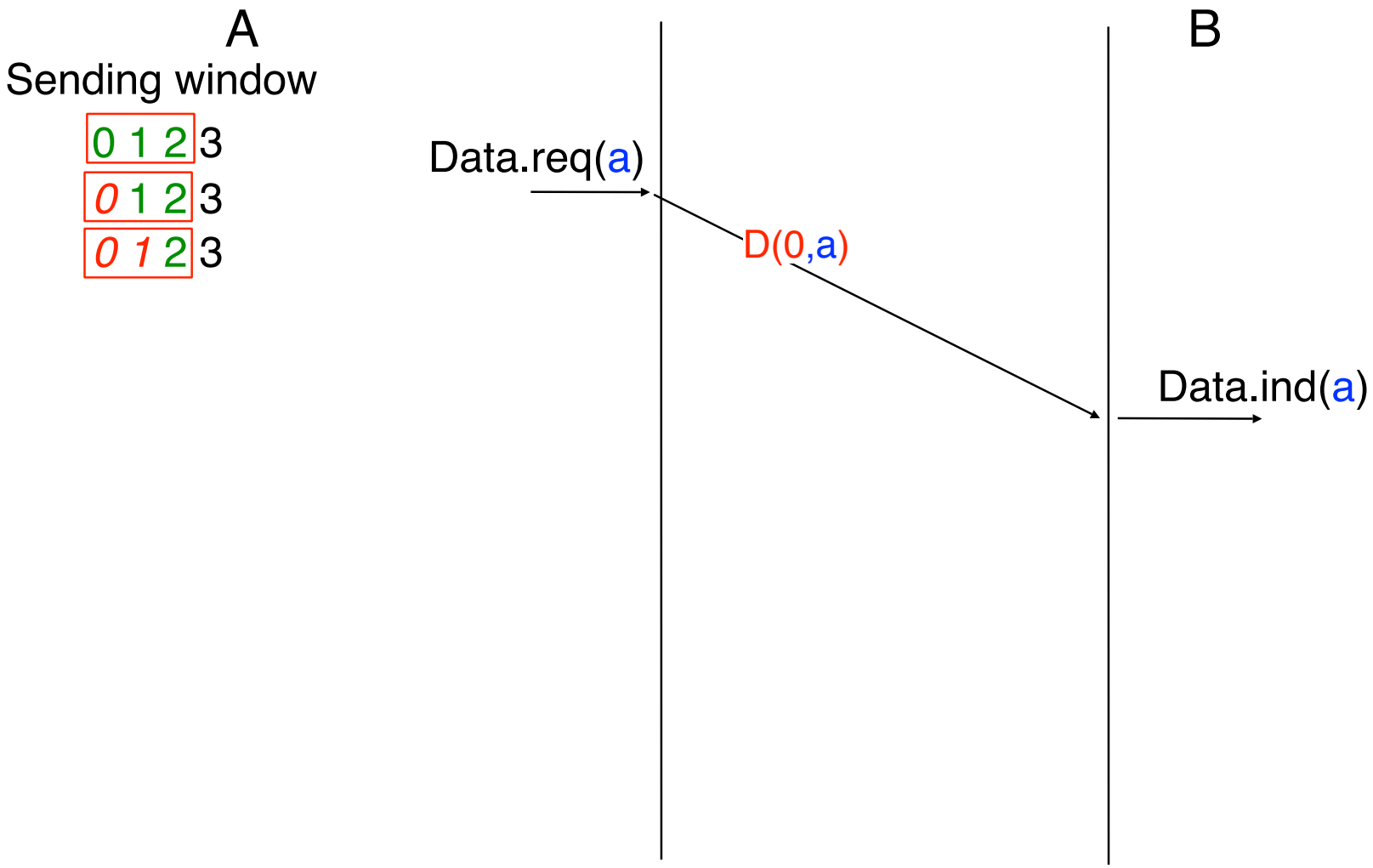
Go-Back-N : Example



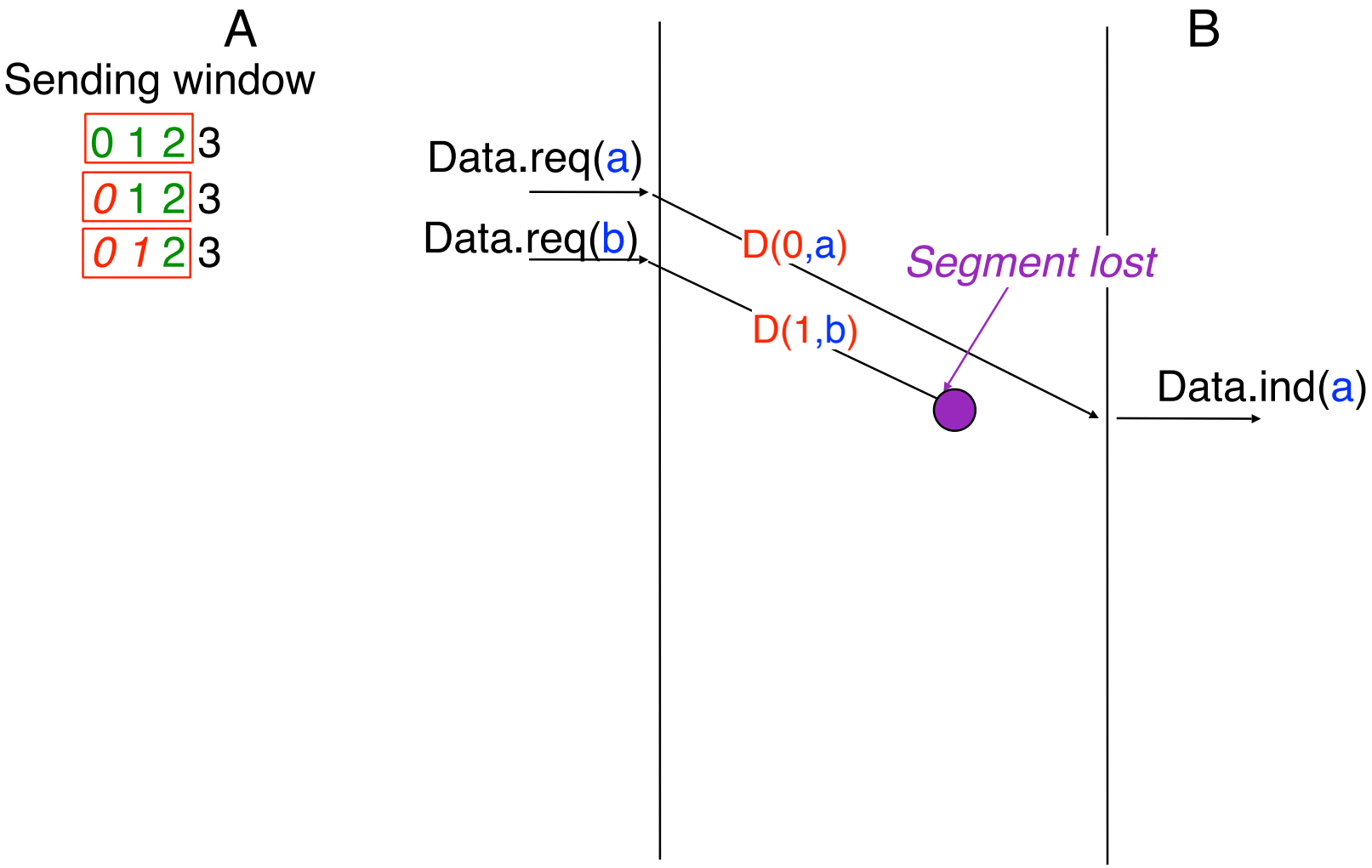
Go-Back-N : Example (2)



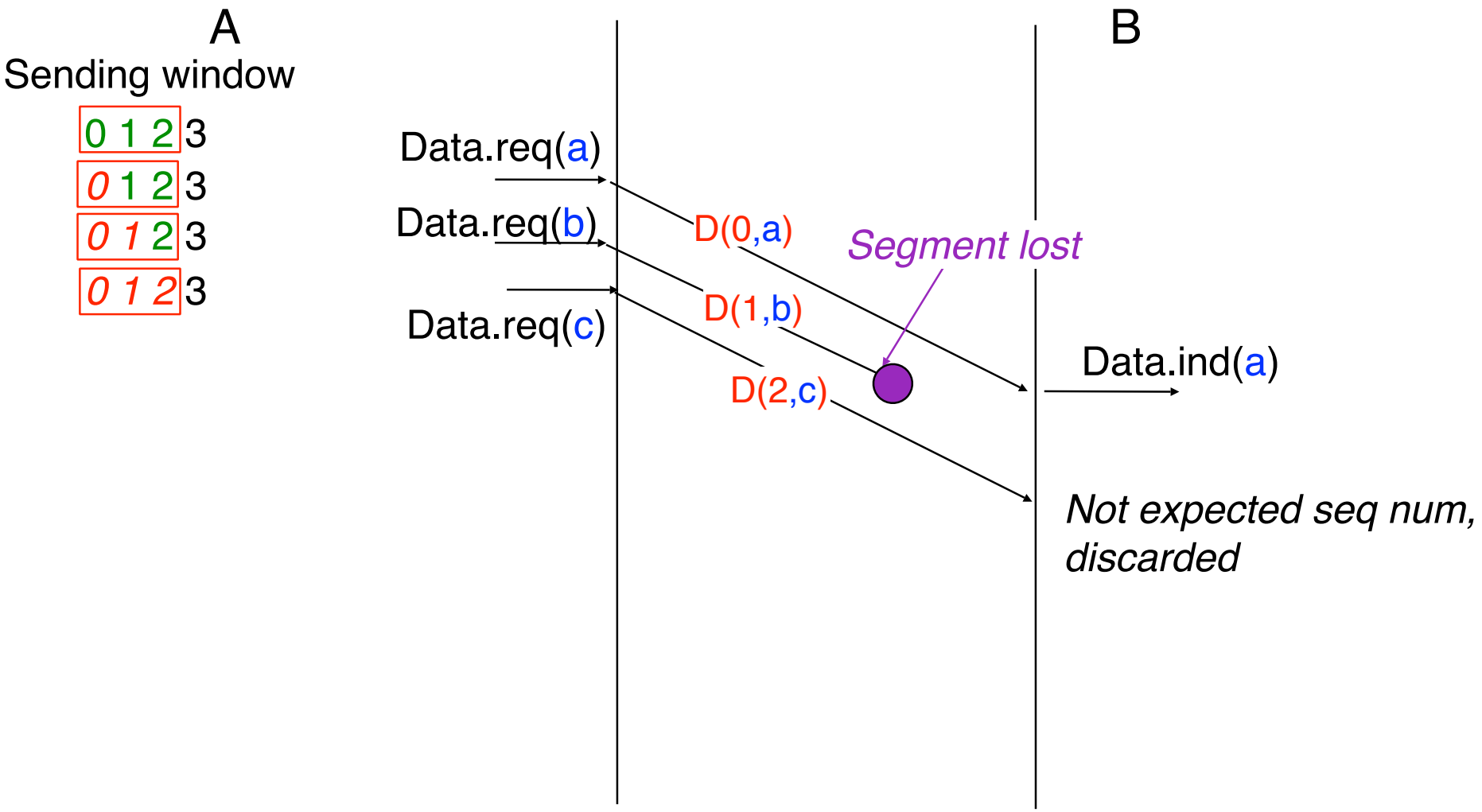
Go-Back-N : Example (2)



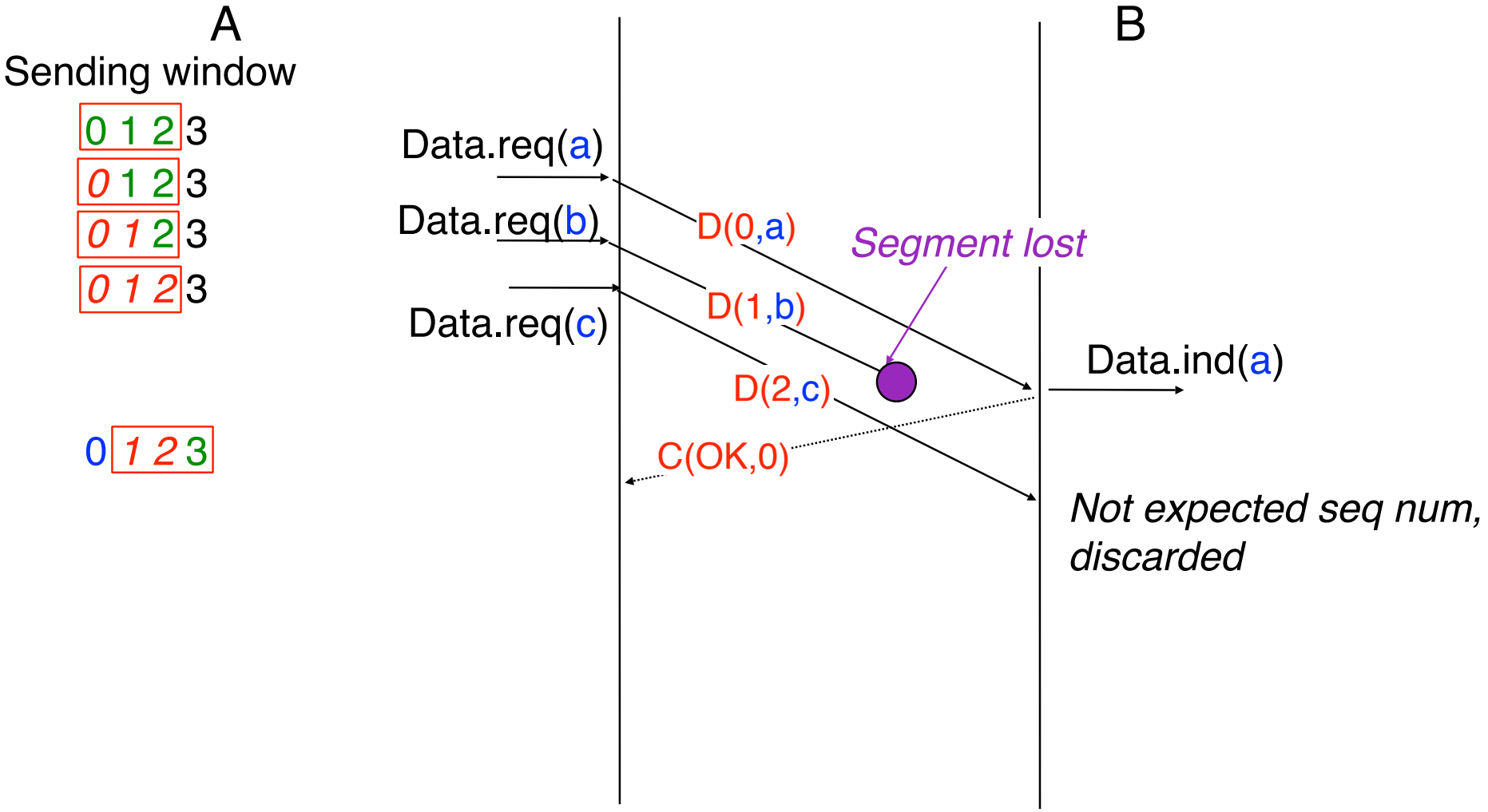
Go-Back-N : Example (2)



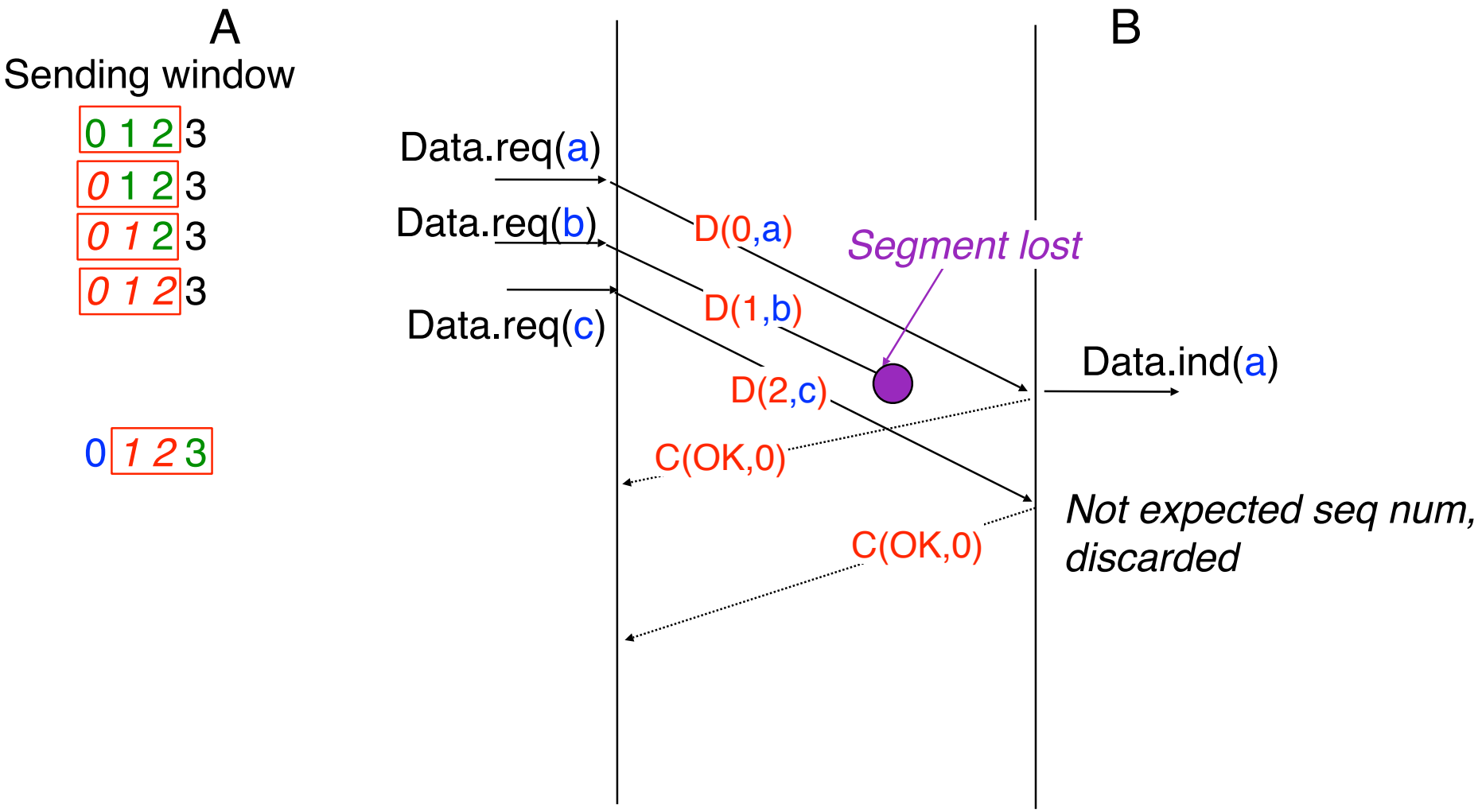
Go-Back-N : Example (2)



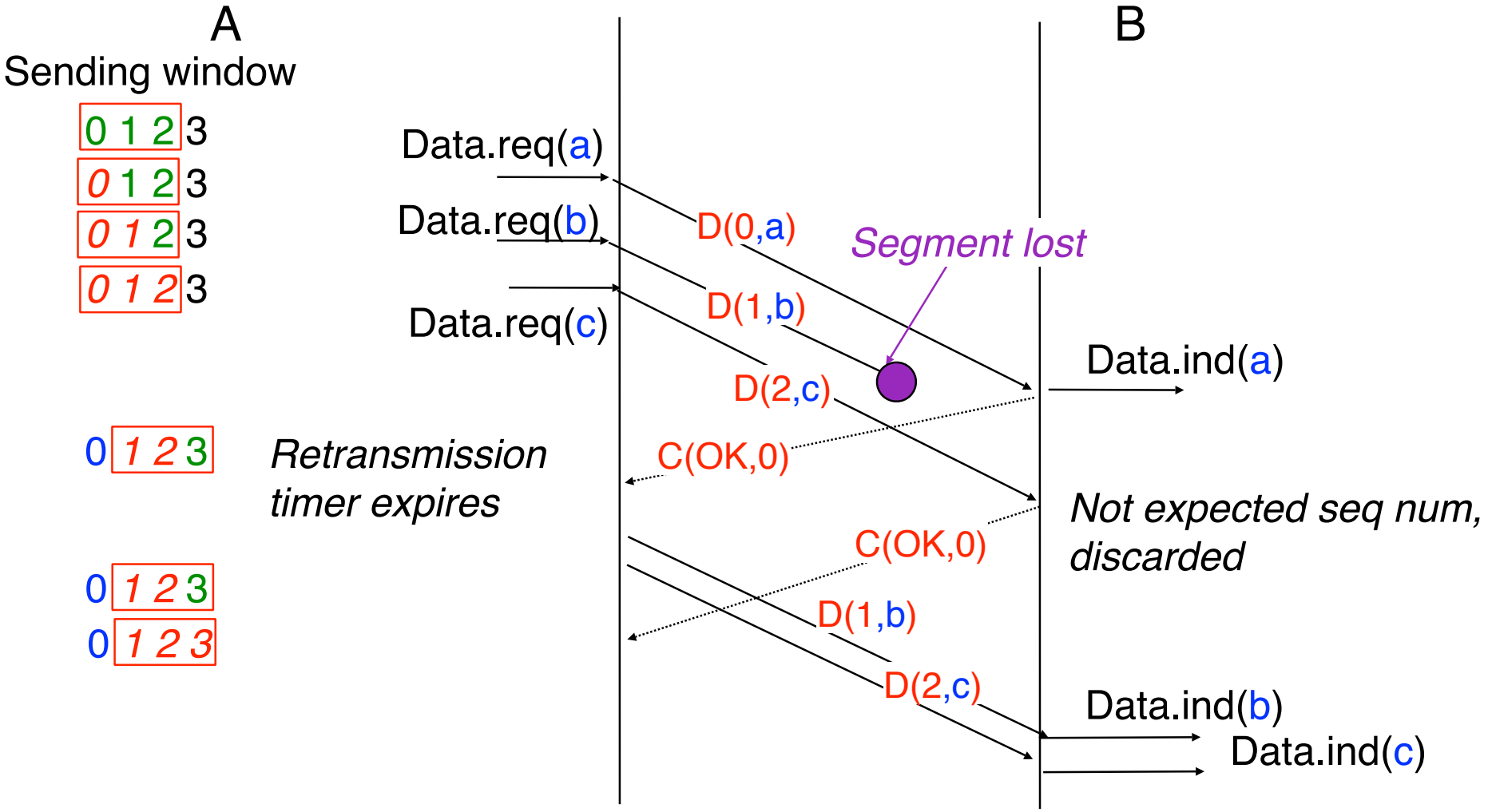
Go-Back-N : Example (2)



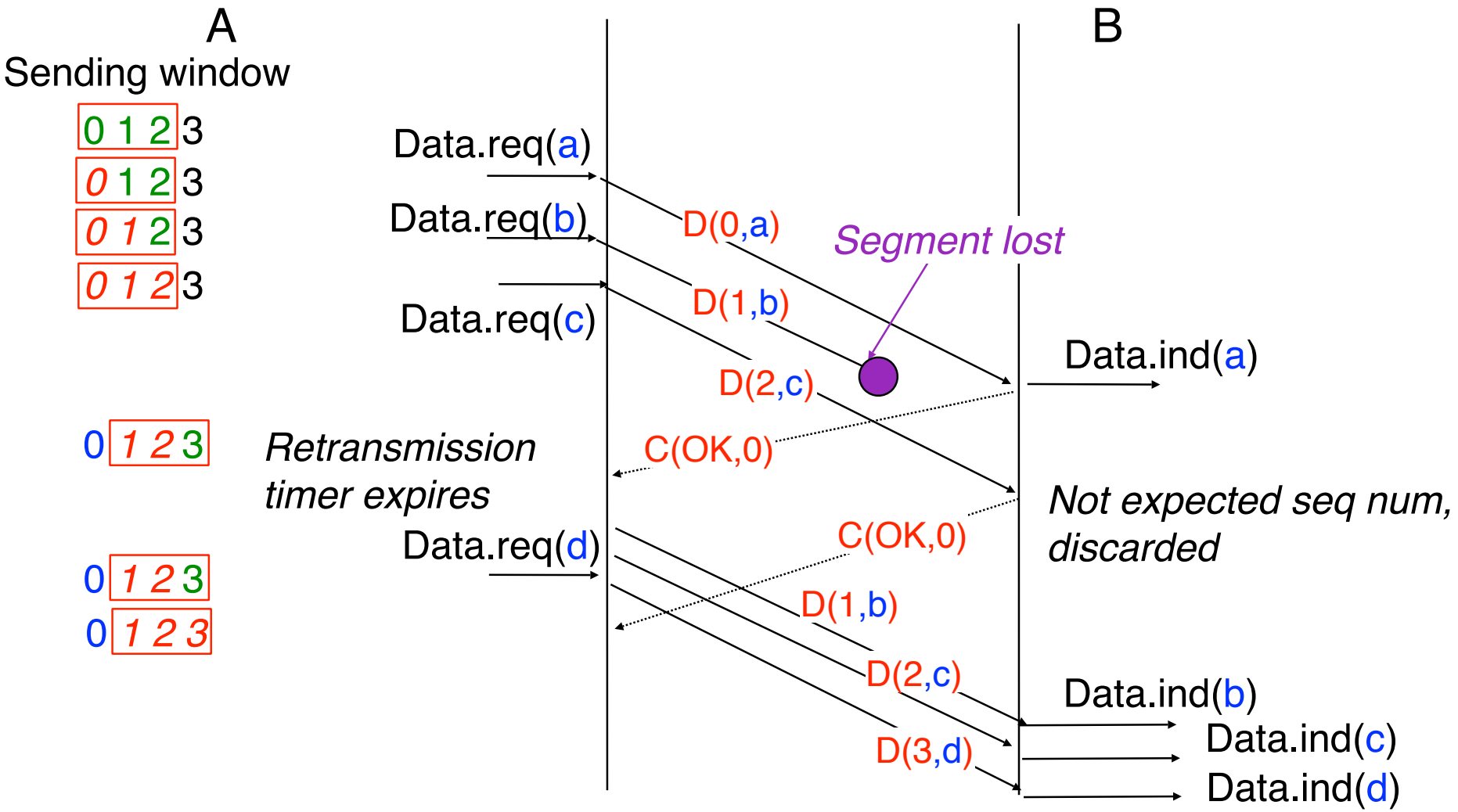
Go-Back-N : Example (2)



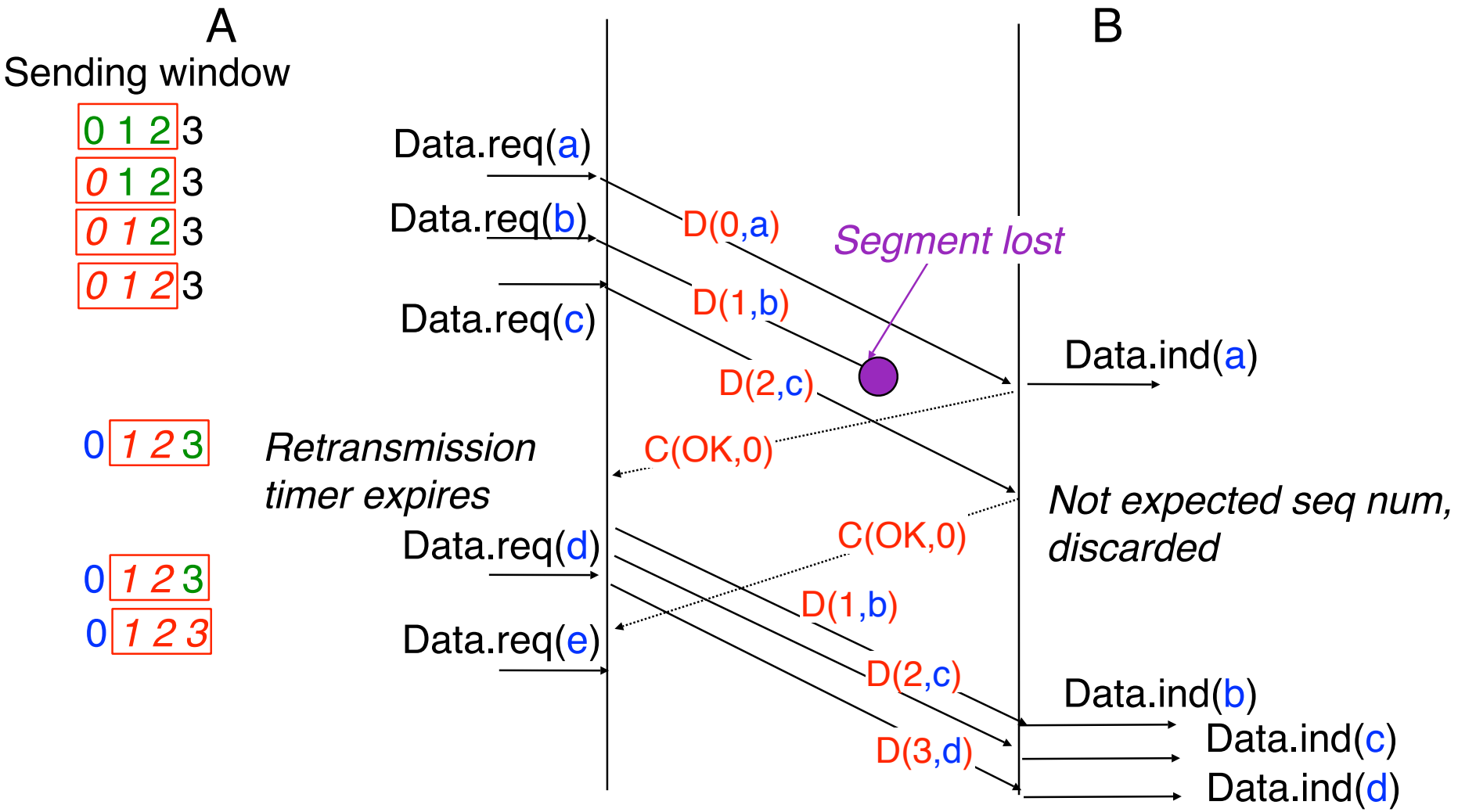
Go-Back-N : Example (2)



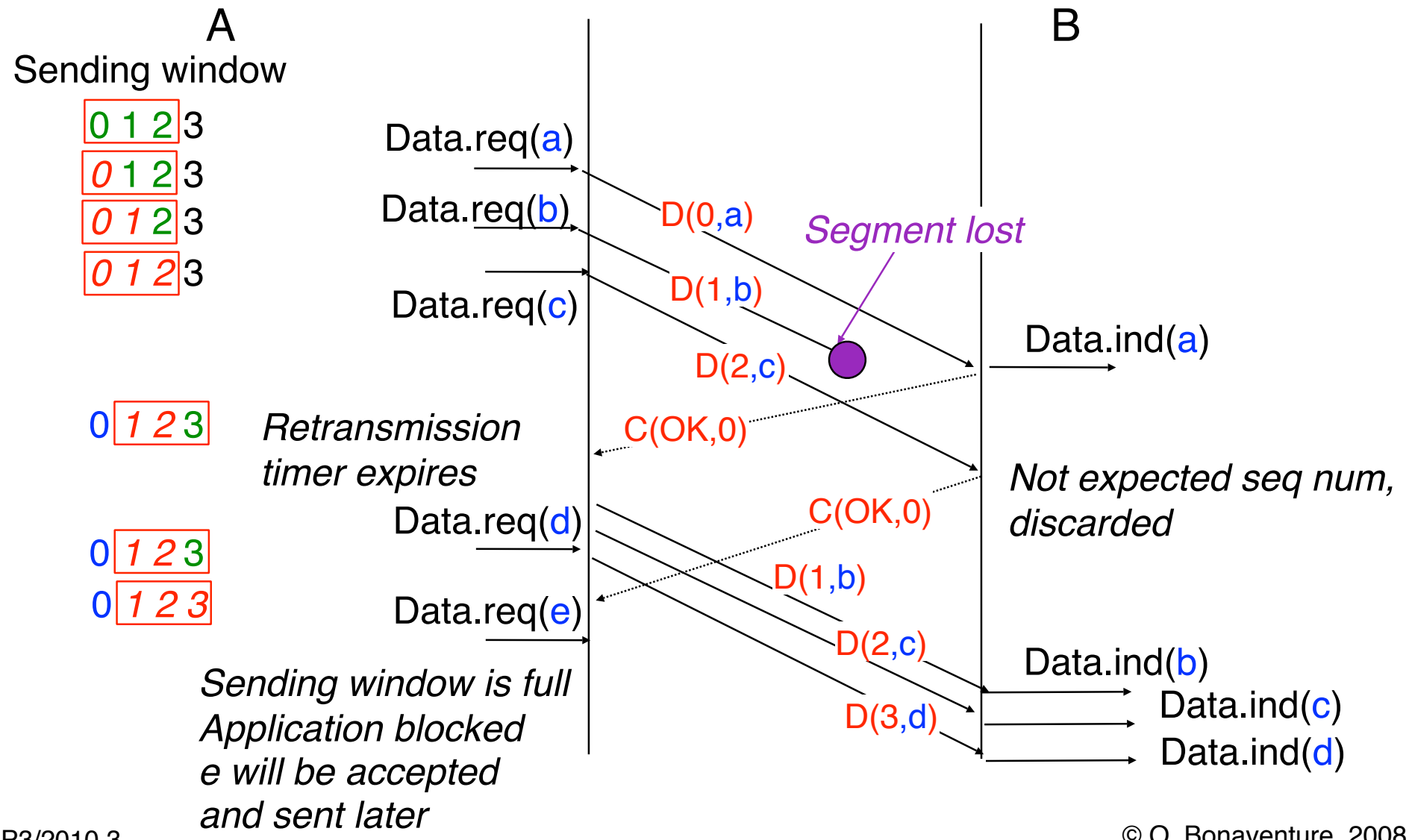
Go-Back-N : Example (2)



Go-Back-N : Example (2)



Go-Back-N : Example (2)

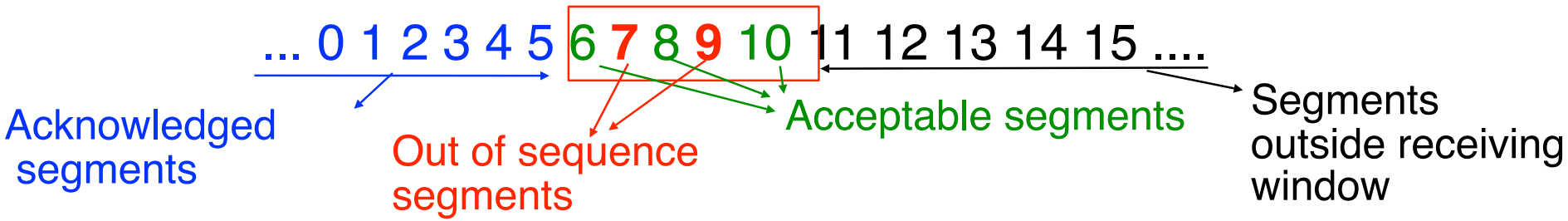


Selective Repeat

Receiver

Uses a buffer to store the segments received out of sequence and reorder their content

Receiving window

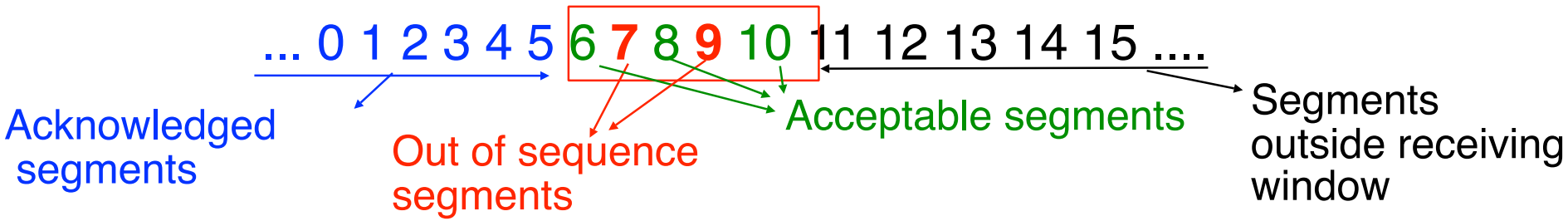


Selective Repeat

Receiver

Uses a buffer to store the segments received out of sequence and reorder their content

Receiving window



Semantics of the control segments

OKX

The segments up to and including sequence number X have been received

NAKX

The segment with sequence number X was errored

Sender

Upon detection of an errored or lost segment, sender retransmits only this segment

may require one retransmission timer per segment

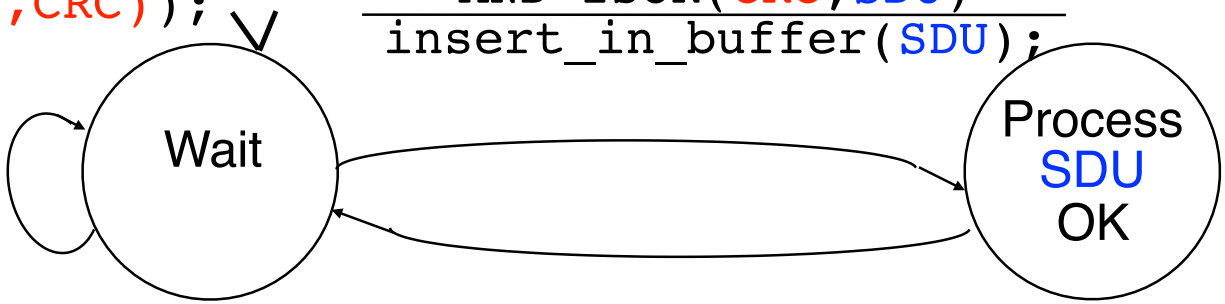
Selective-Repeat : Receiver

State variable

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

```
Recvd(D(t,SDU,CRC))
AND NOT(IsOK(CRC,SDU))
-----------------------
discard(SDU);
send(C(NAK,t,CRC));
```

```
Recvd(D(t,SDU,CRC))
AND IsOK(CRC,SDU)
-----------------------
insert_in_buffer(SDU);
```




```
-----------------------
For all in sequence segments inside buffer
        Data.ind(SDU);
slide the sliding window;
update next and last
send(C(OK,(next-1)));
```

Selective Repeat : Sender

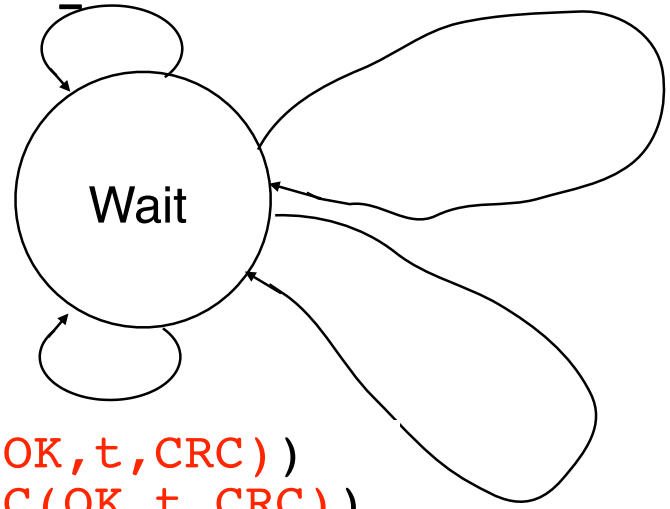
State variables

`base` : sequence number of oldest unacknowledged segment

`seq` : first free sequence number

`W` : size of sending window

```
Recvd(C(?,?,CRC))
and NOT( CRCOK(C(?,?,CRC)))
```

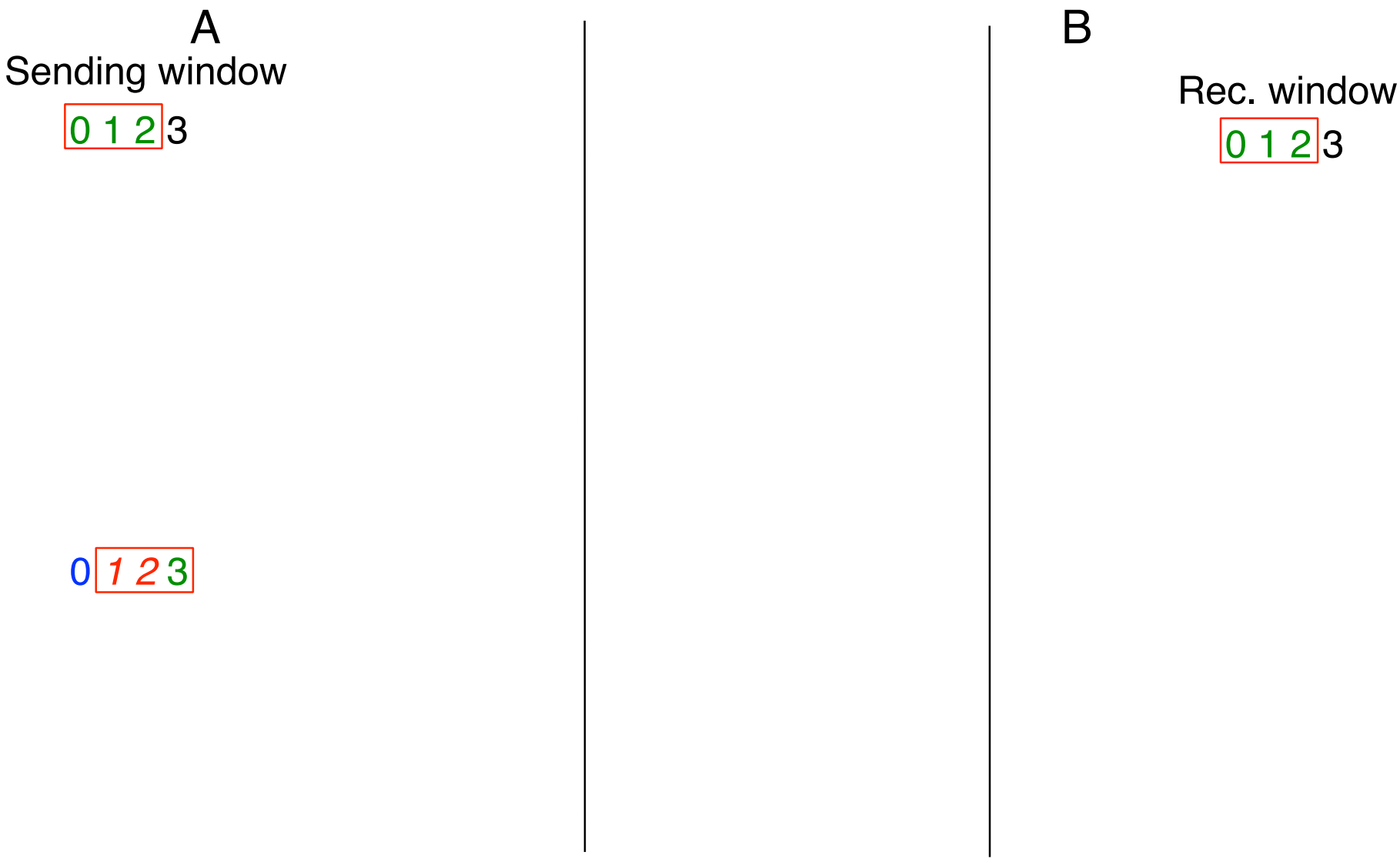



```
Data.req(SDU)
AND ( window not full )
start_timer(seq) ;
insert_in_buffer(SDU);
send(D(seq,SDU,CRC));
seq=(seq+1) ;
```

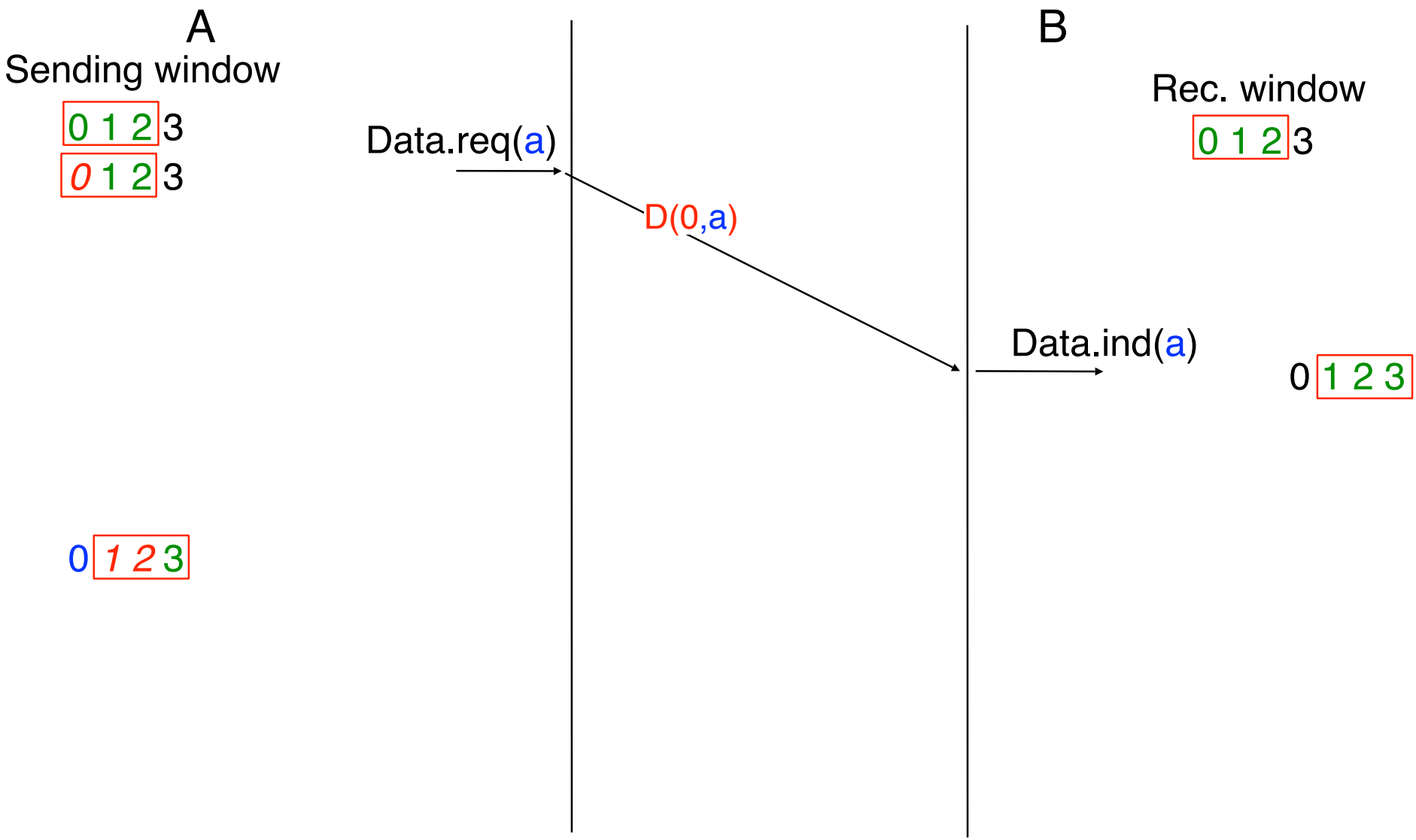
```
Recvd(C(OK,t,CRC))
and CRCOK(C(OK,t,CRC))
For all segments i<=t
    cancel_timer(t);
slide sliding window to
the right;
```

```
[ Recvd(C(NAK,t,CRC))
and CRCOK(C(NAK,t,CRC))]
  or timer (t) expires
send(D(t,SDU,CRC)); }
restart_timer(t);
```

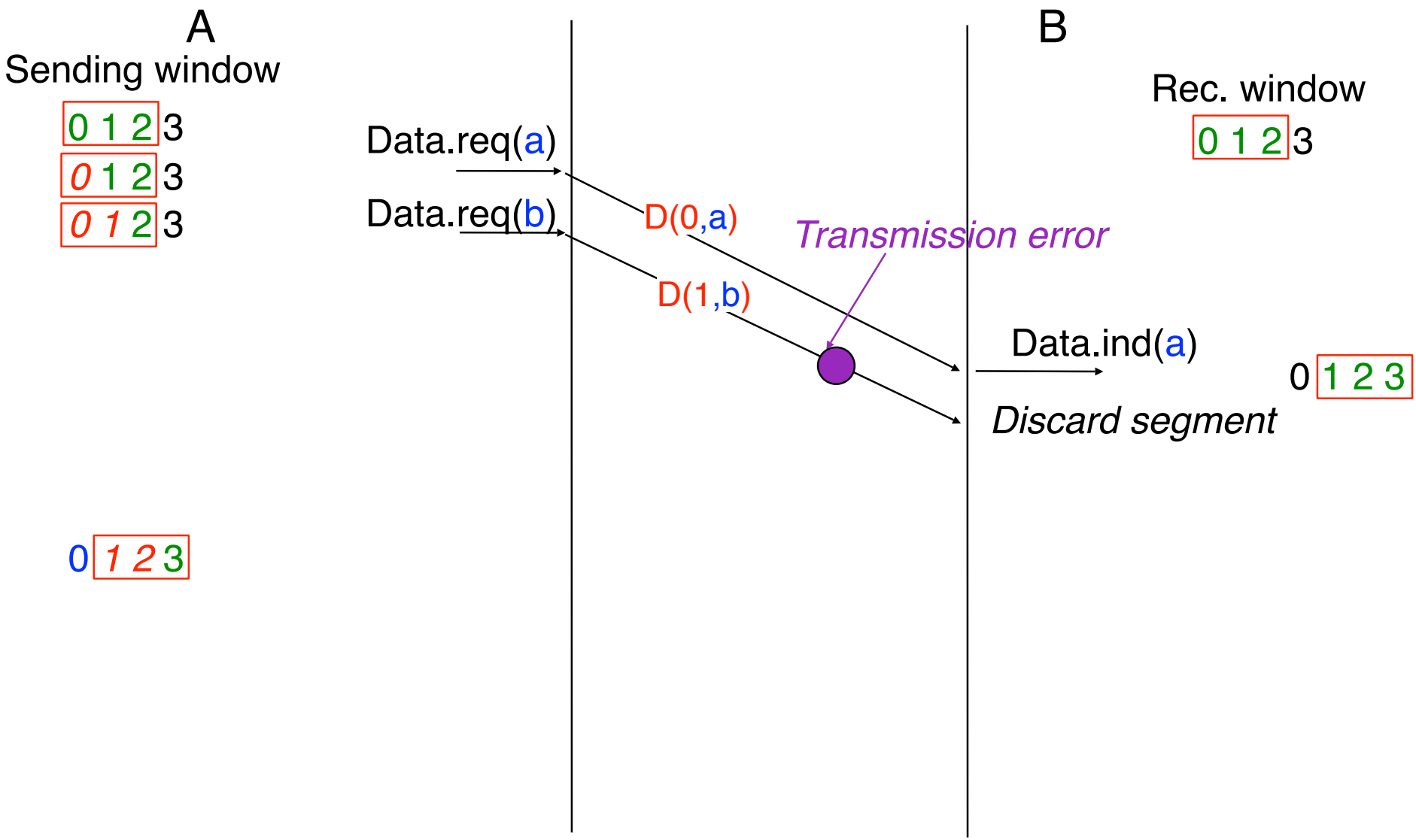
Selective Repeat : Example



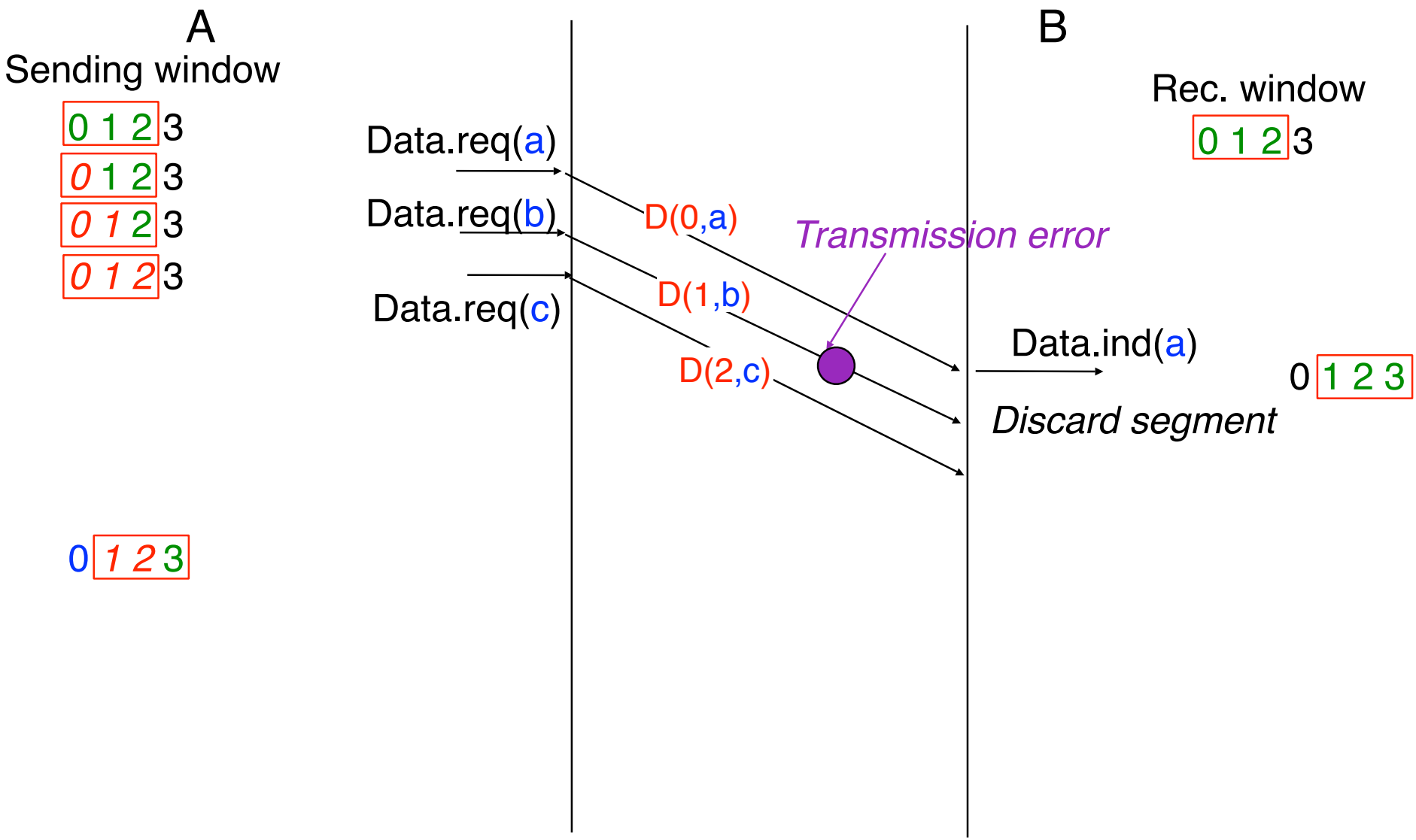
Selective Repeat : Example



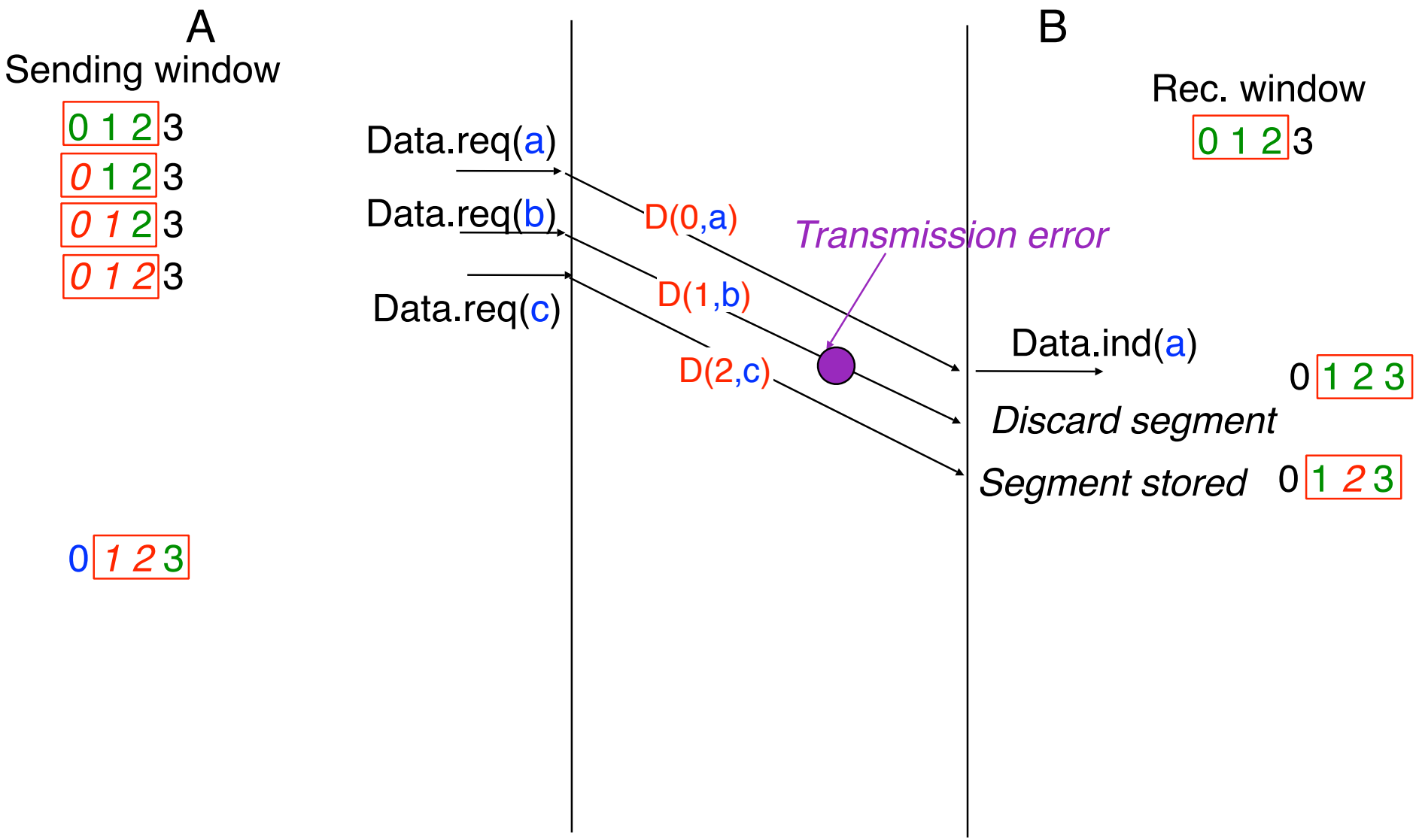
Selective Repeat : Example



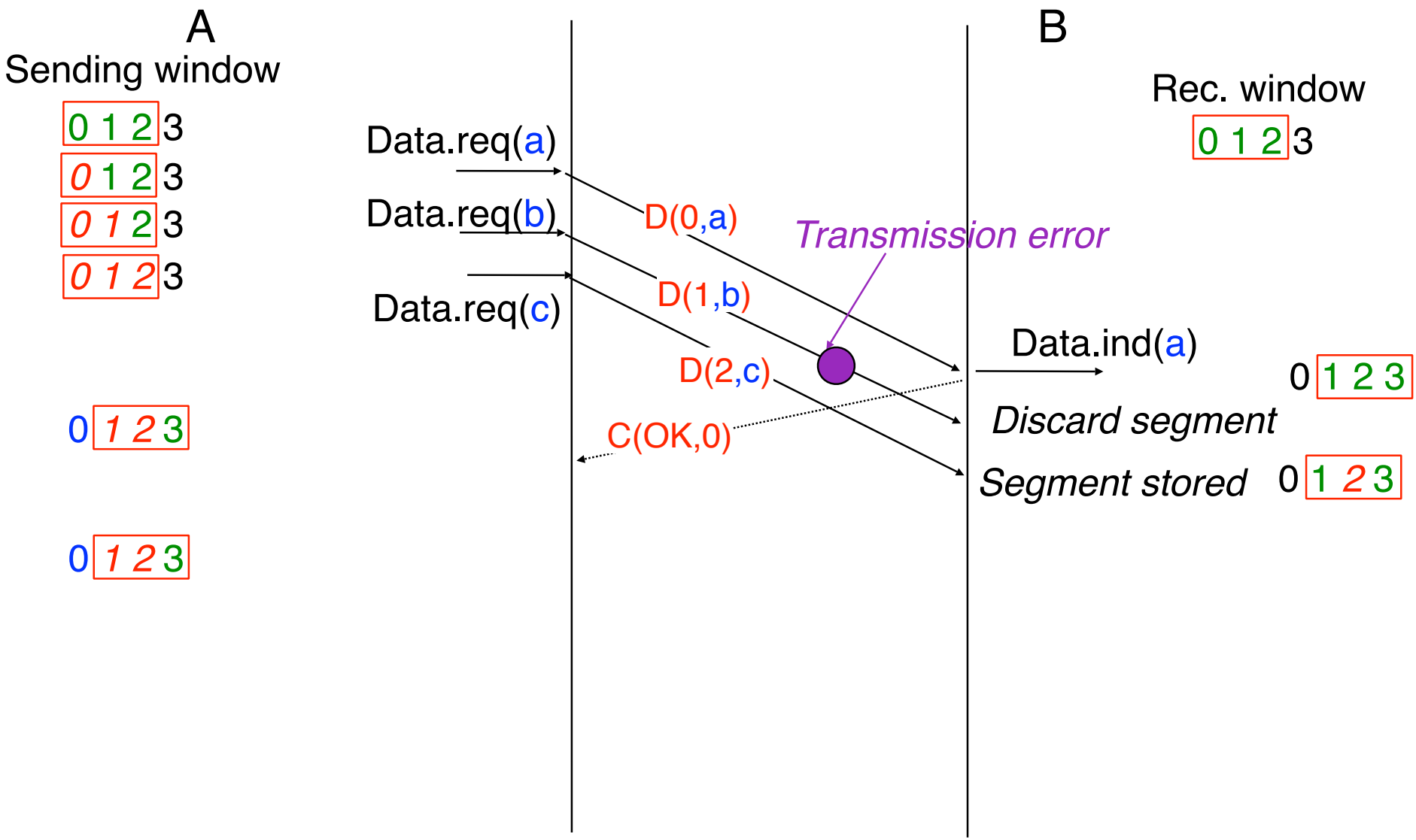
Selective Repeat : Example



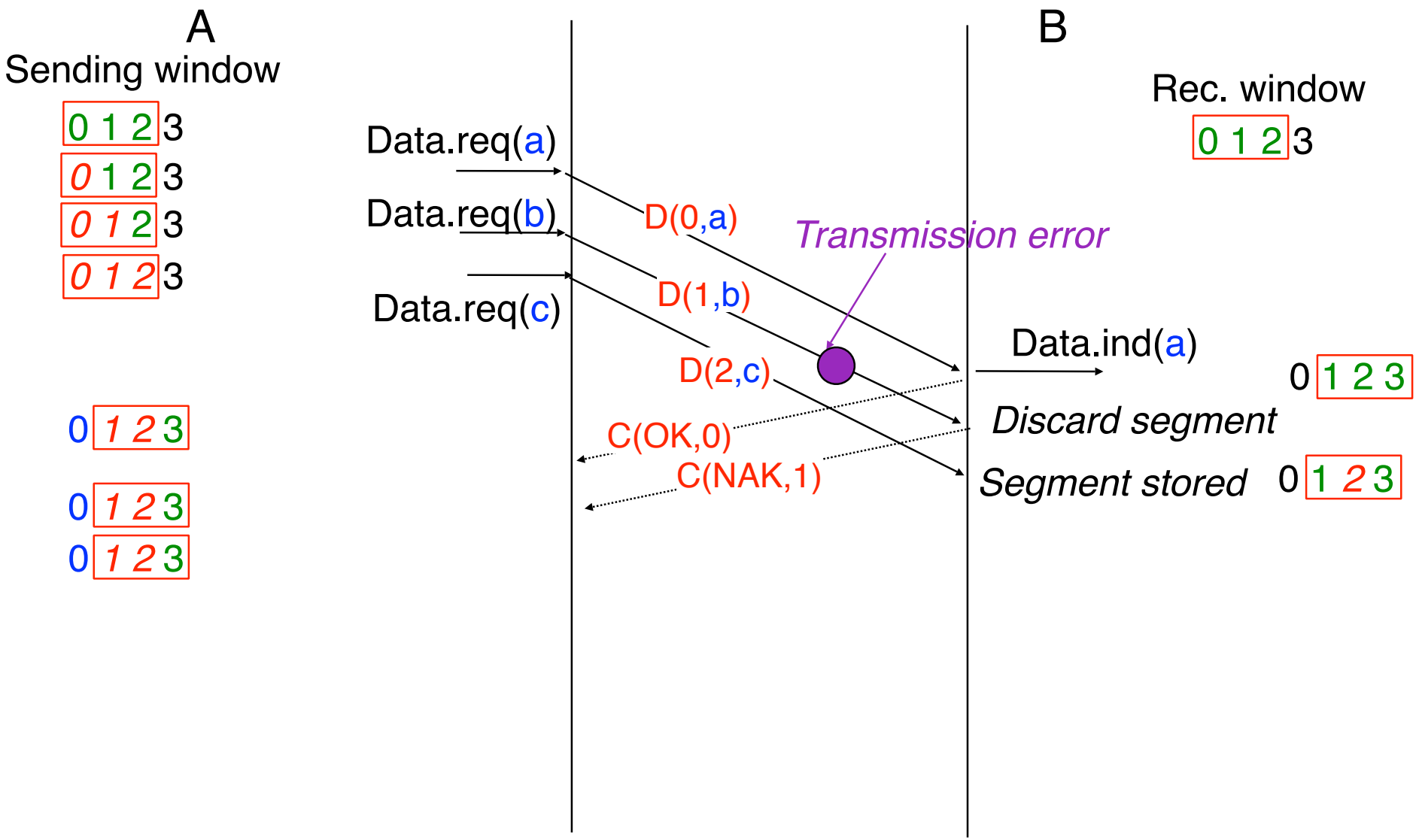
Selective Repeat : Example



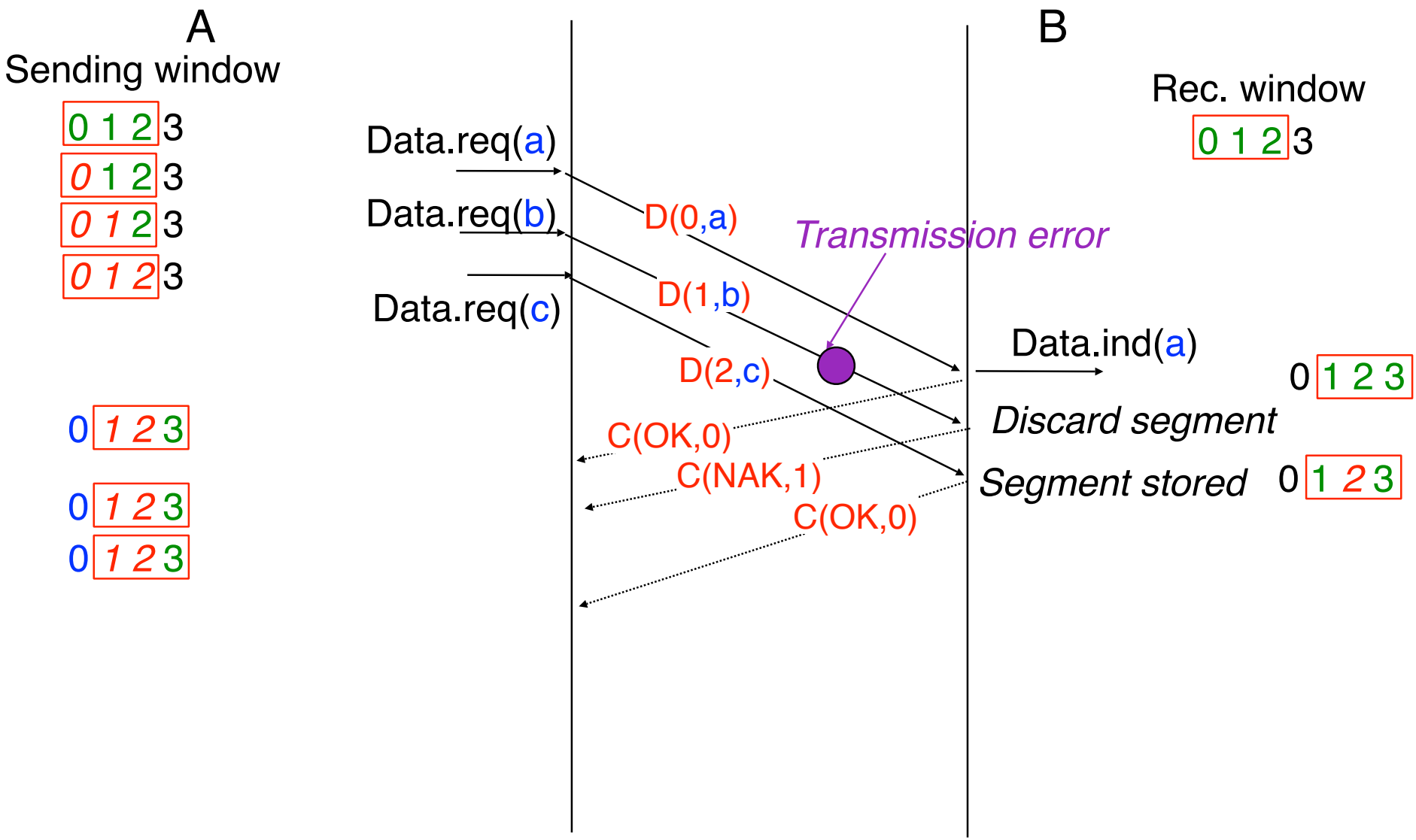
Selective Repeat : Example



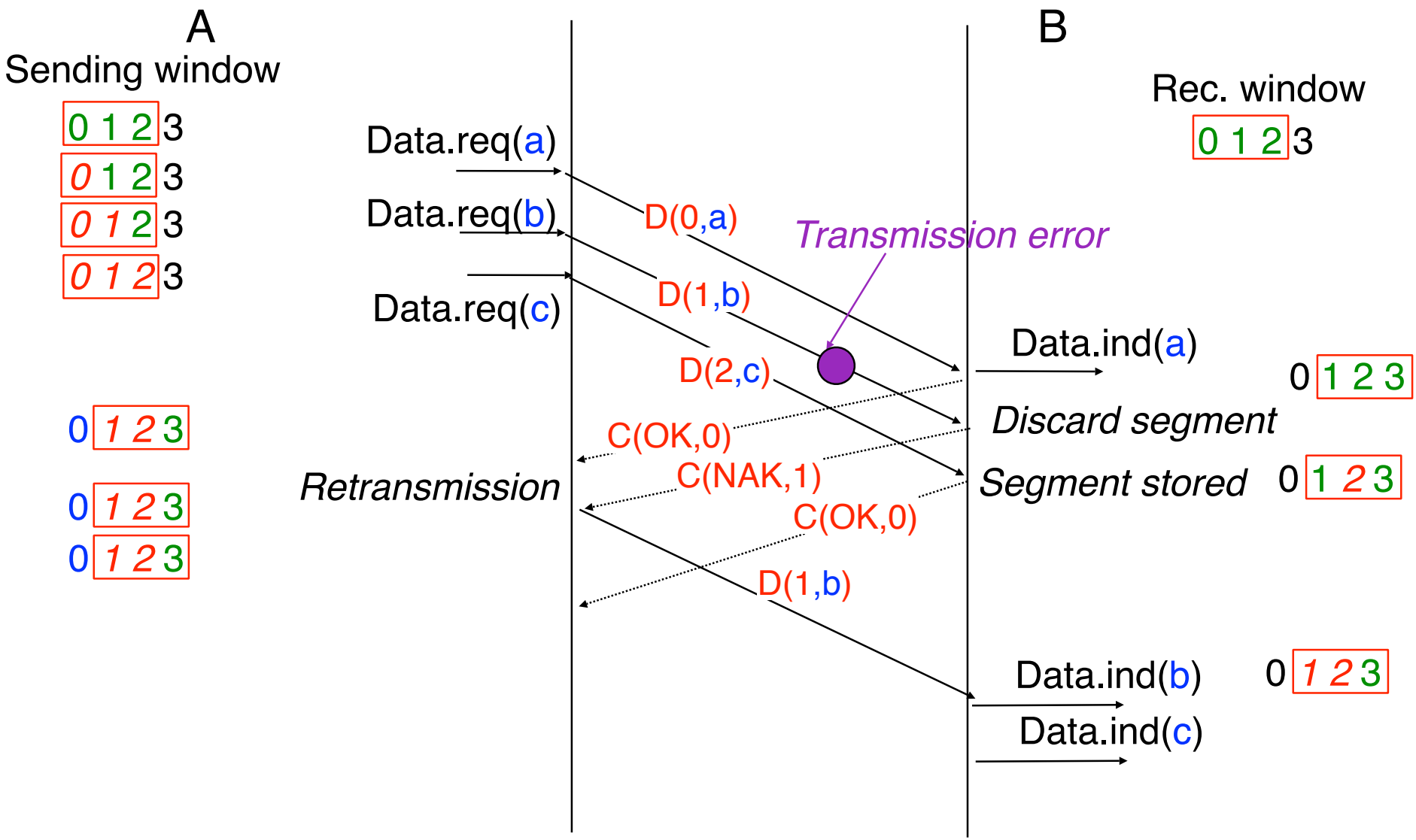
Selective Repeat : Example



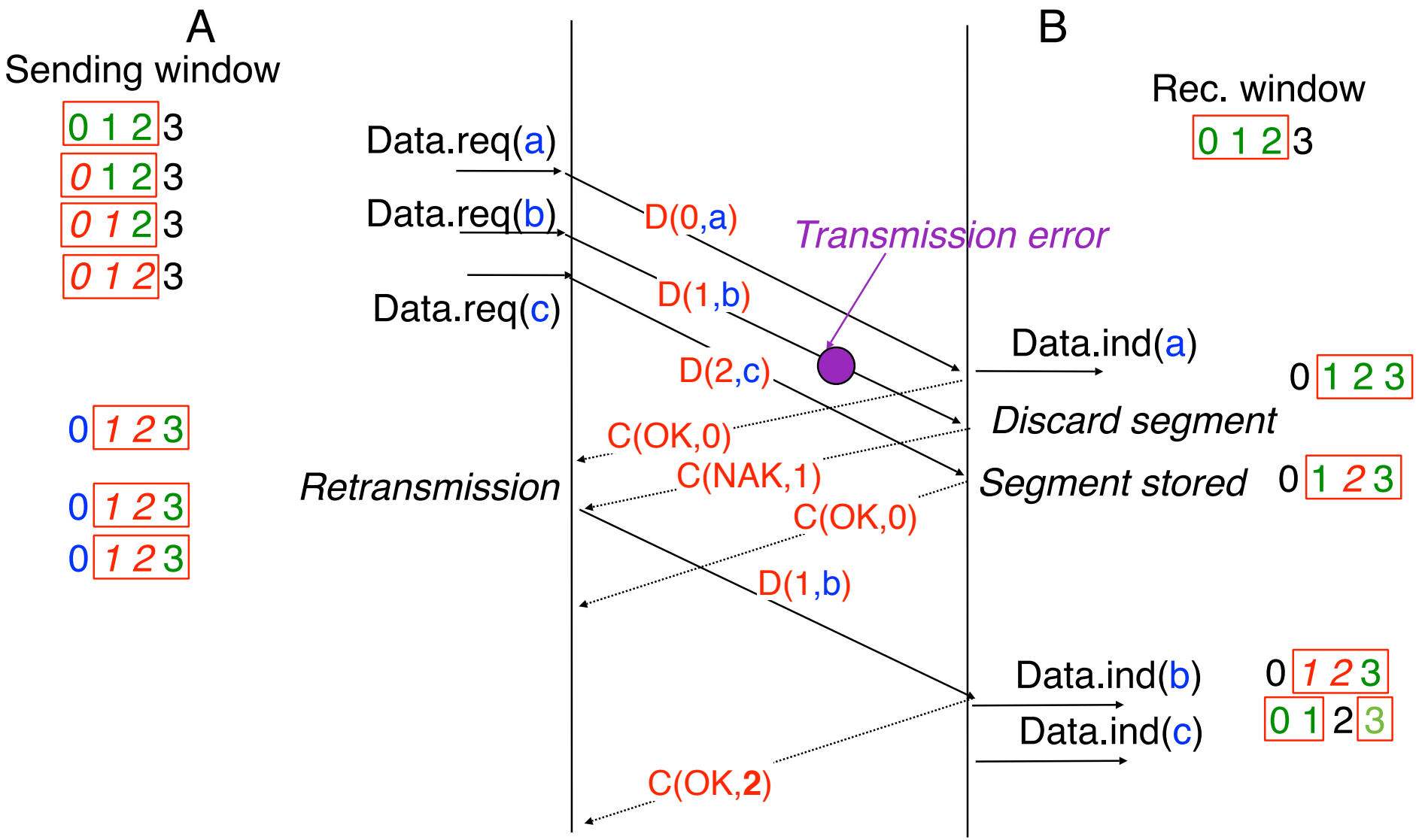
Selective Repeat : Example



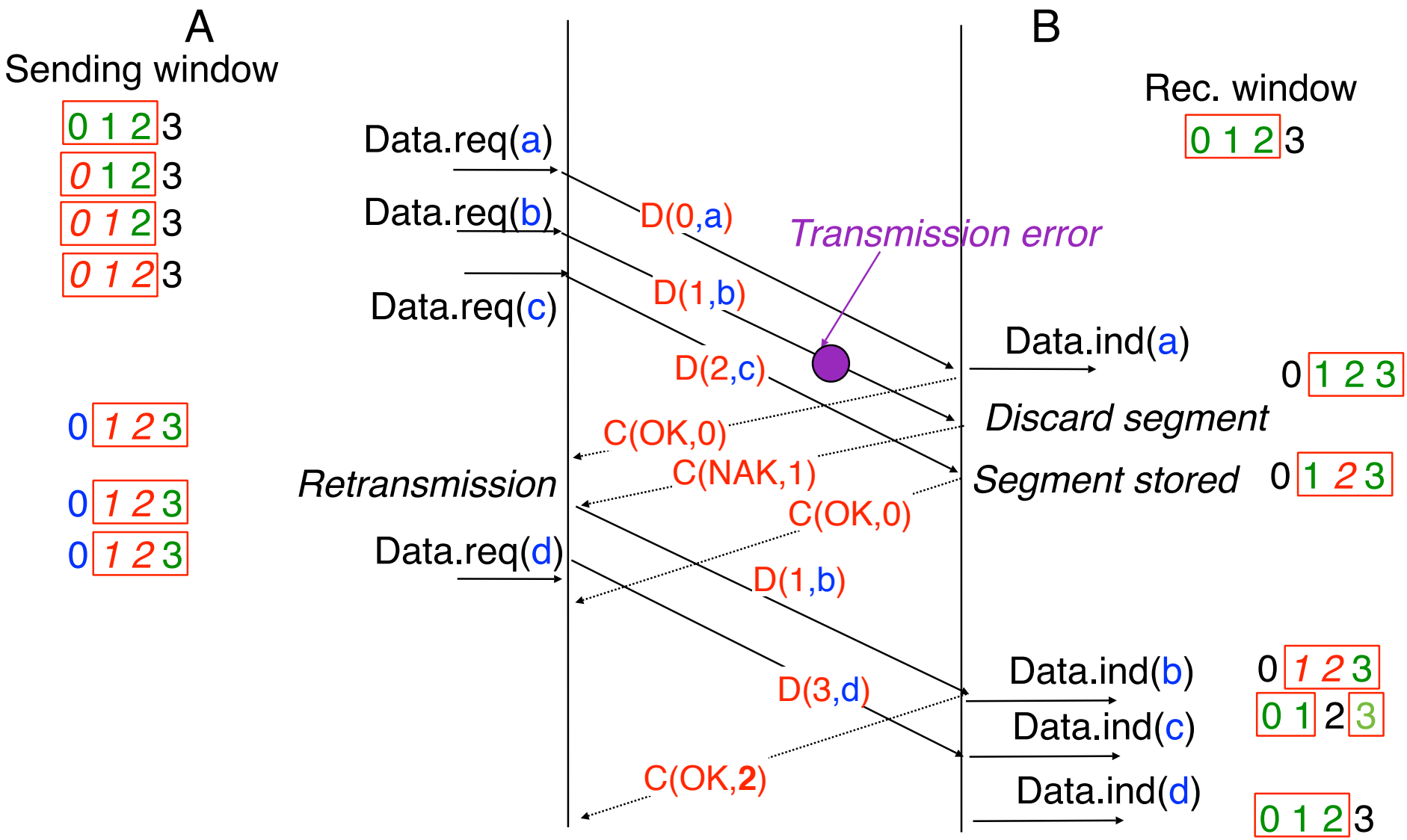
Selective Repeat : Example



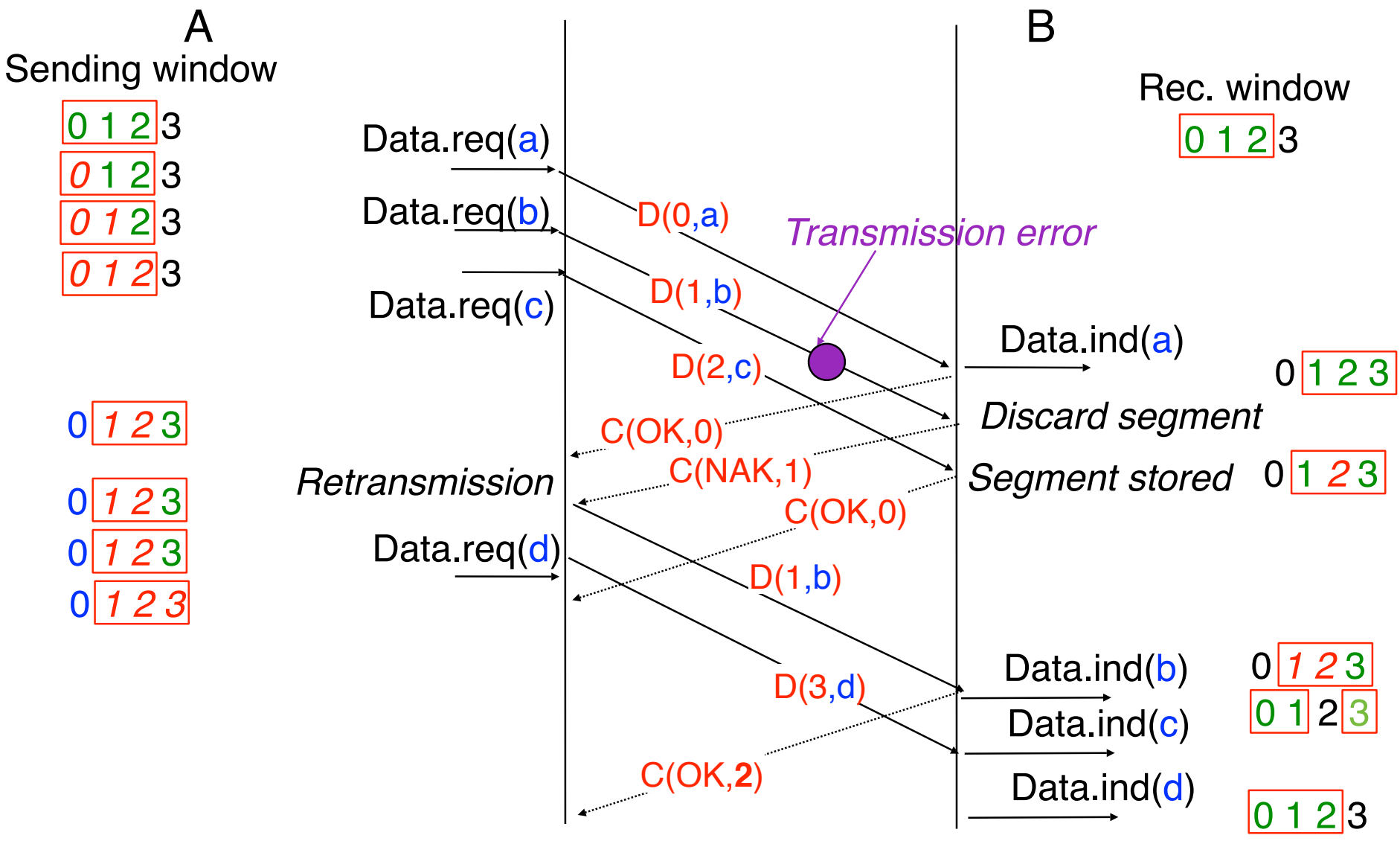
Selective Repeat : Example



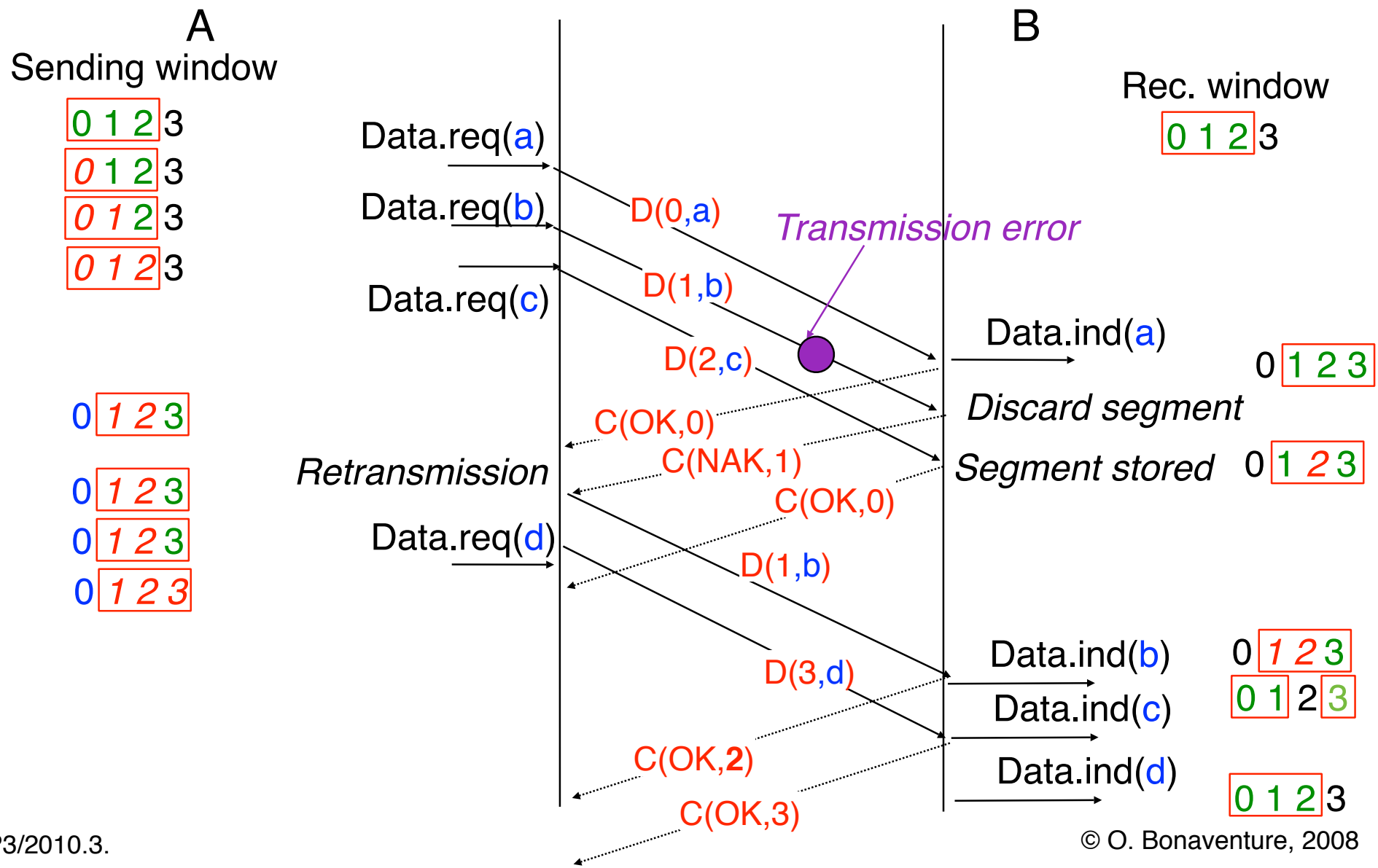
Selective Repeat : Example



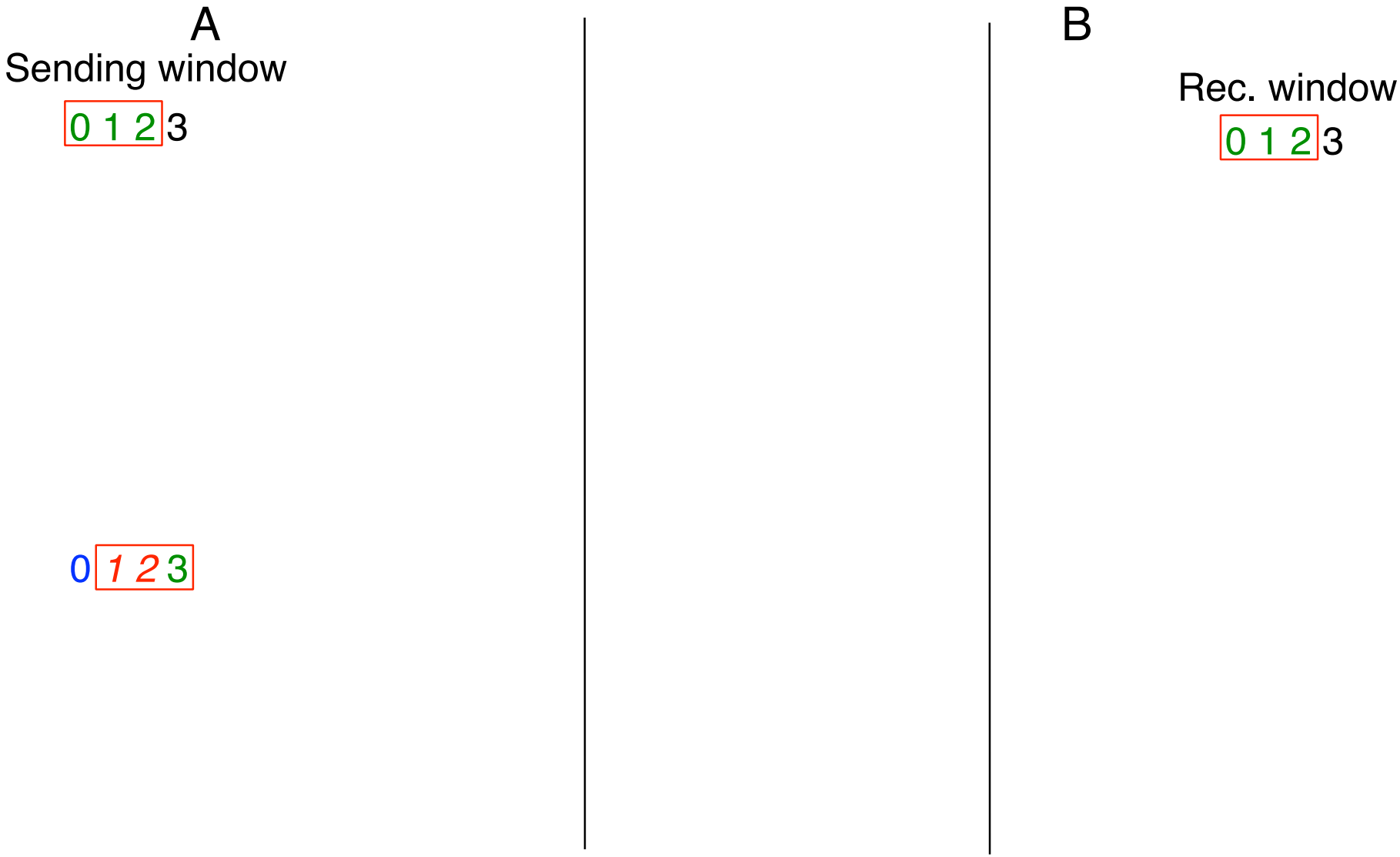
Selective Repeat : Example



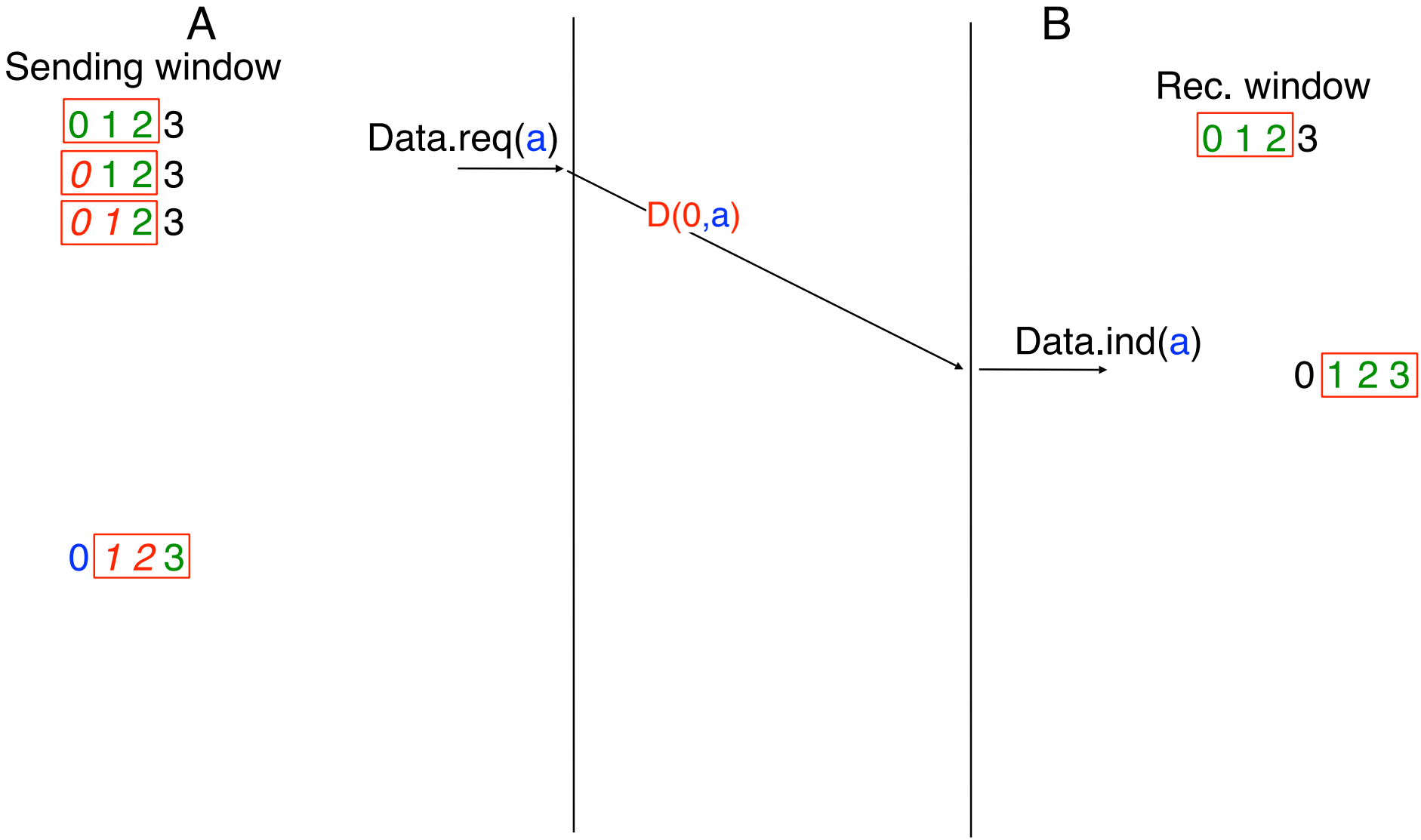
Selective Repeat : Example



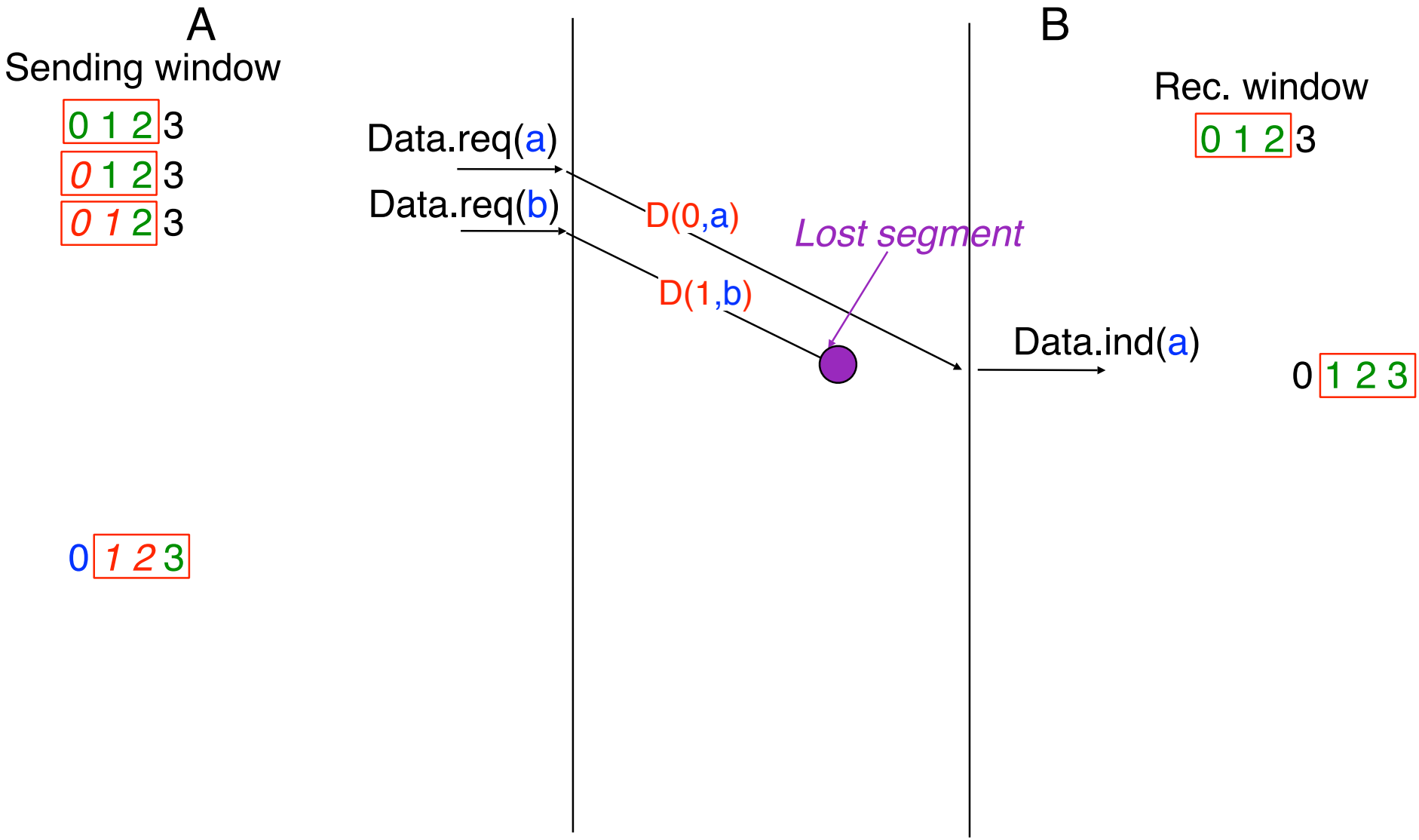
Selective Repeat : Example (2)



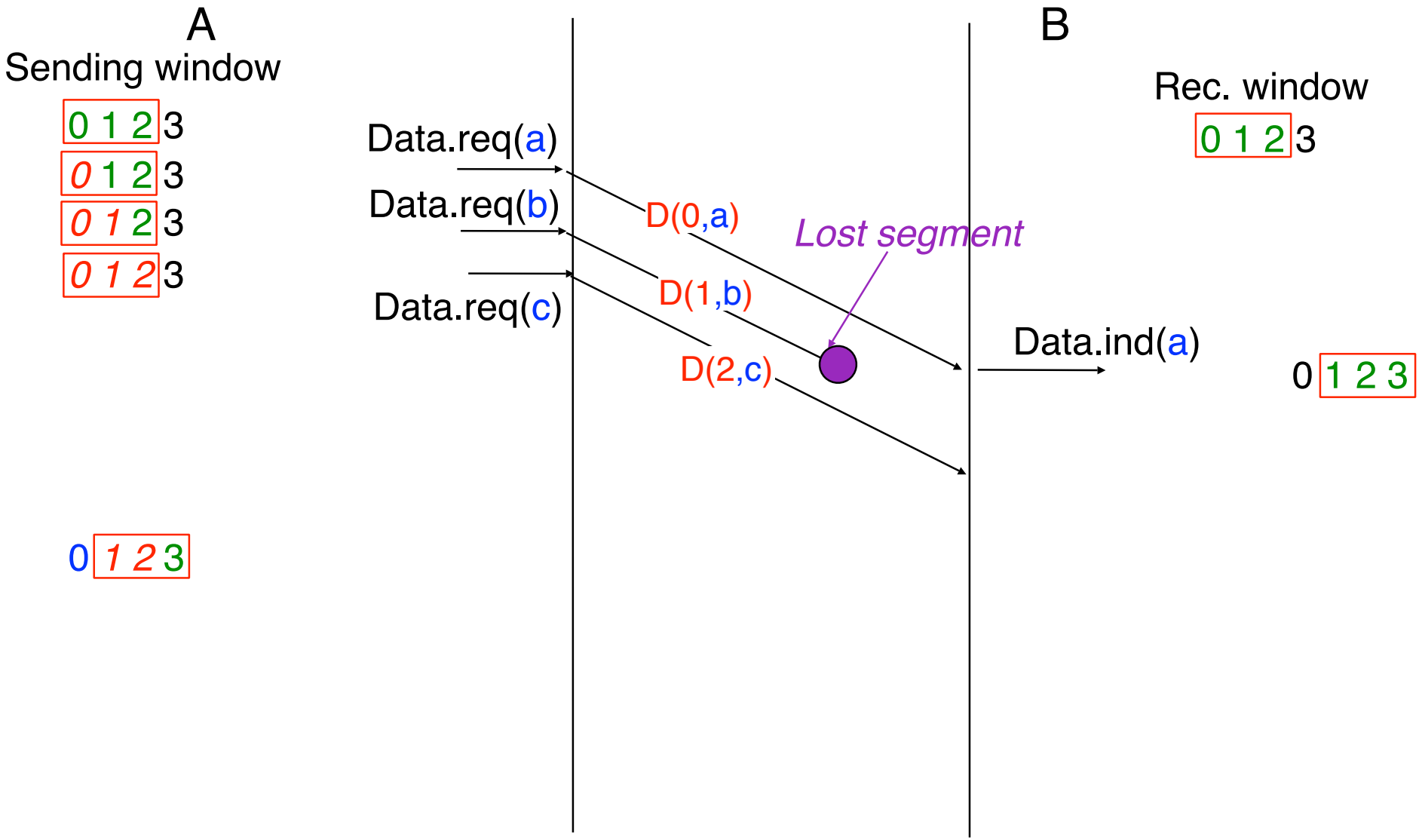
Selective Repeat : Example (2)



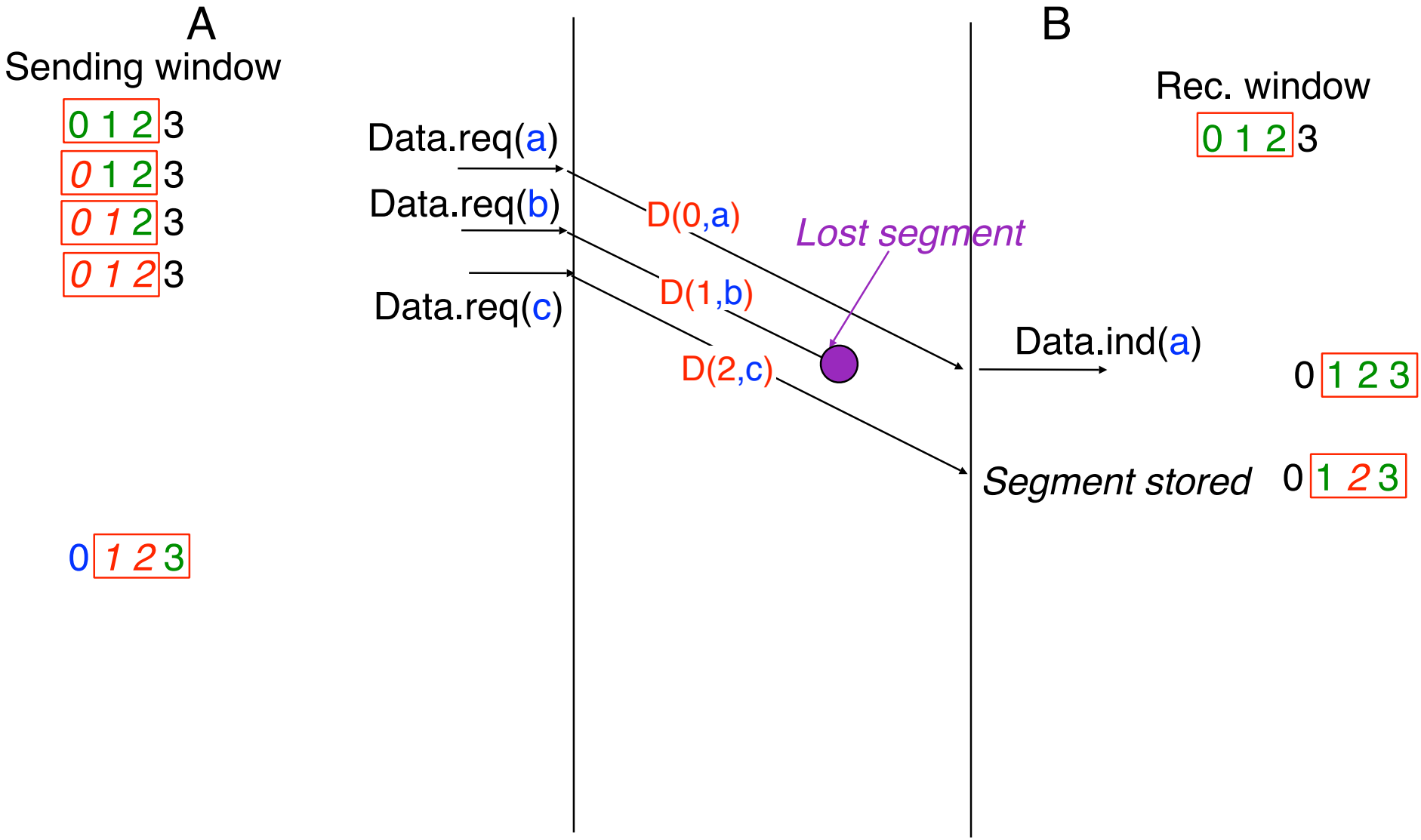
Selective Repeat : Example (2)



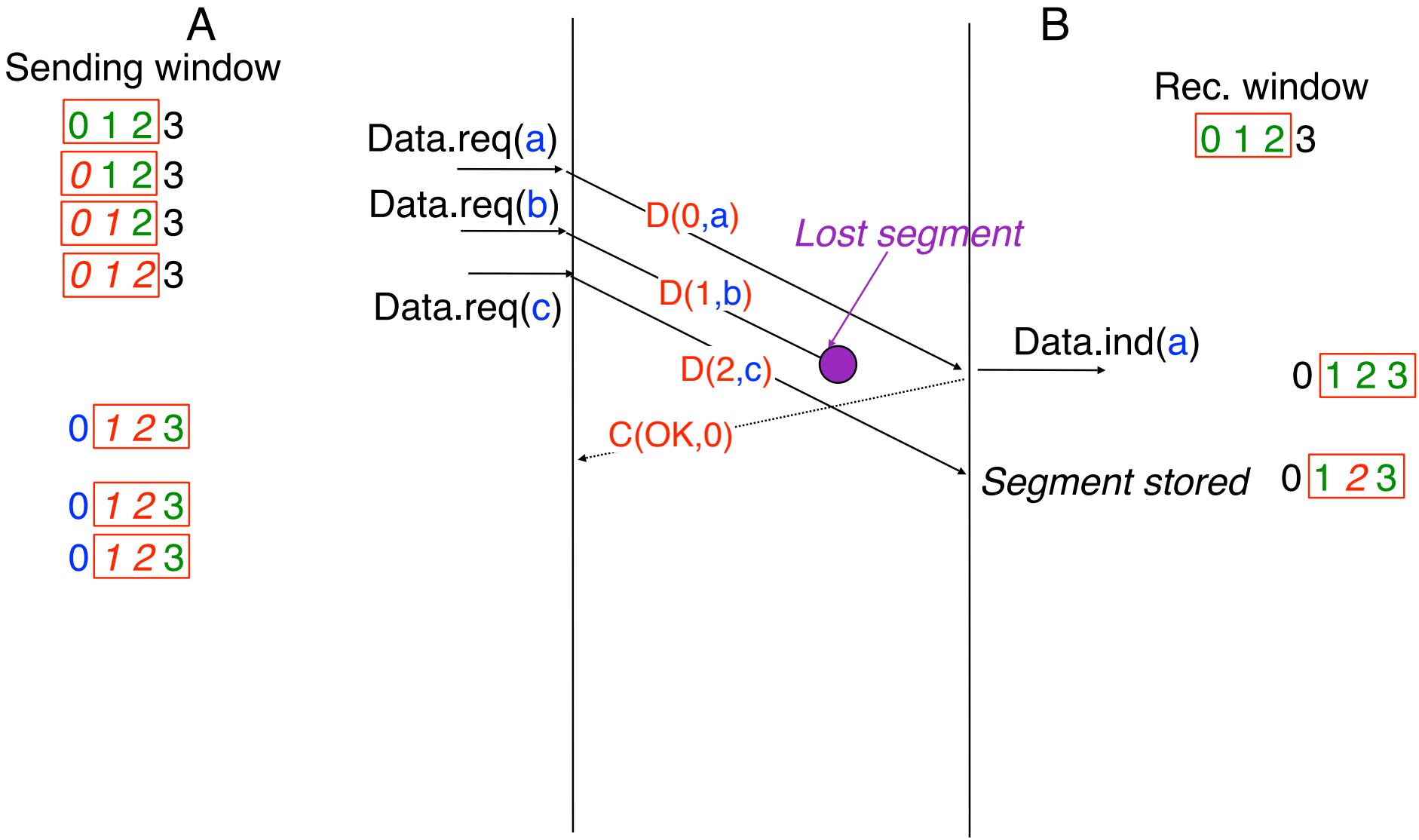
Selective Repeat : Example (2)



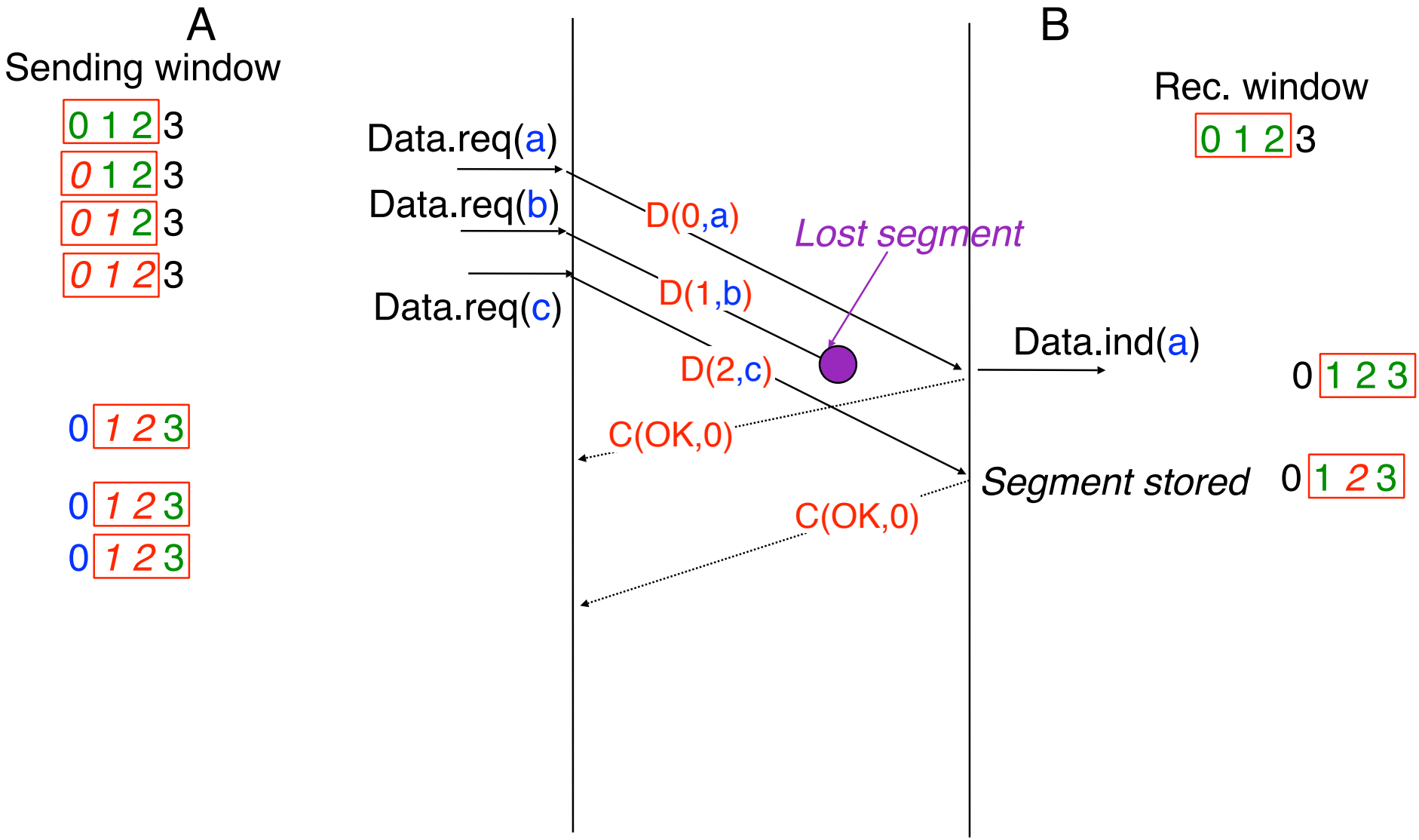
Selective Repeat : Example (2)



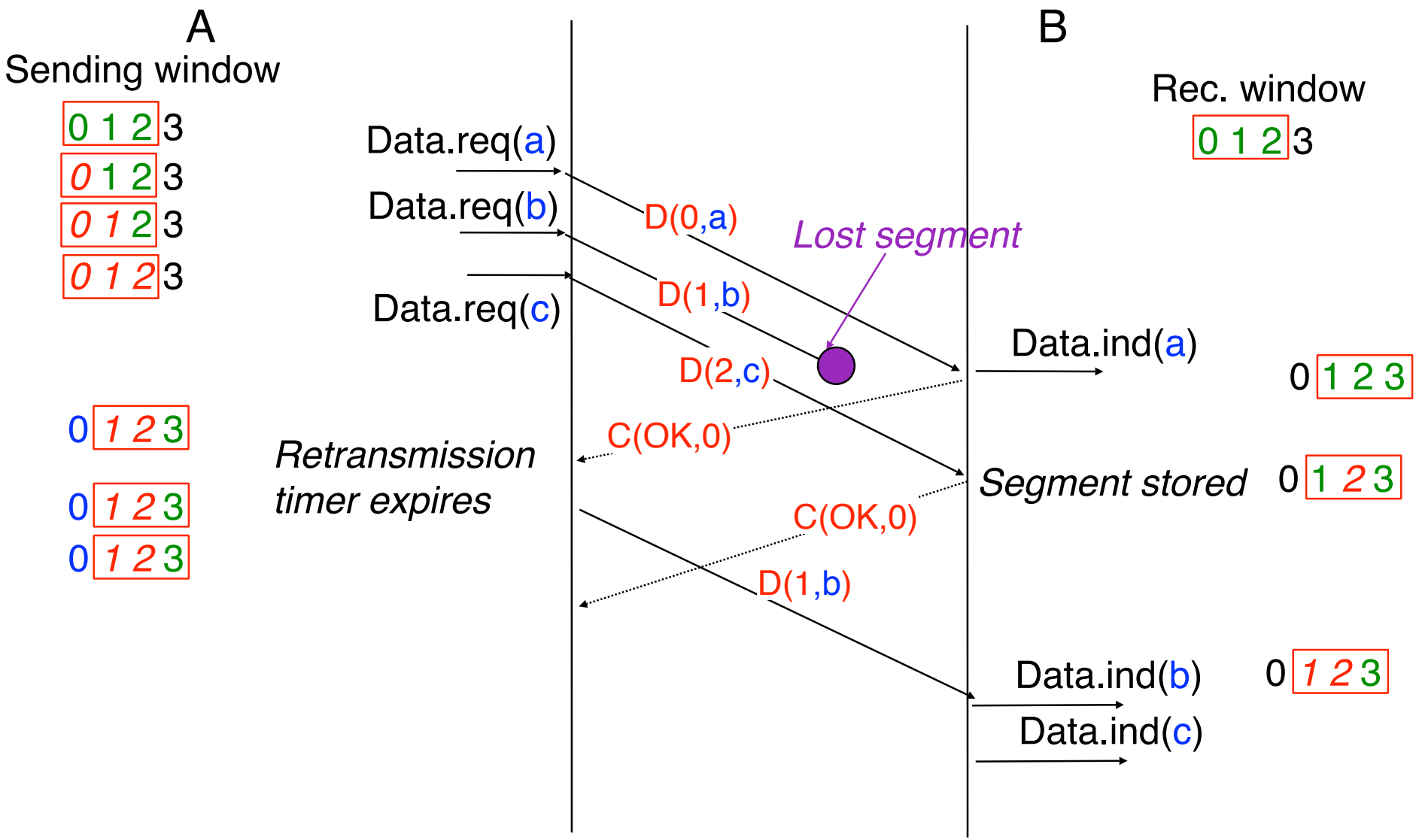
Selective Repeat : Example (2)



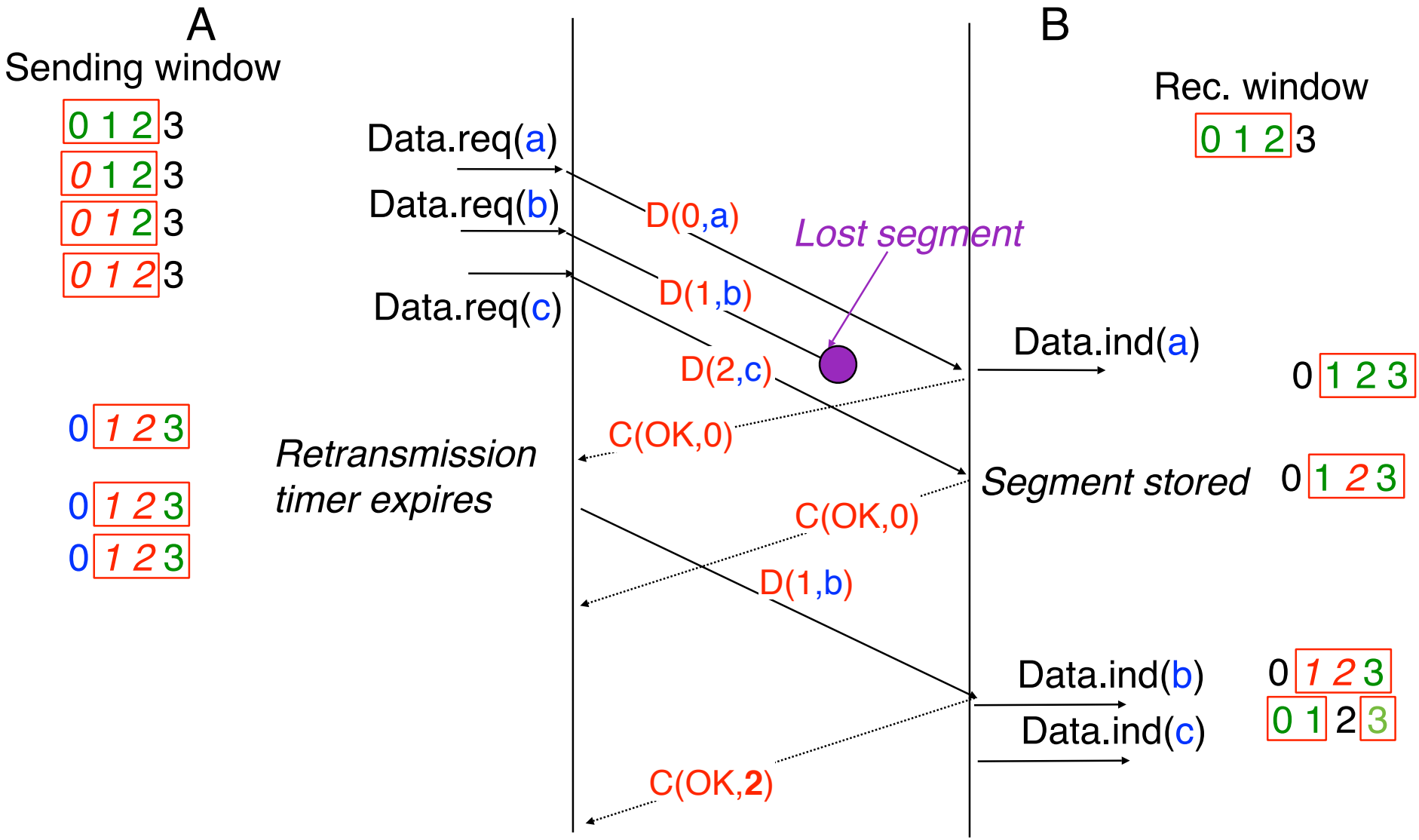
Selective Repeat : Example (2)



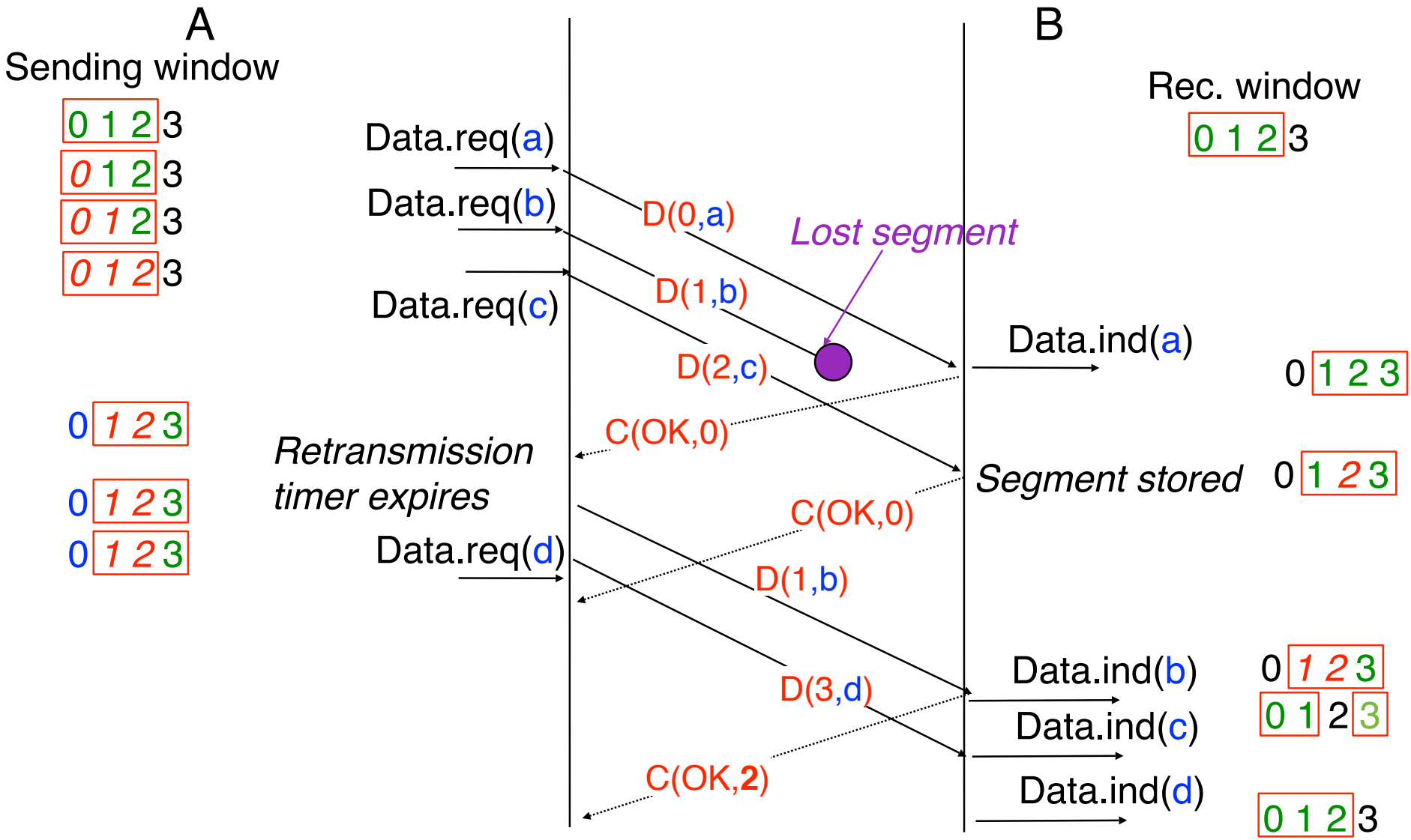
Selective Repeat : Example (2)



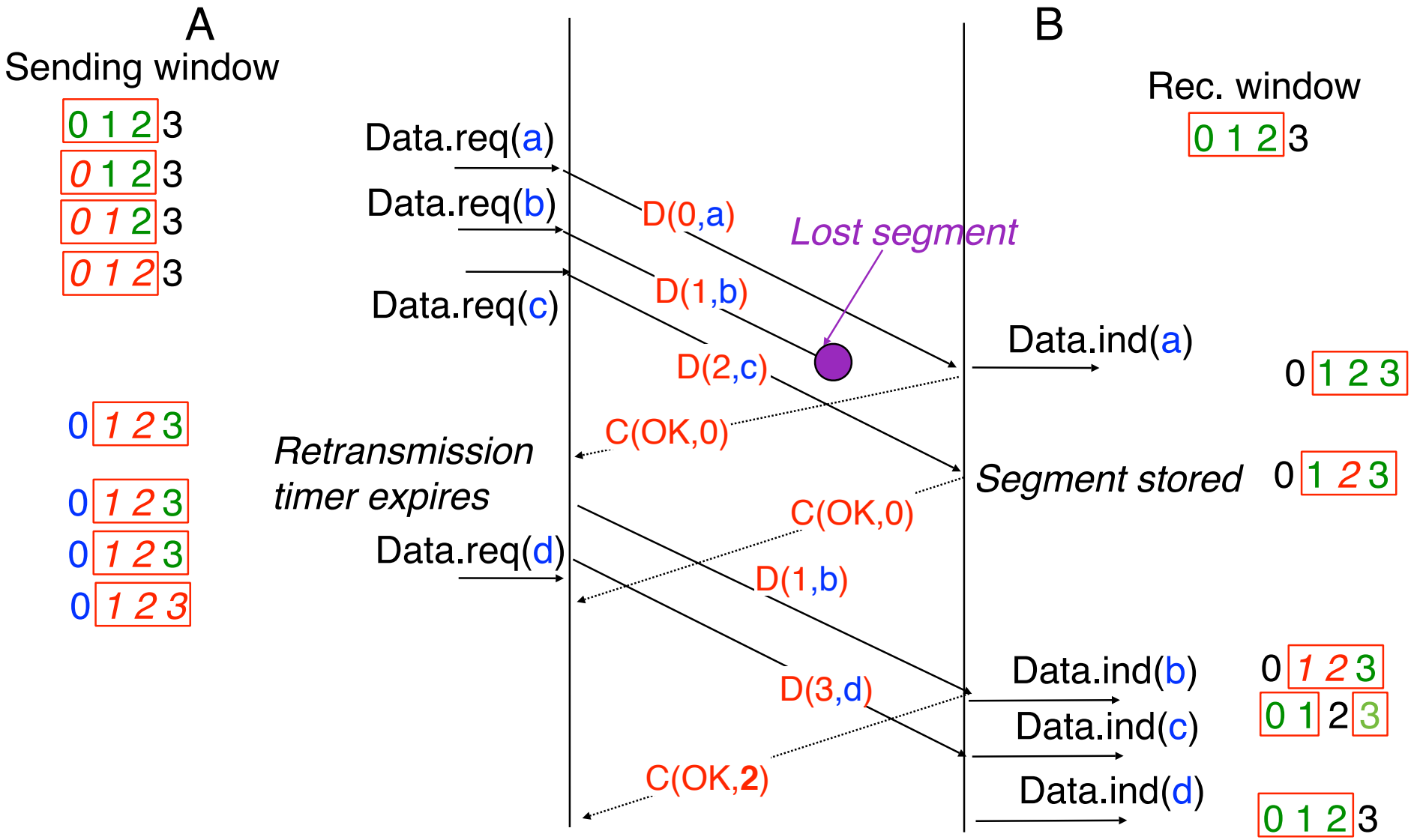
Selective Repeat : Example (2)



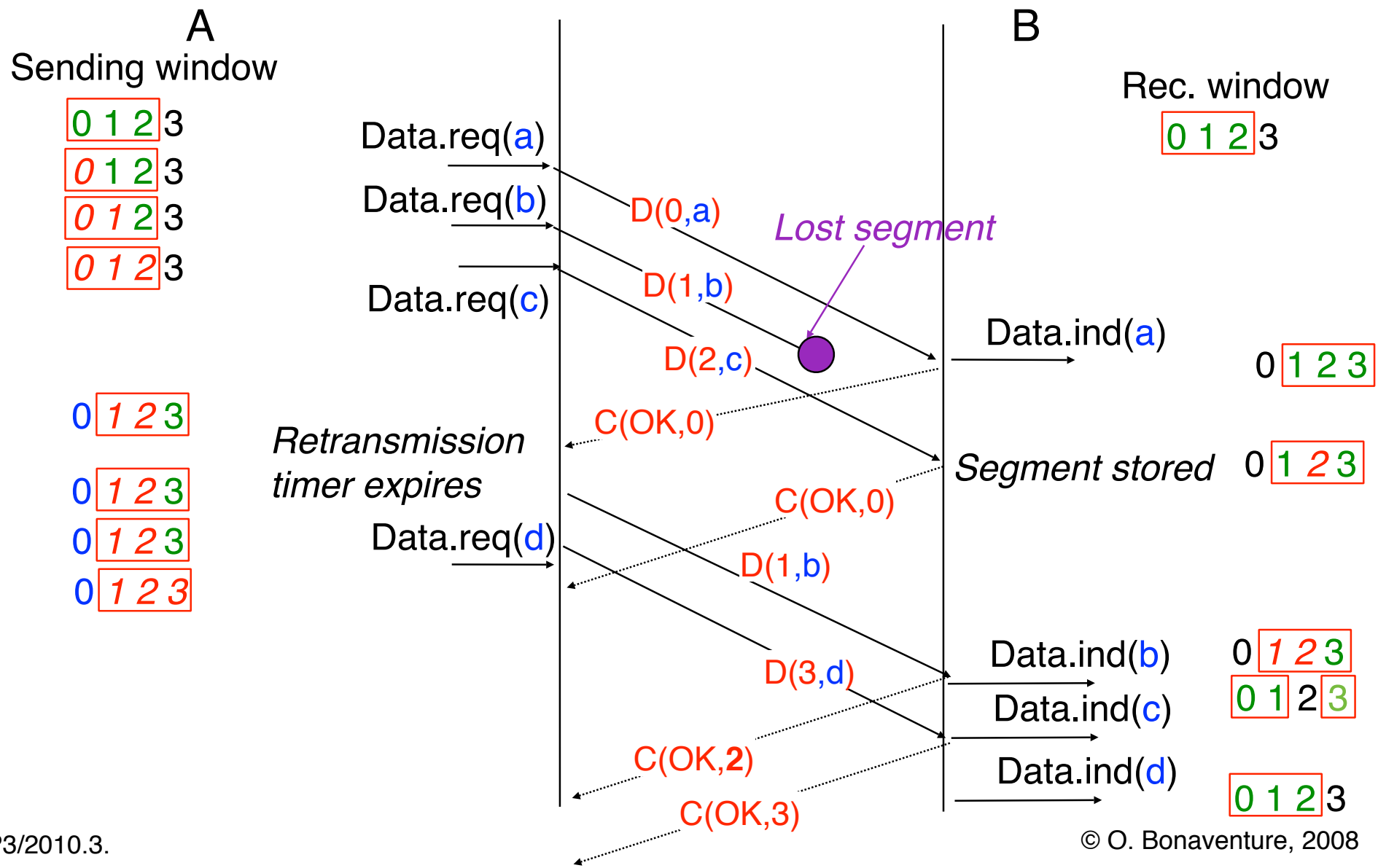
Selective Repeat : Example (2)



Selective Repeat : Example (2)

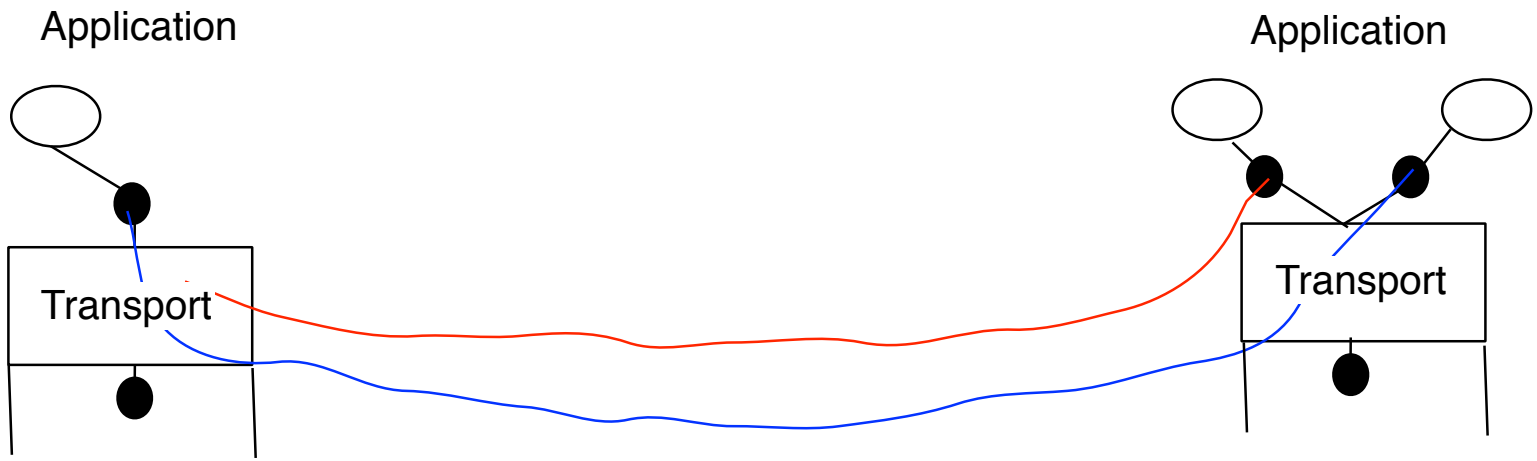


Selective Repeat : Example (2)



Buffer management

Problem



A transport entity may support many transport connections at the same time

- How can we share the available buffer among these connections ?
- The number of connections changes with time
- Some connections require large buffers while others can easily use smaller ones
  - `ftp` versus `telnet`

Buffer management (2)

Principle

Adjust the size of the receiving window according to the amount of buffering available on the receiver
Allow the receiver to advertise its current receiving window size to the sender

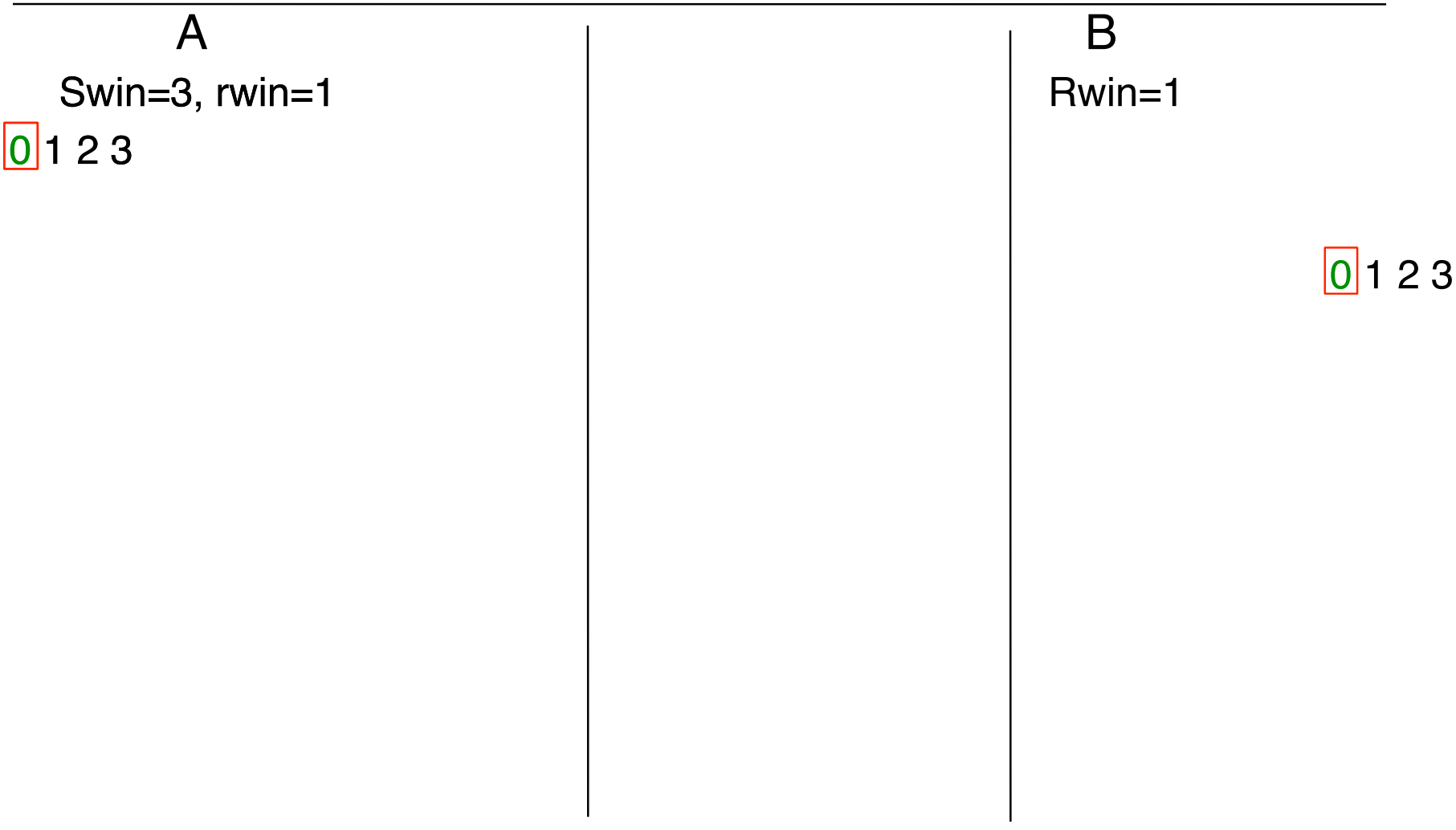
New information carried in control segments

`win` indicates the current receiving window's size

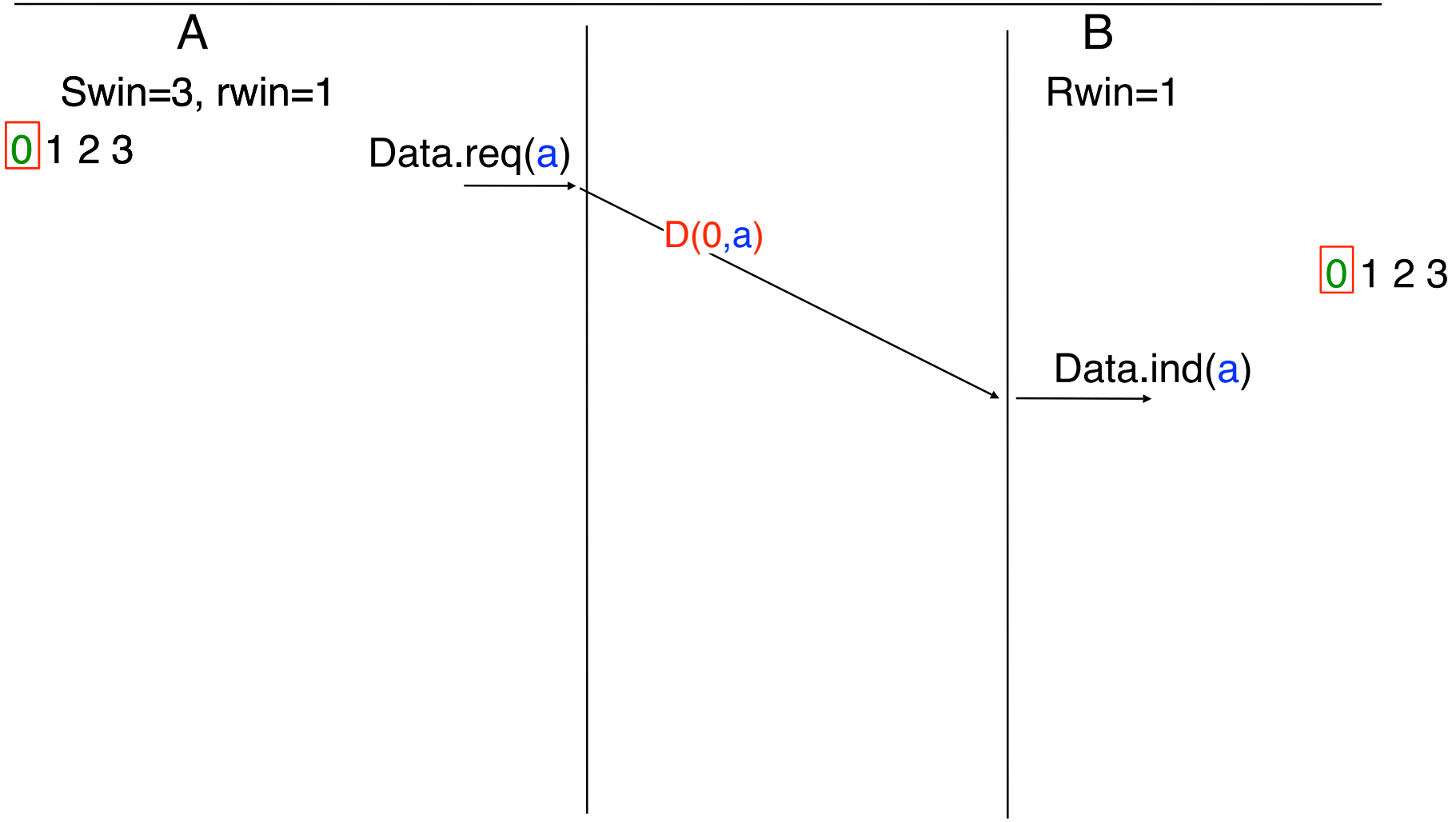
Changes to sender

Sending window : `swin` (function of available memory)
Keep in a state variable the receiving window advertised by the receiver : `rwin`
At any time, the sender is only allowed to send data segments whose sequence number fits inside
min(`rwin`, `swin`)

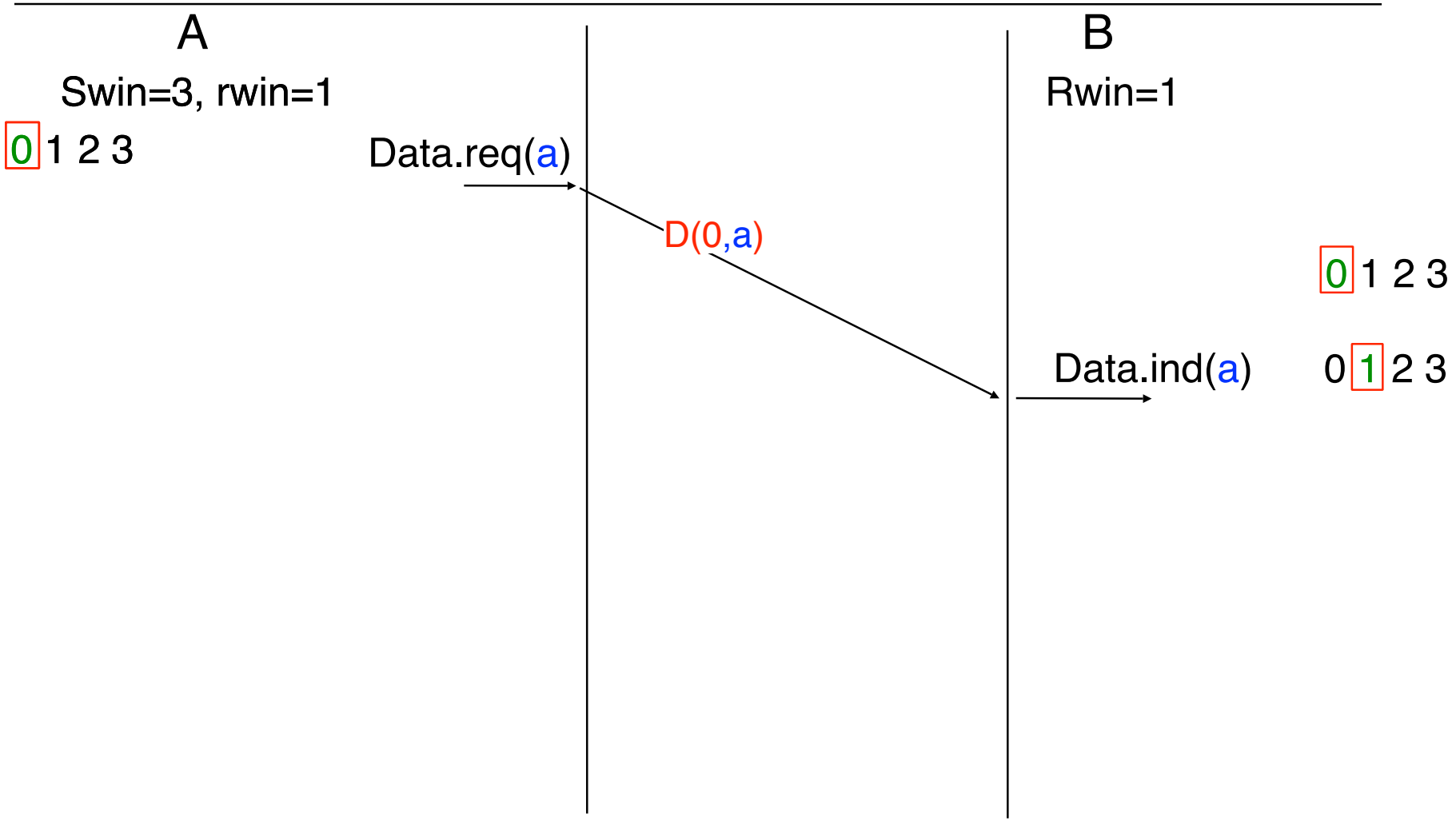
Buffer management (3)



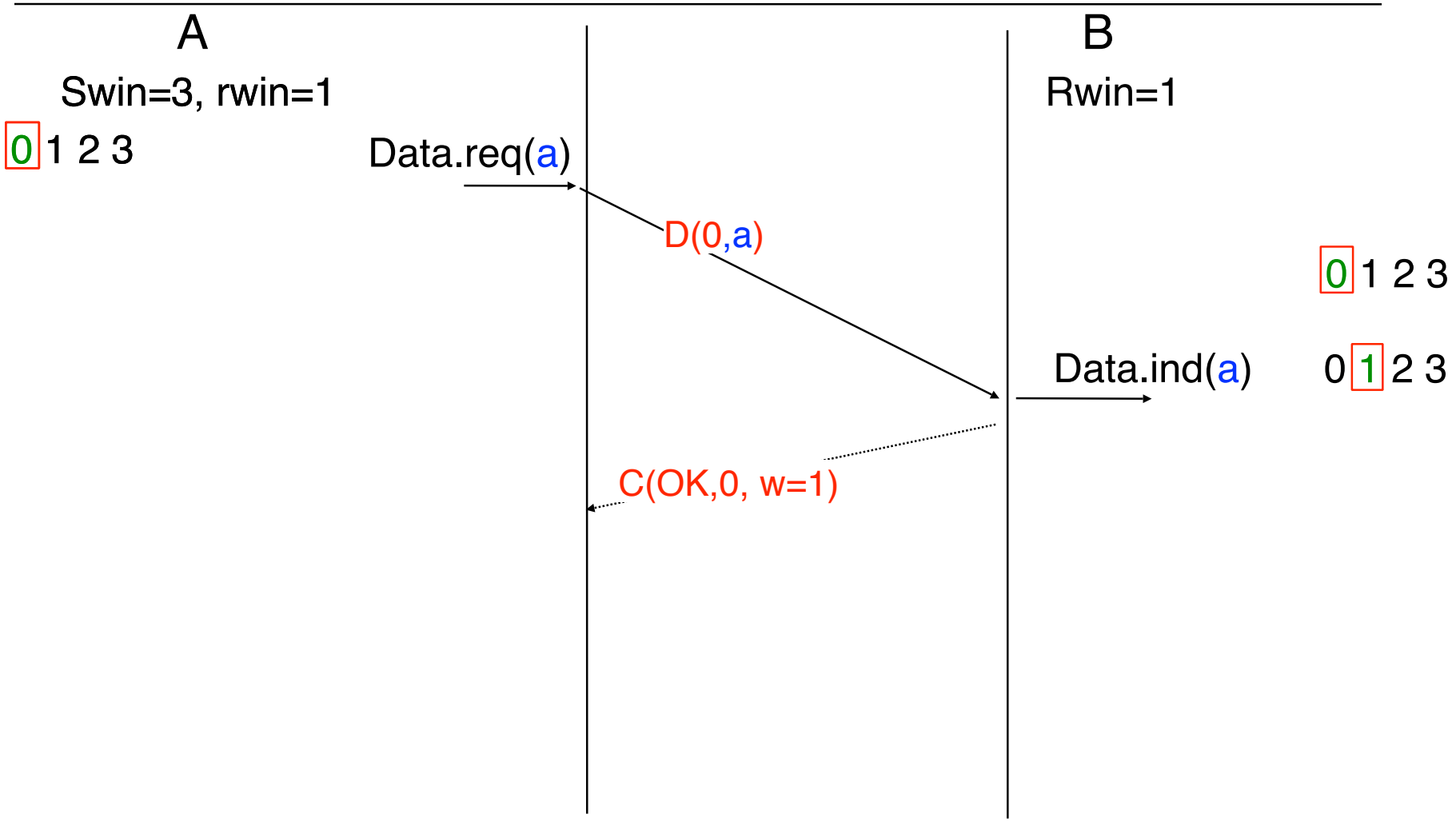
Buffer management (3)



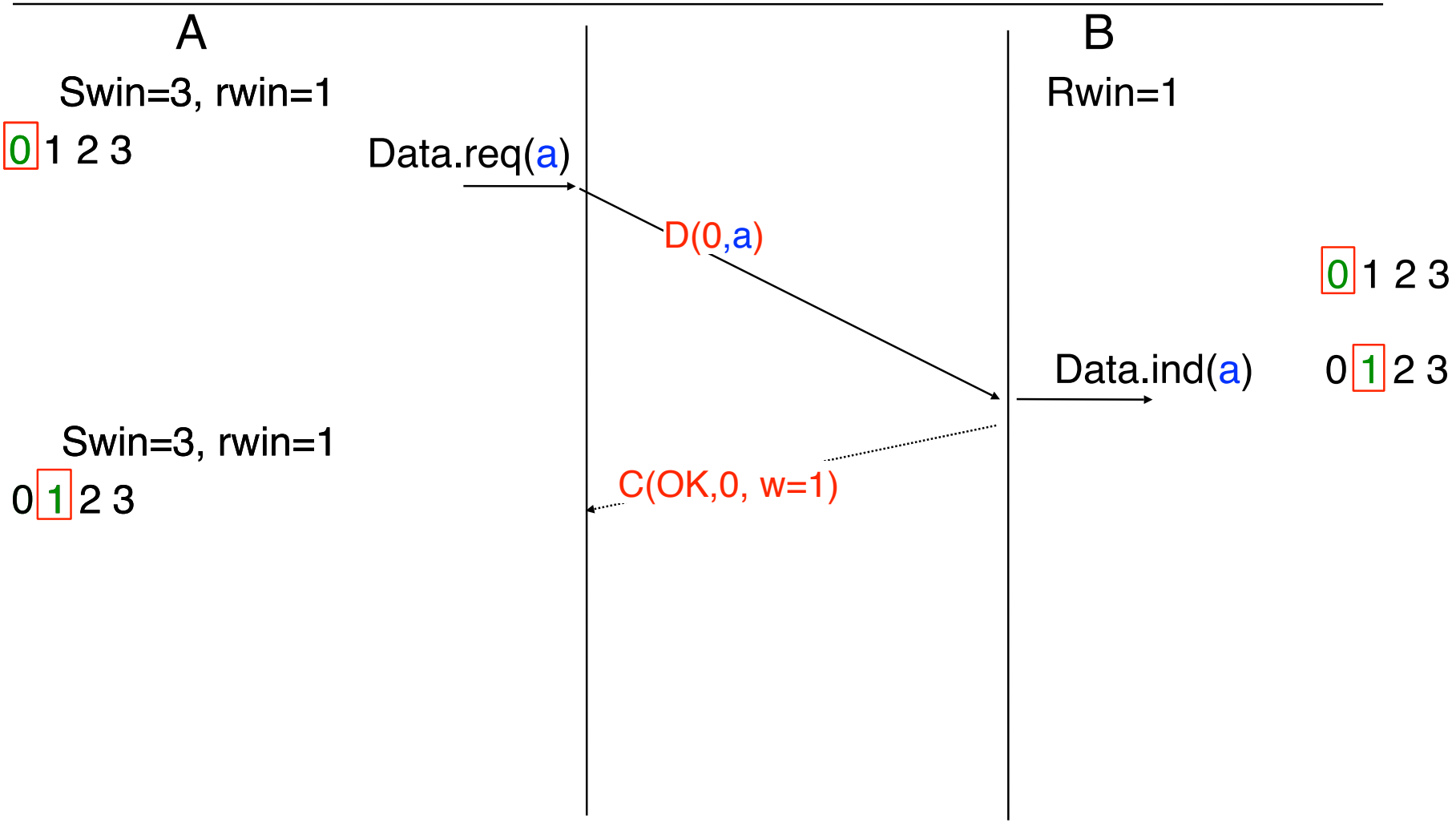
Buffer management (3)



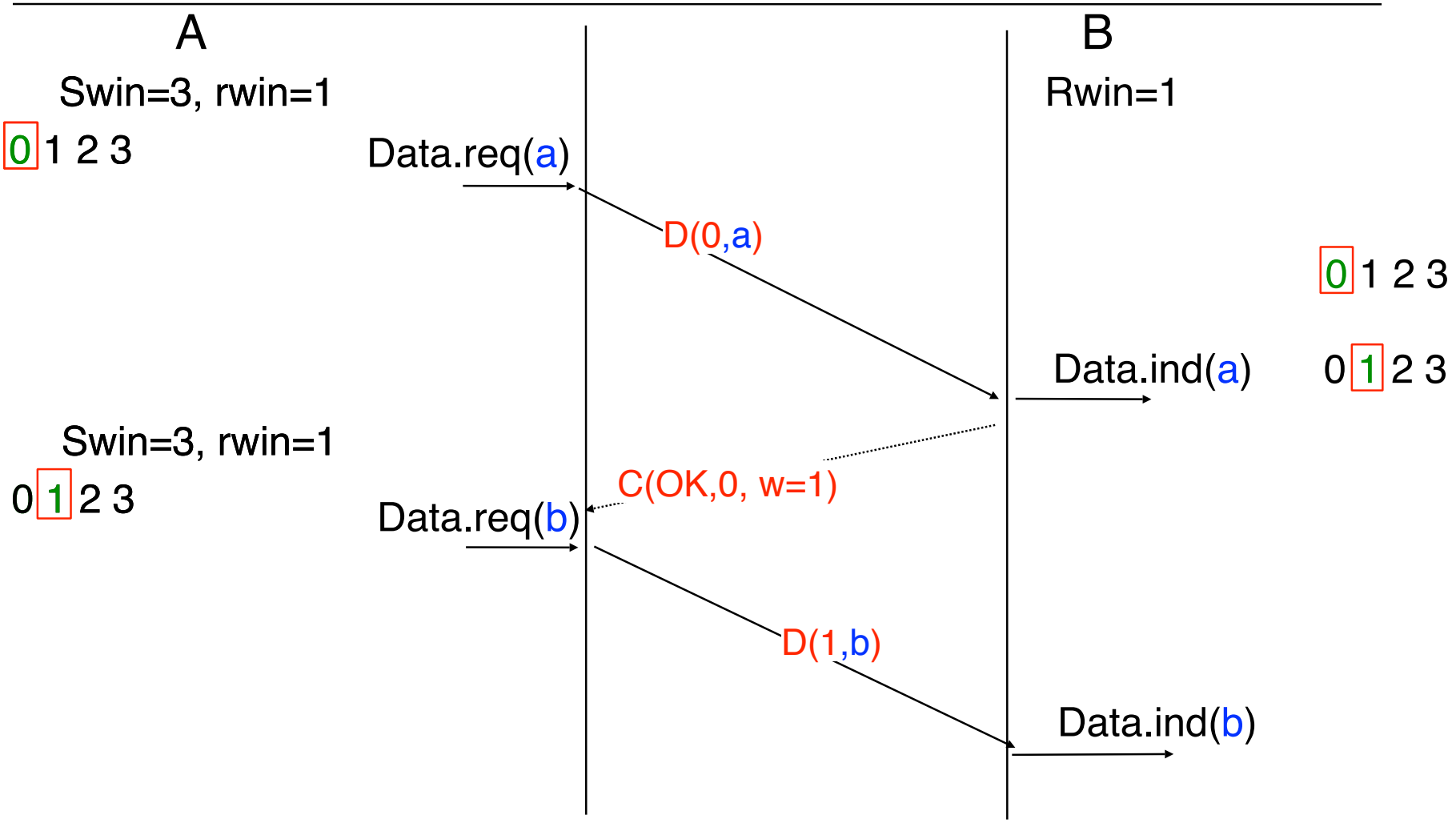
Buffer management (3)



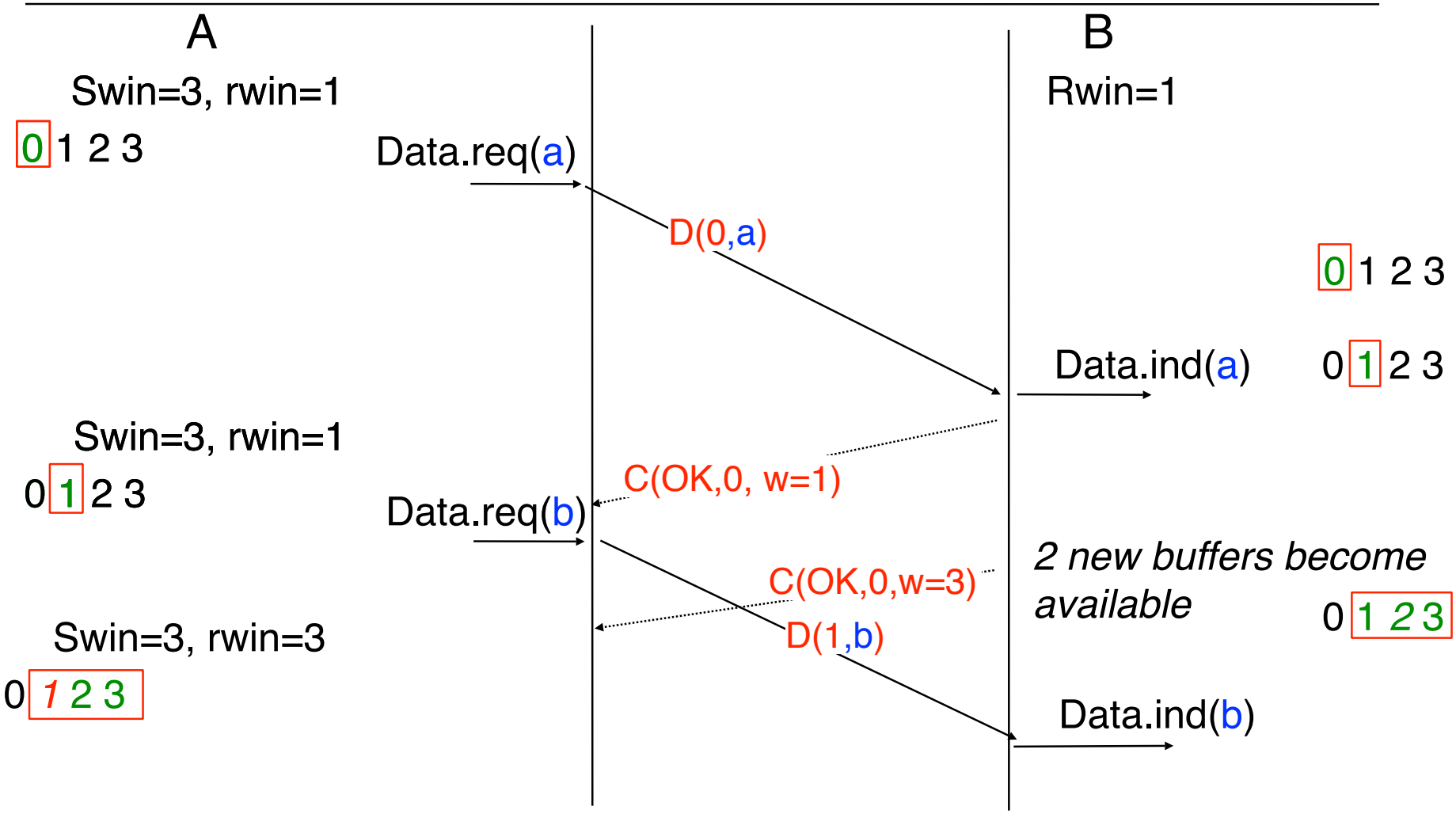
Buffer management (3)



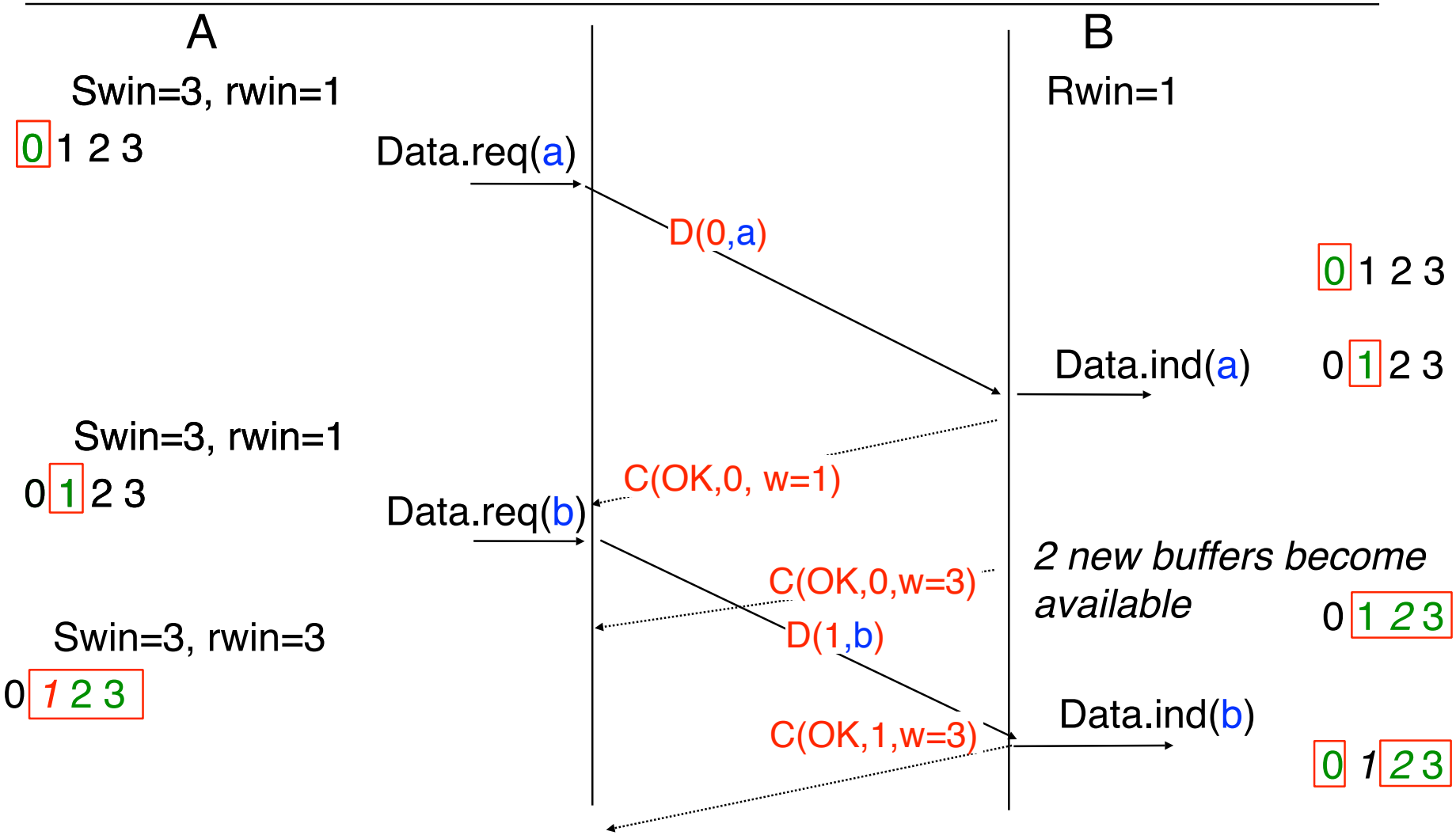
Buffer management (3)



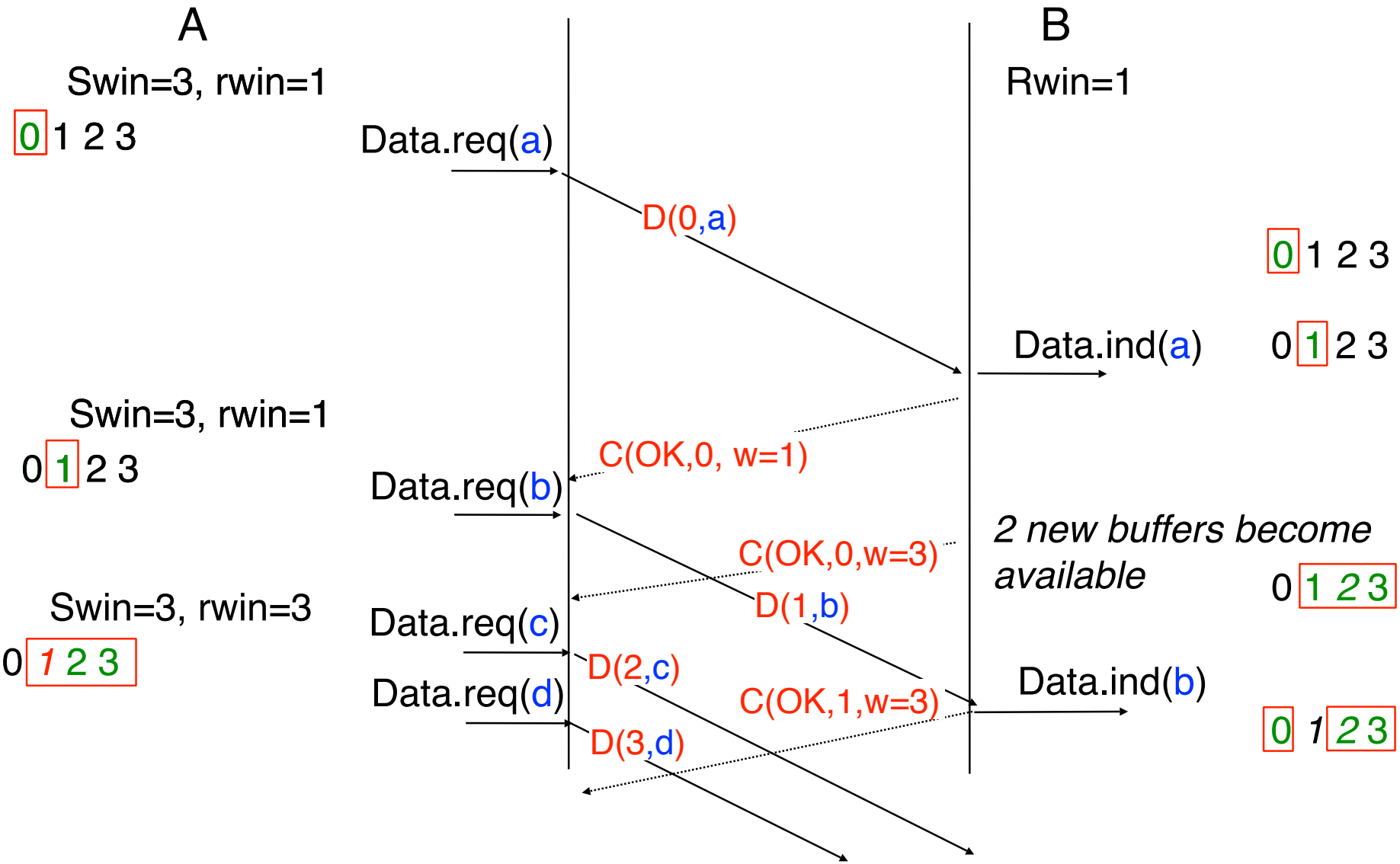
Buffer management (3)



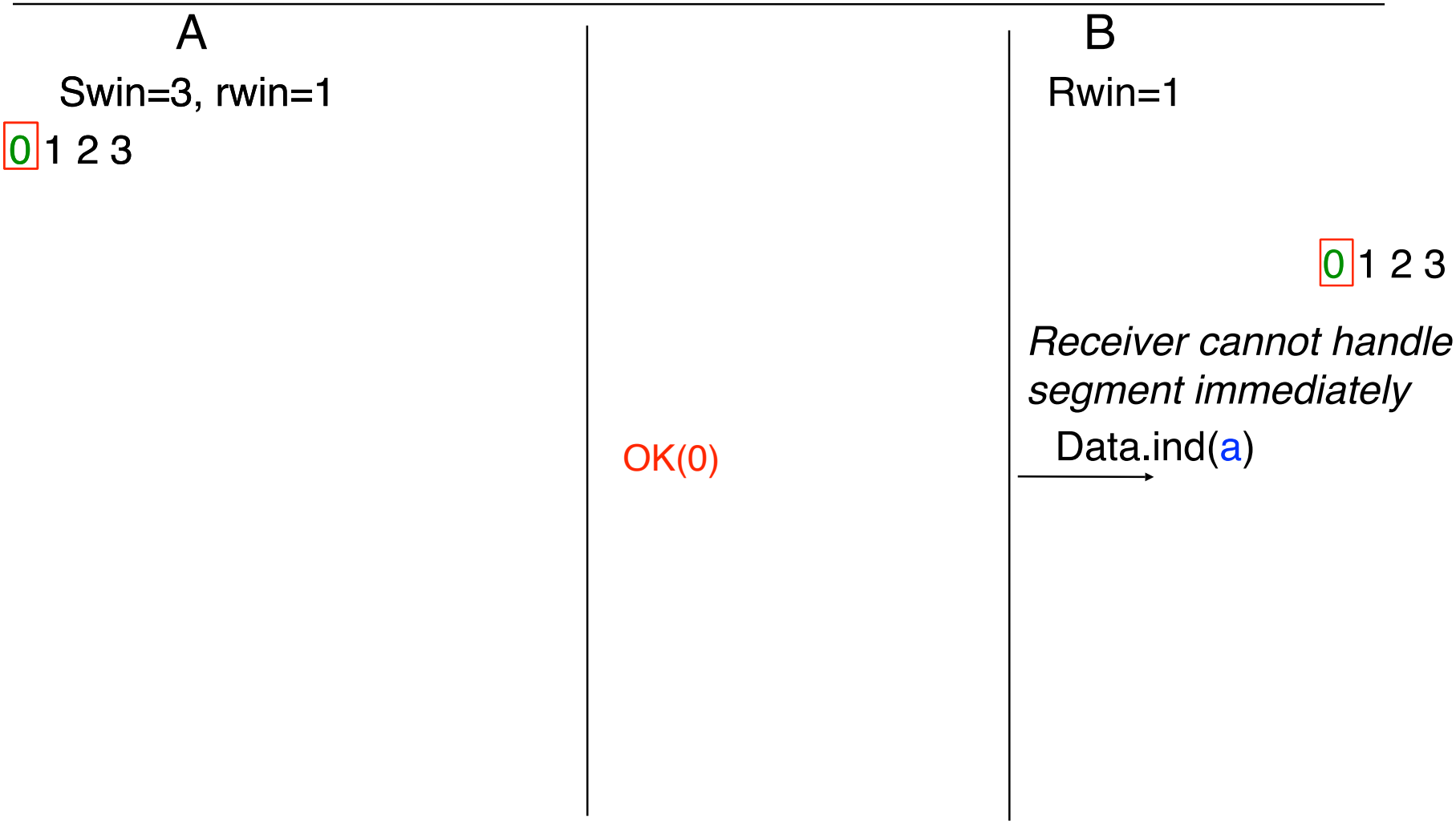
Buffer management (3)



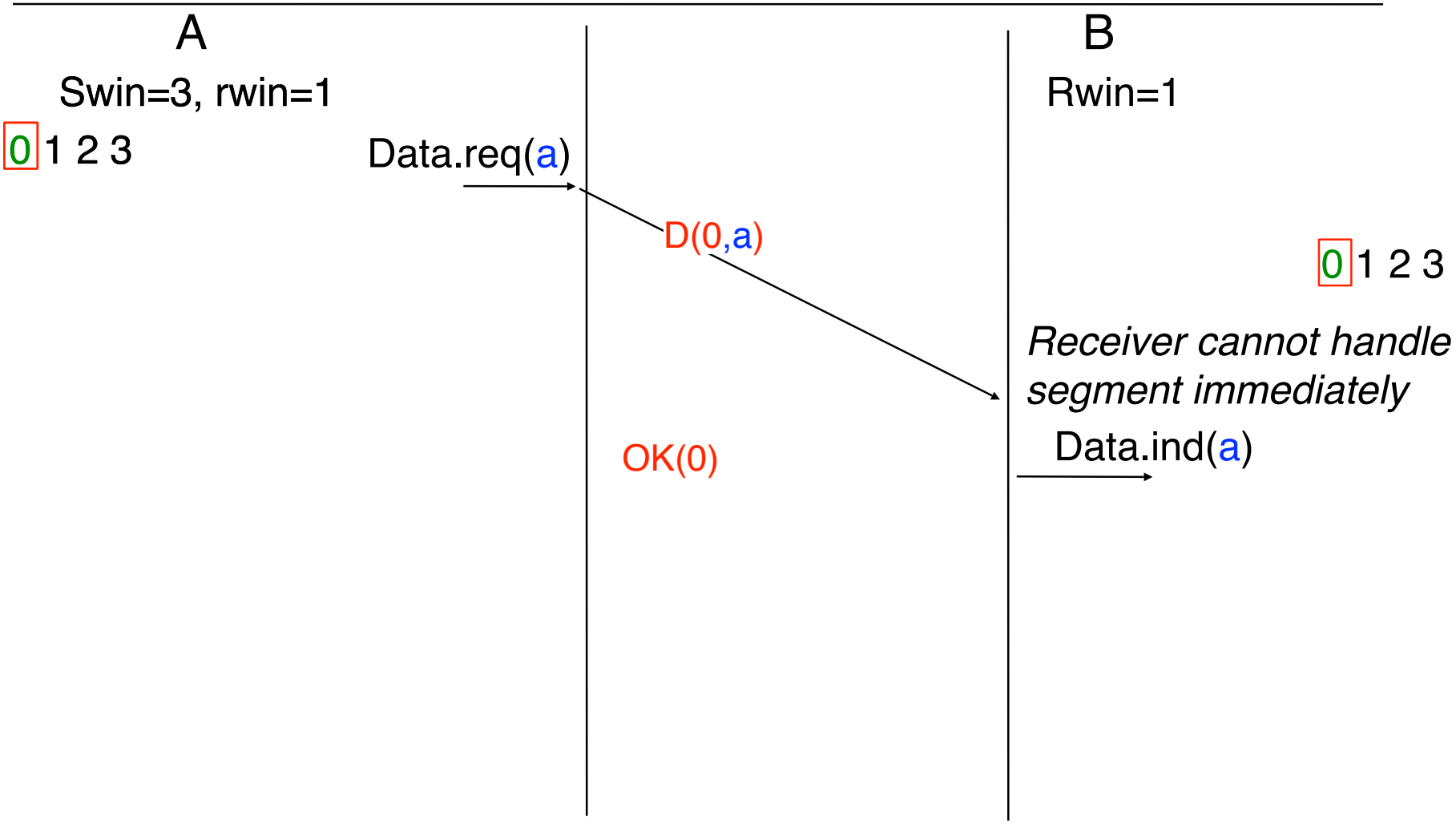
Buffer management (3)



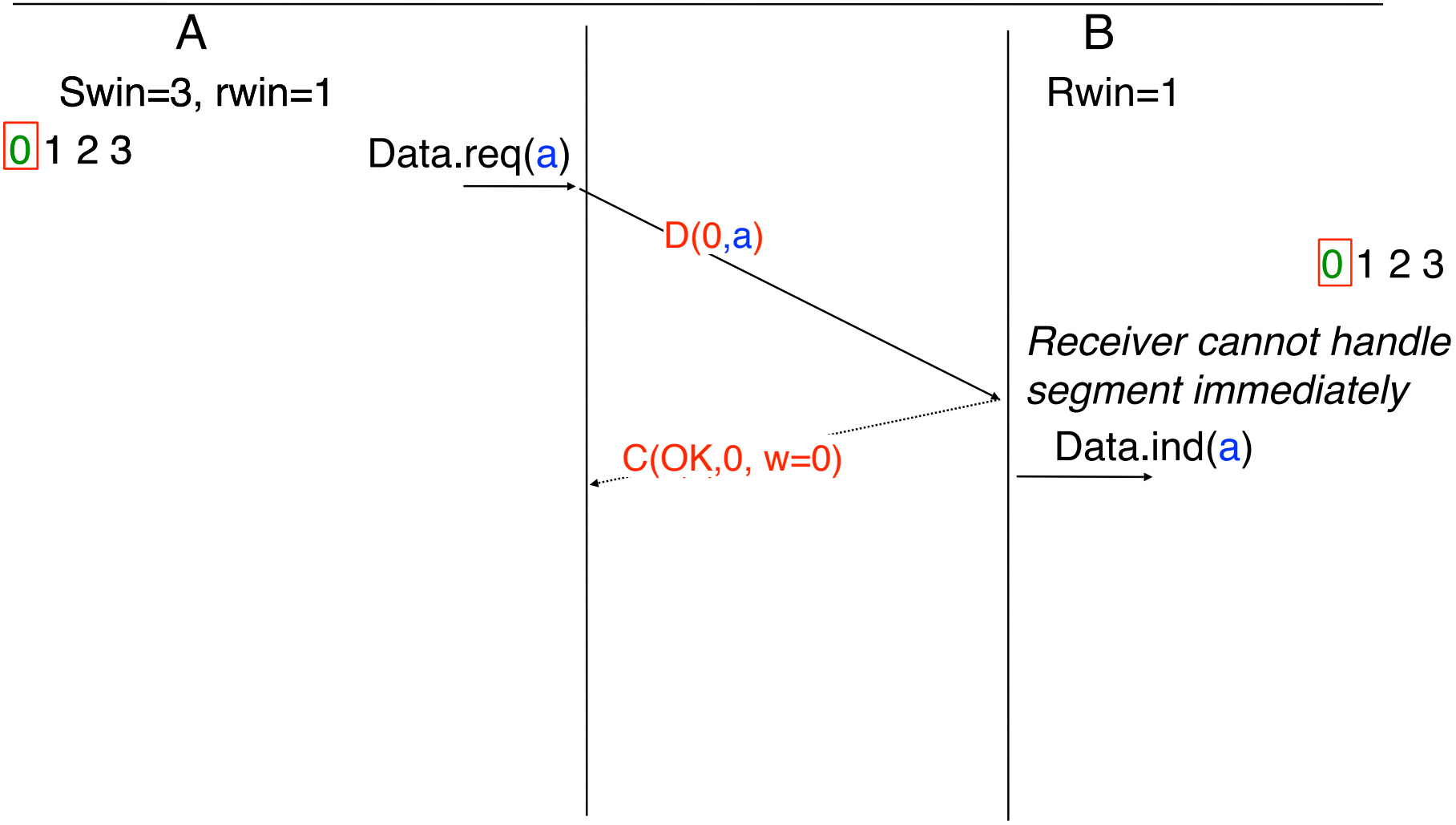
Buffer management (4)



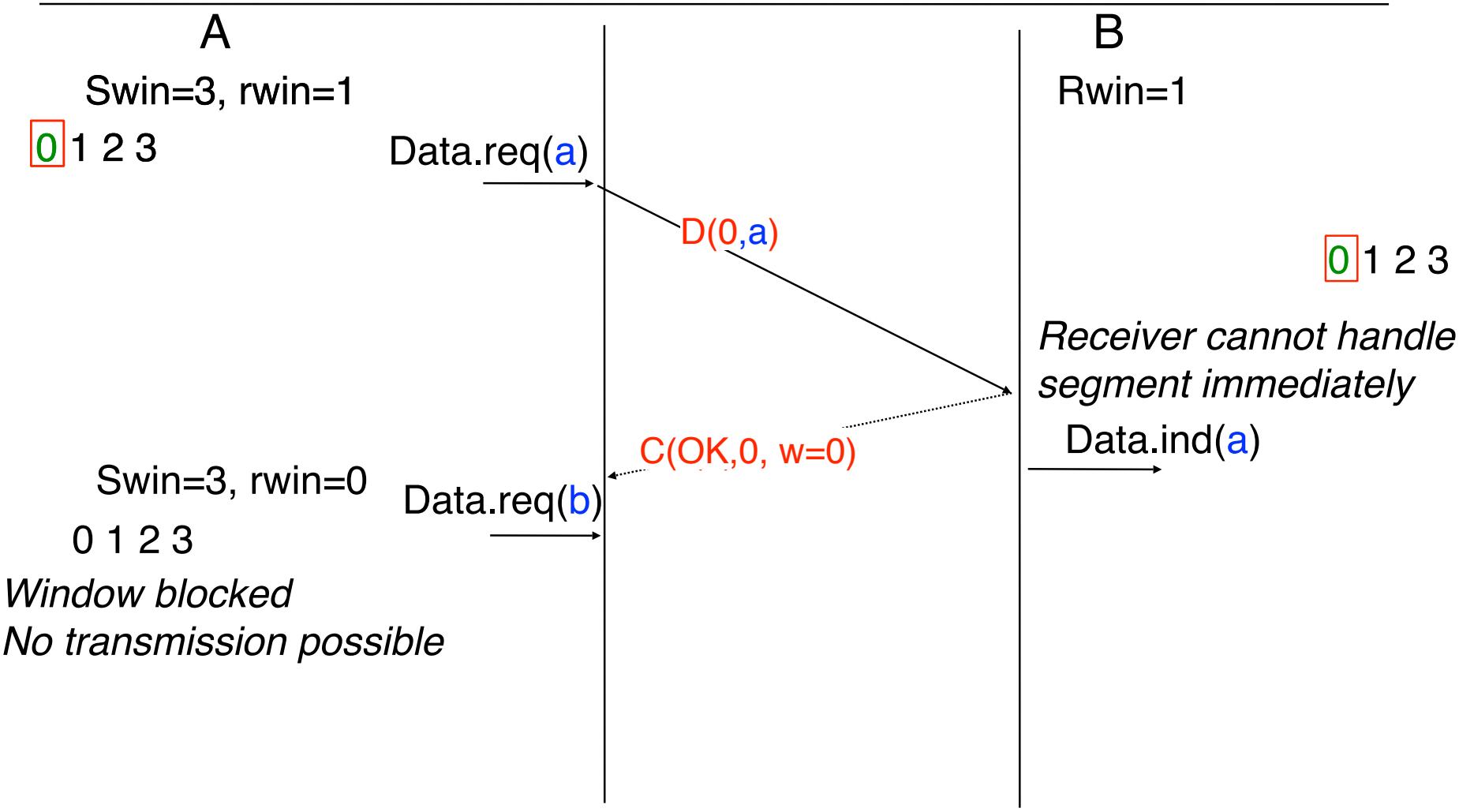
Buffer management (4)



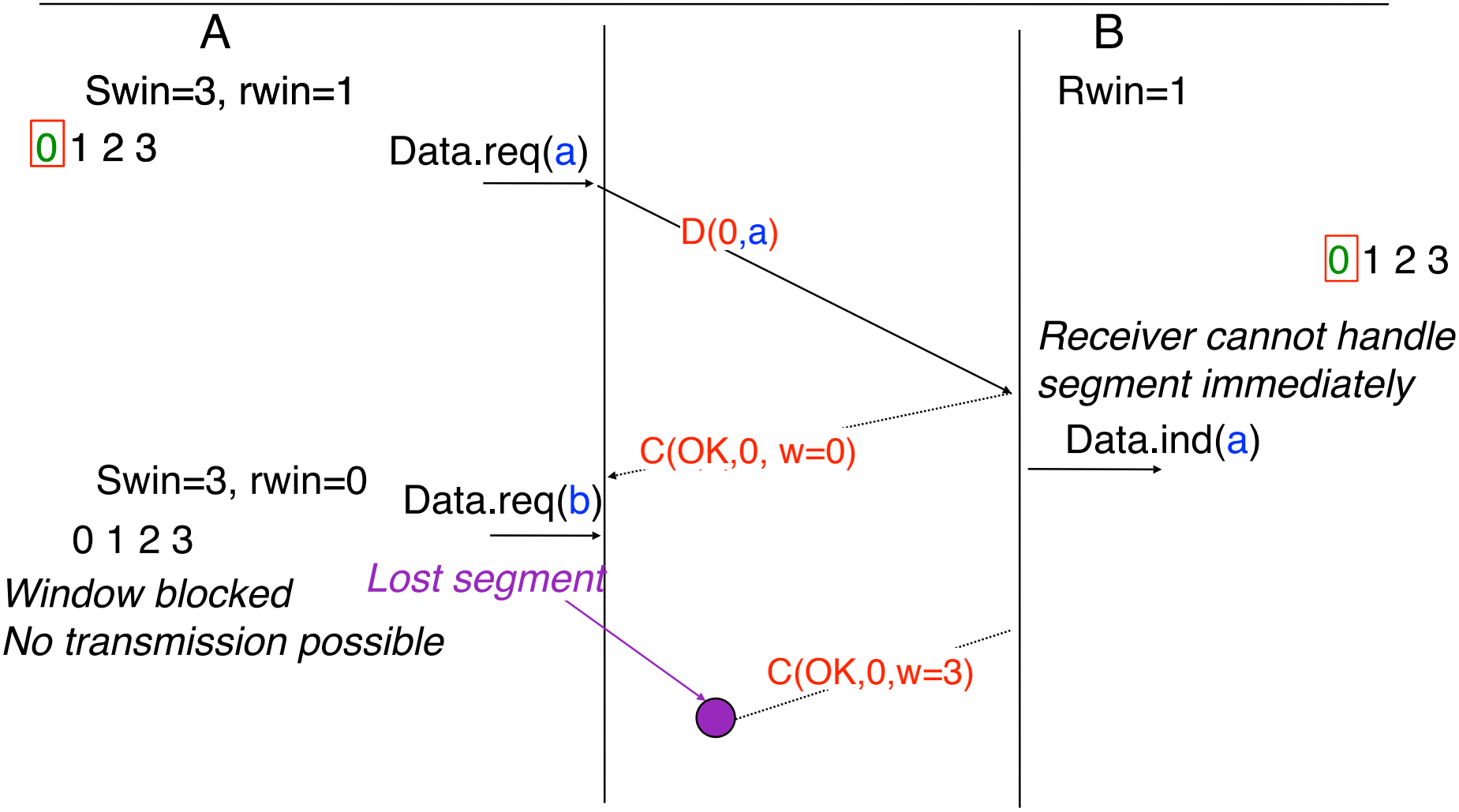
Buffer management (4)



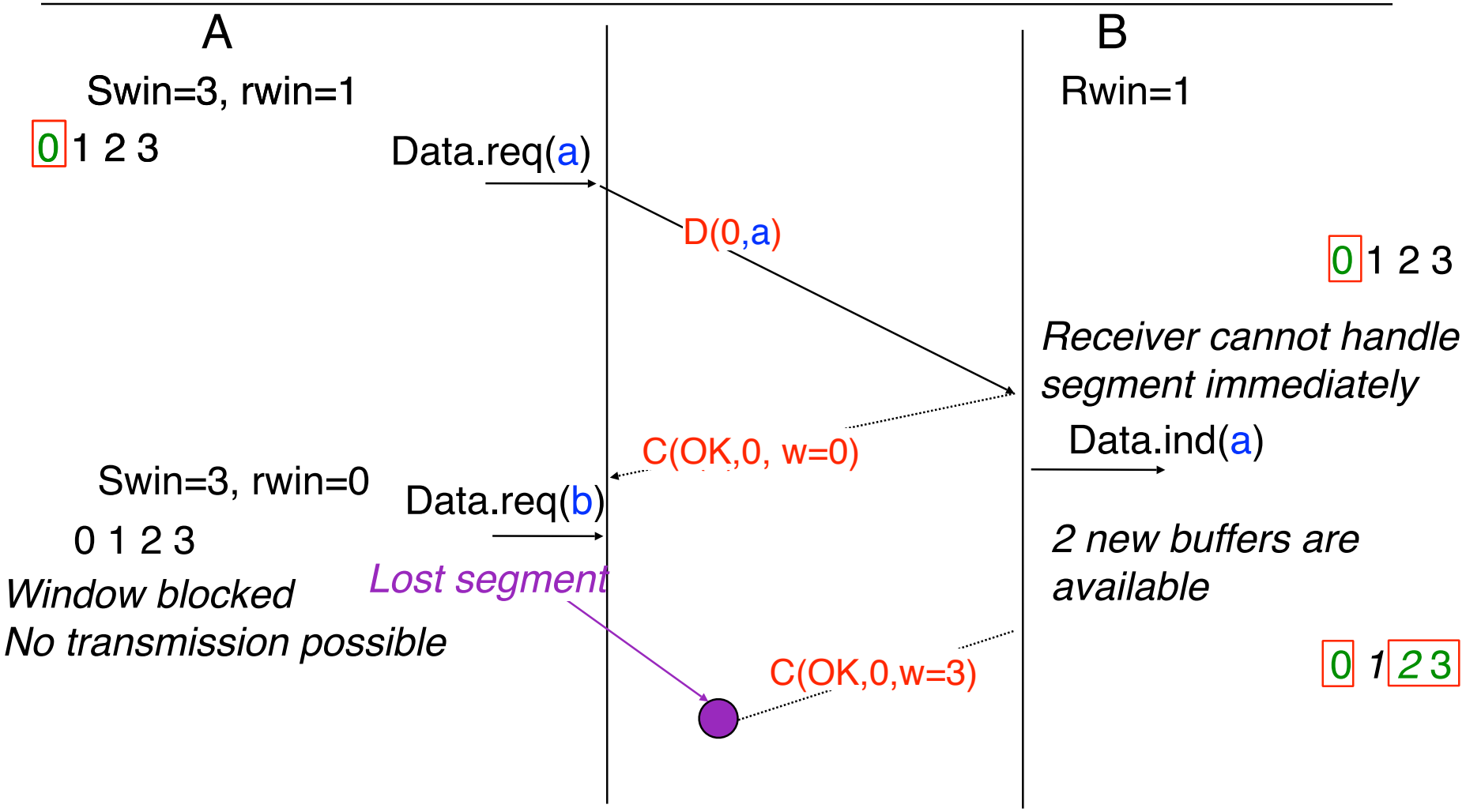
Buffer management (4)



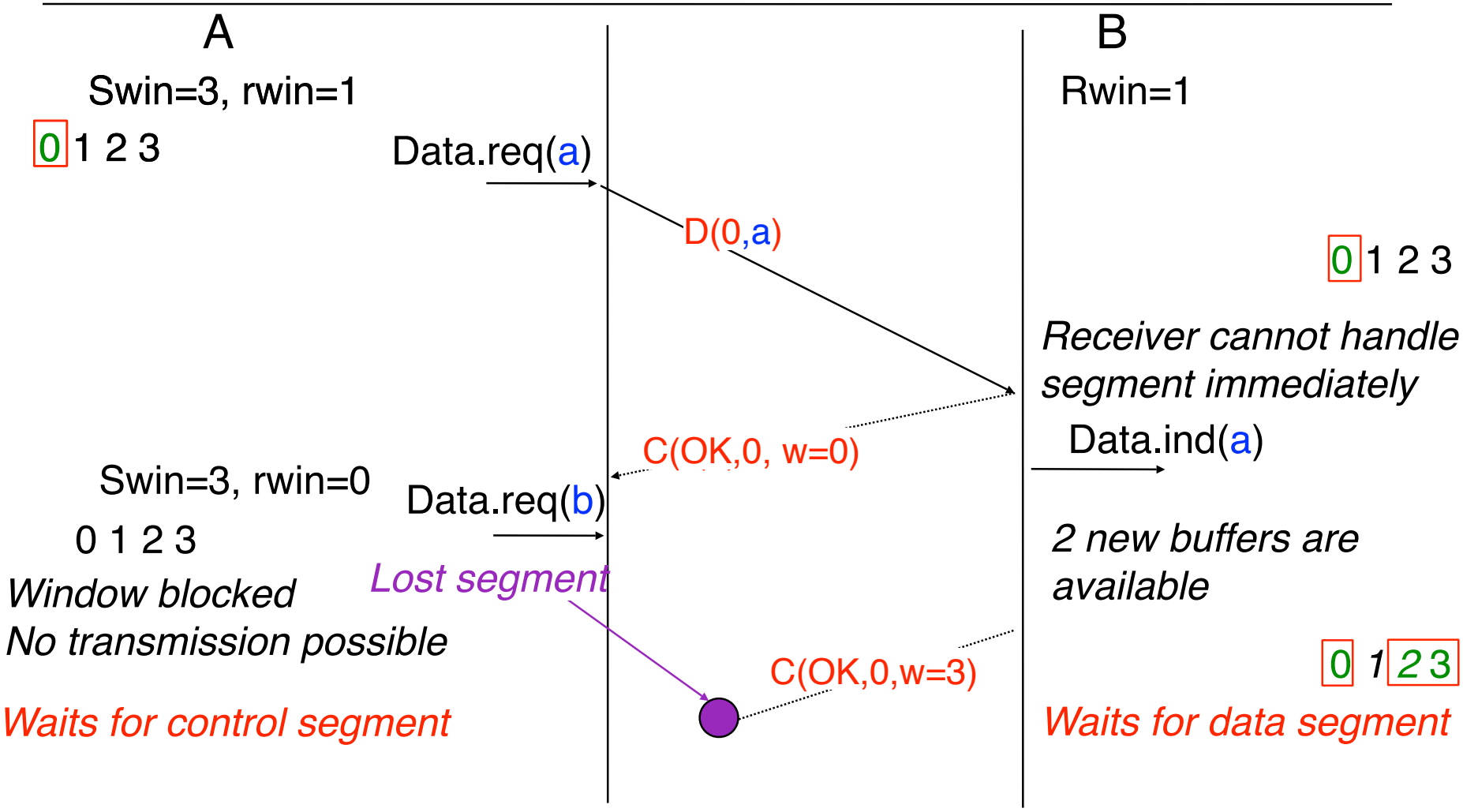
Buffer management (4)



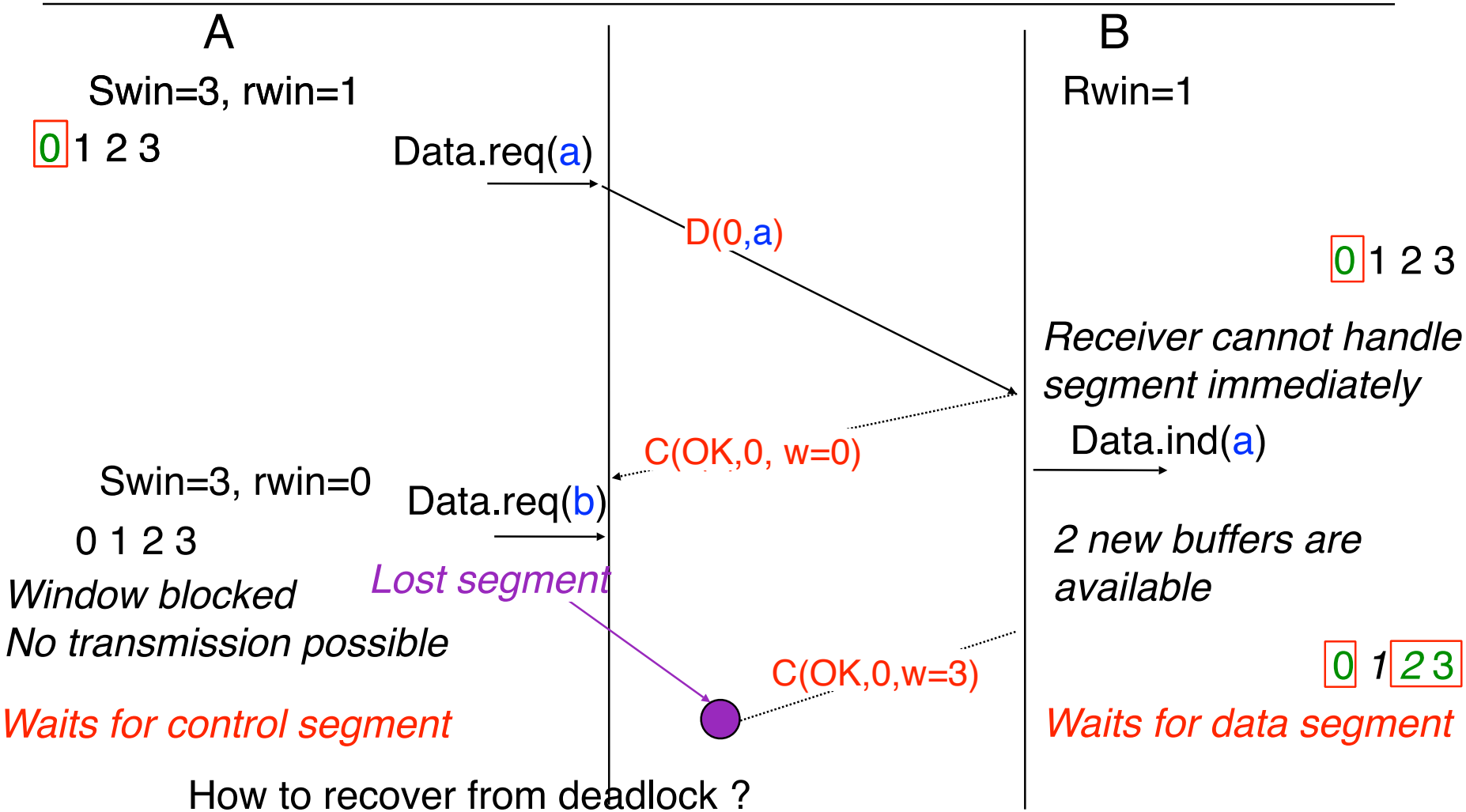
Buffer management (4)



Buffer management (4)

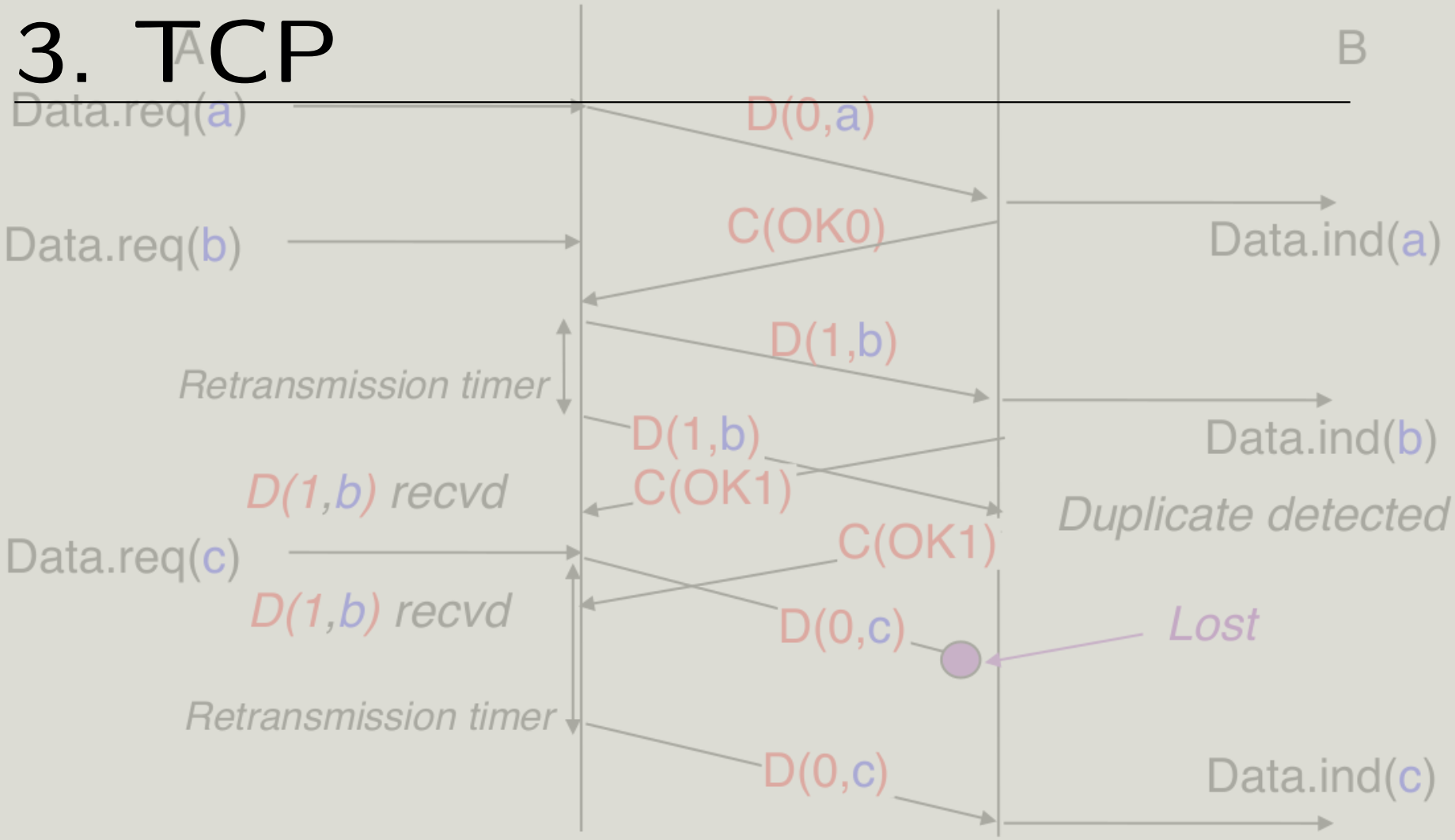


Buffer management (4)



How to recover from deadlock ?
Persitence timer on receiver, resend control segment after timer expiration

3. TCP


A
B
Data.req(a)
D(0,a)
Data.req(b)
C(OK0)
Data.ind(a)
D(1,b)
Retransmission timer
D(1,b)
Data.ind(b)
D(1,b) recvd
C(OK1)
Duplicate detected
Data.req(c)
C(OK1)
D(1,b) recvd
D(0,c)
Lost
Retransmission timer
D(0,c)
Data.ind(c)

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.

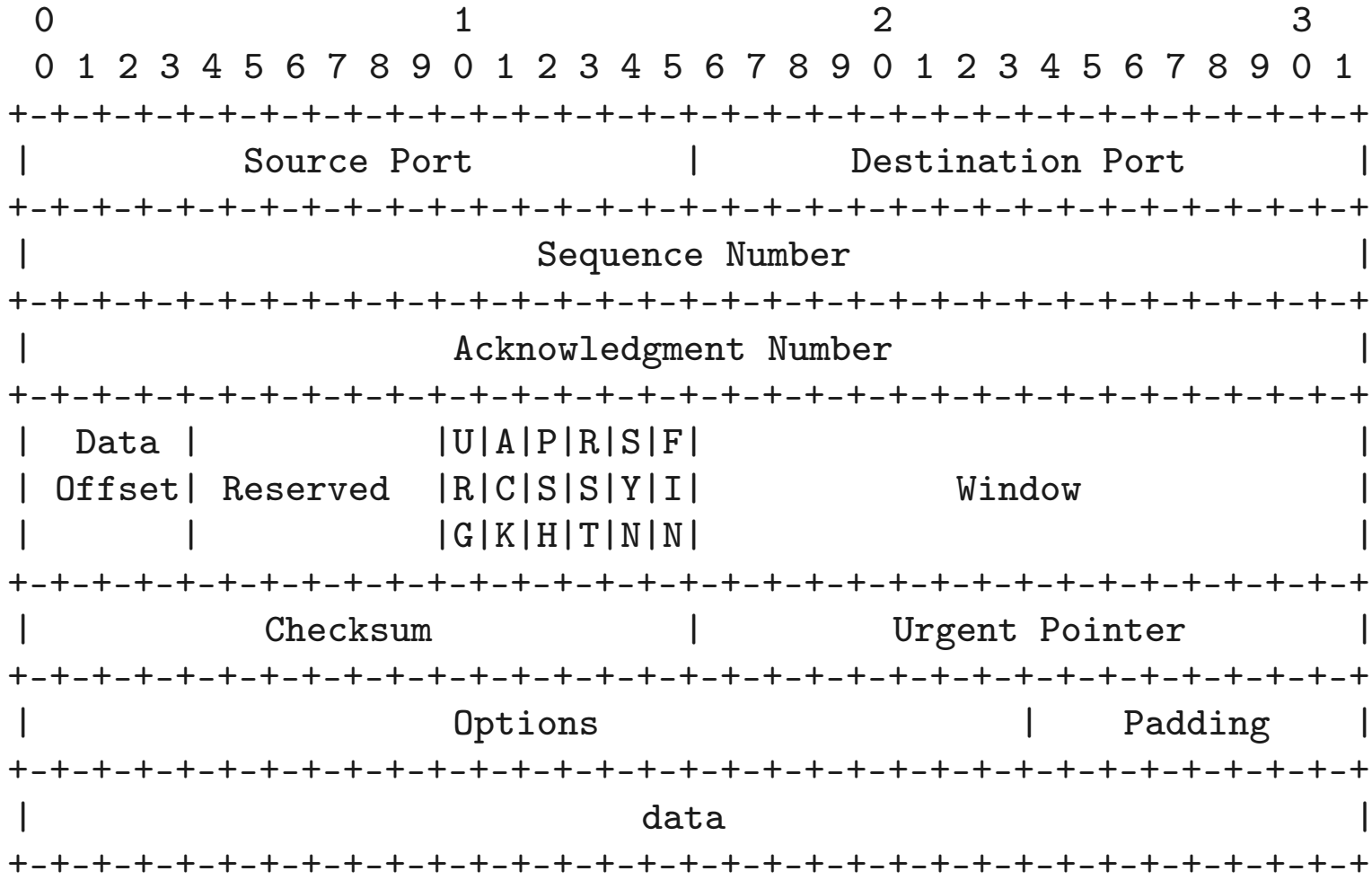
Ajout de la **fiabilité**.

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
- 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

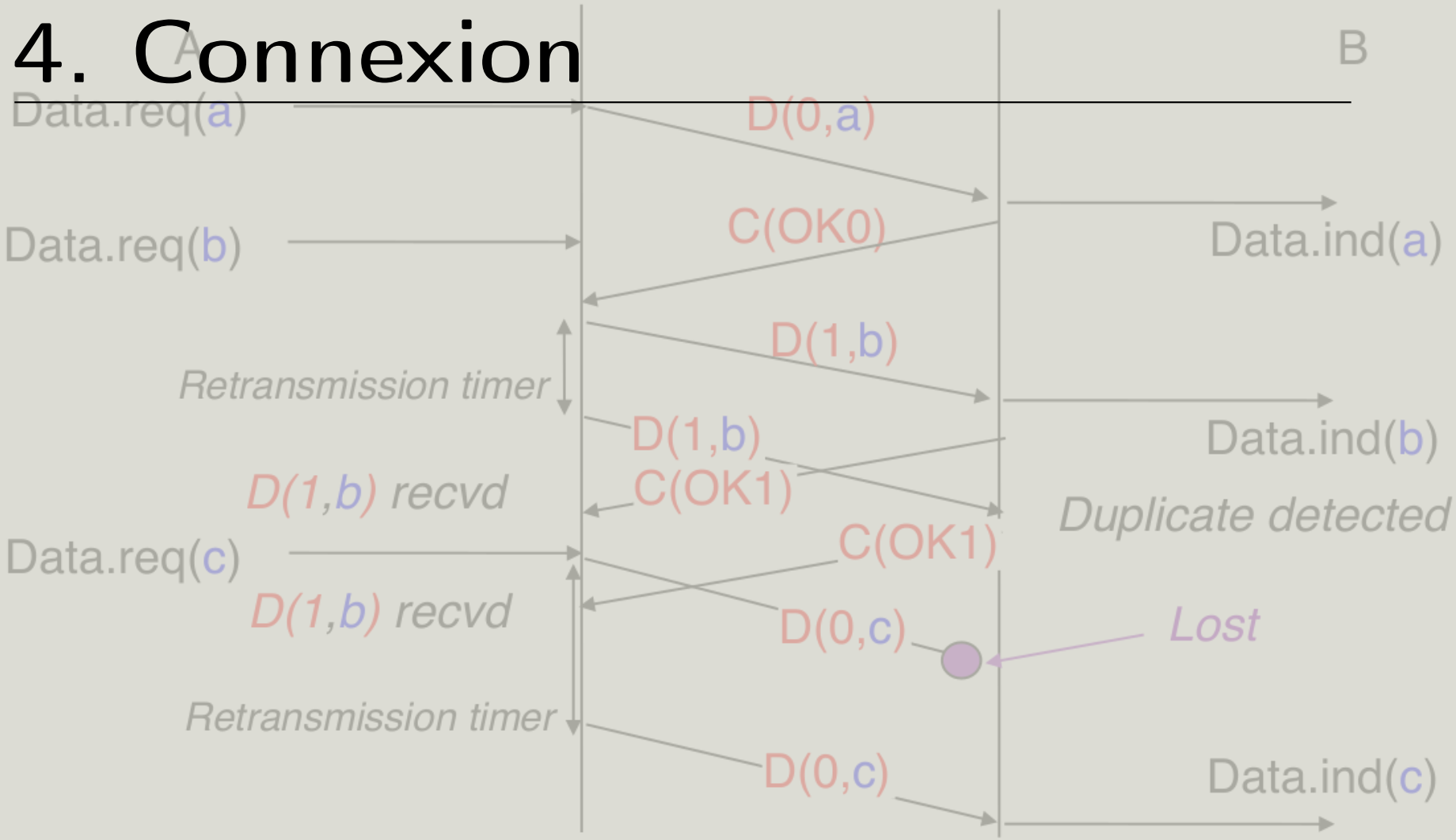
$0.8 \leq \alpha \leq 0.9$

Comment fixer la durée de l'alarme ?

$$TIMER := ERTT + 4 Deviation$$

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

4. Connexion


A
B
Data.req(a)
D(0,a)
Data.req(b)
C(OK0)
Data.ind(a)
D(1,b)
Retransmission timer
D(1,b)
Data.ind(b)
D(1,b) recvd
C(OK1)
Duplicate detected
Data.req(c)
C(OK1)
D(1,b) recvd
D(0,c)
Lost
Retransmission timer
D(0,c)
Data.ind(c)

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)

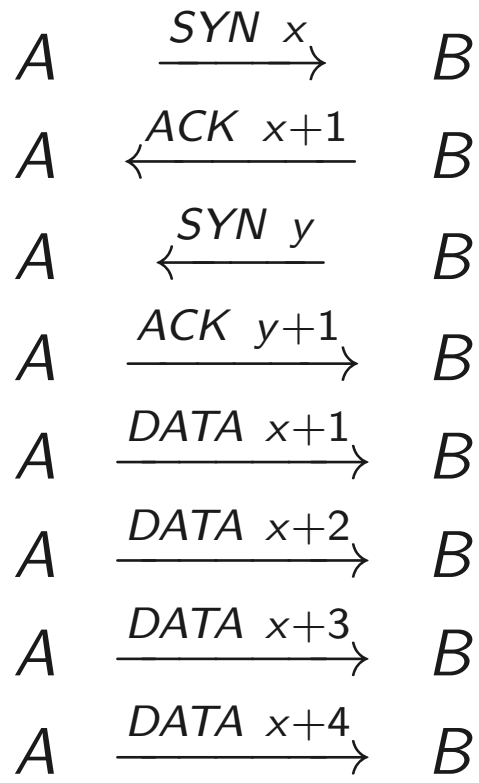
Création de la connexion

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.

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

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

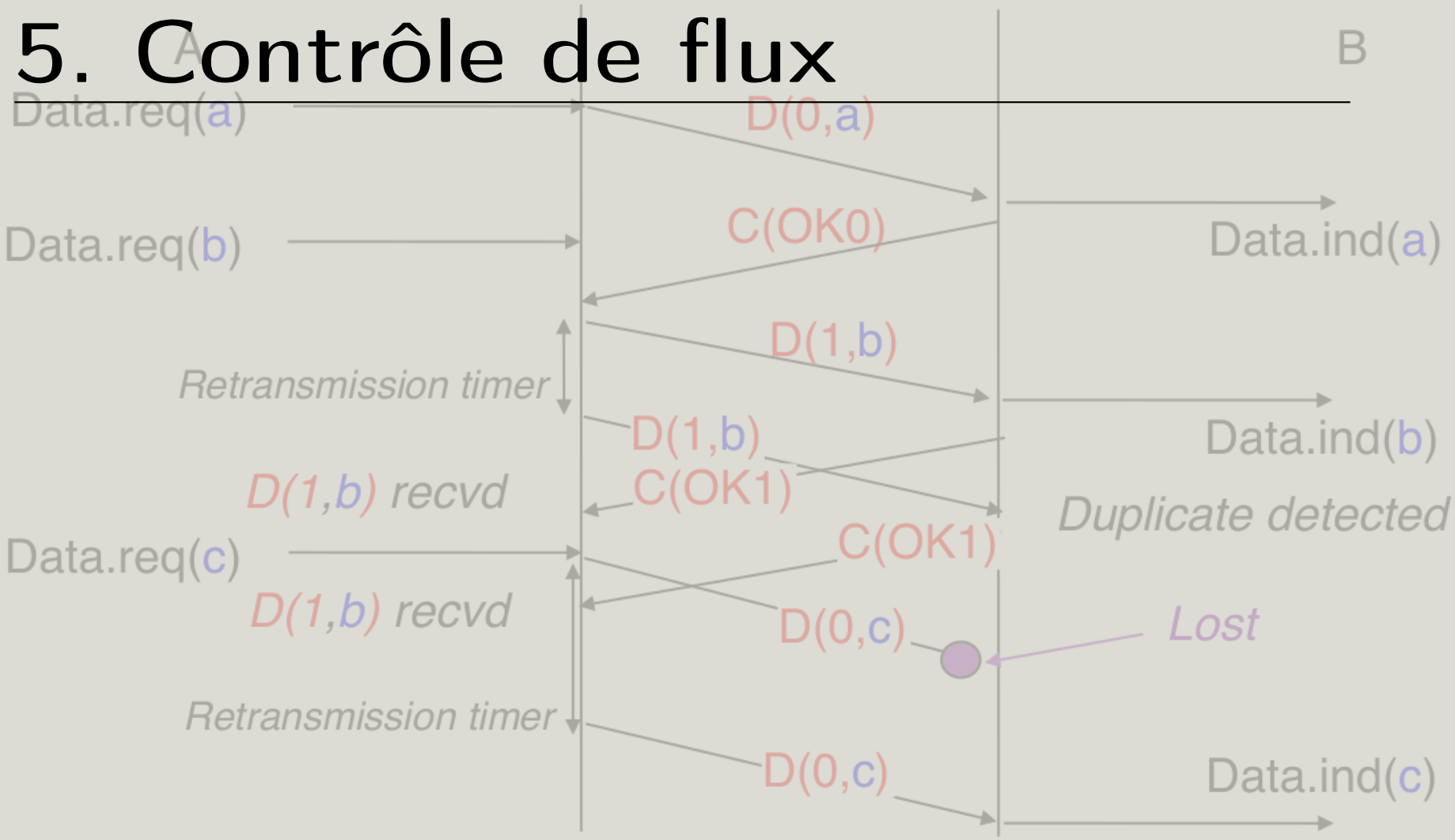
Fermeture

$$
\begin{array}{ccc}
A & \xrightarrow{FIN} & B \\
A & \xleftarrow{ACK} & B \\
A & \xleftarrow{FIN} & B \\
A & \xrightarrow{ACK} & B
\end{array}
$$

Les deux segments envoyés par B ne peuvent pas toujours être confondus

Timeout pour éviter les problèmes.

5. Contrôle de flux



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
- A doit s'arranger pour que

$$DernierBitEnvoye - DernierBitAcquitte \leq p$$

Problème

v1

- B envoie un message comme quoi $p = 0$
- Que se passe-t-il ?

Problème

v1

- B envoie un message comme quoi $p = 0$
- Que se passe-t-il ?
- Solution : A envoie des paquets de 1 octet que B acquitte.
- *Persistence* Timer

Problème

v2

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

Solution au SWS

Côté destinataire

- Attendre que p redevienne suffisament 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

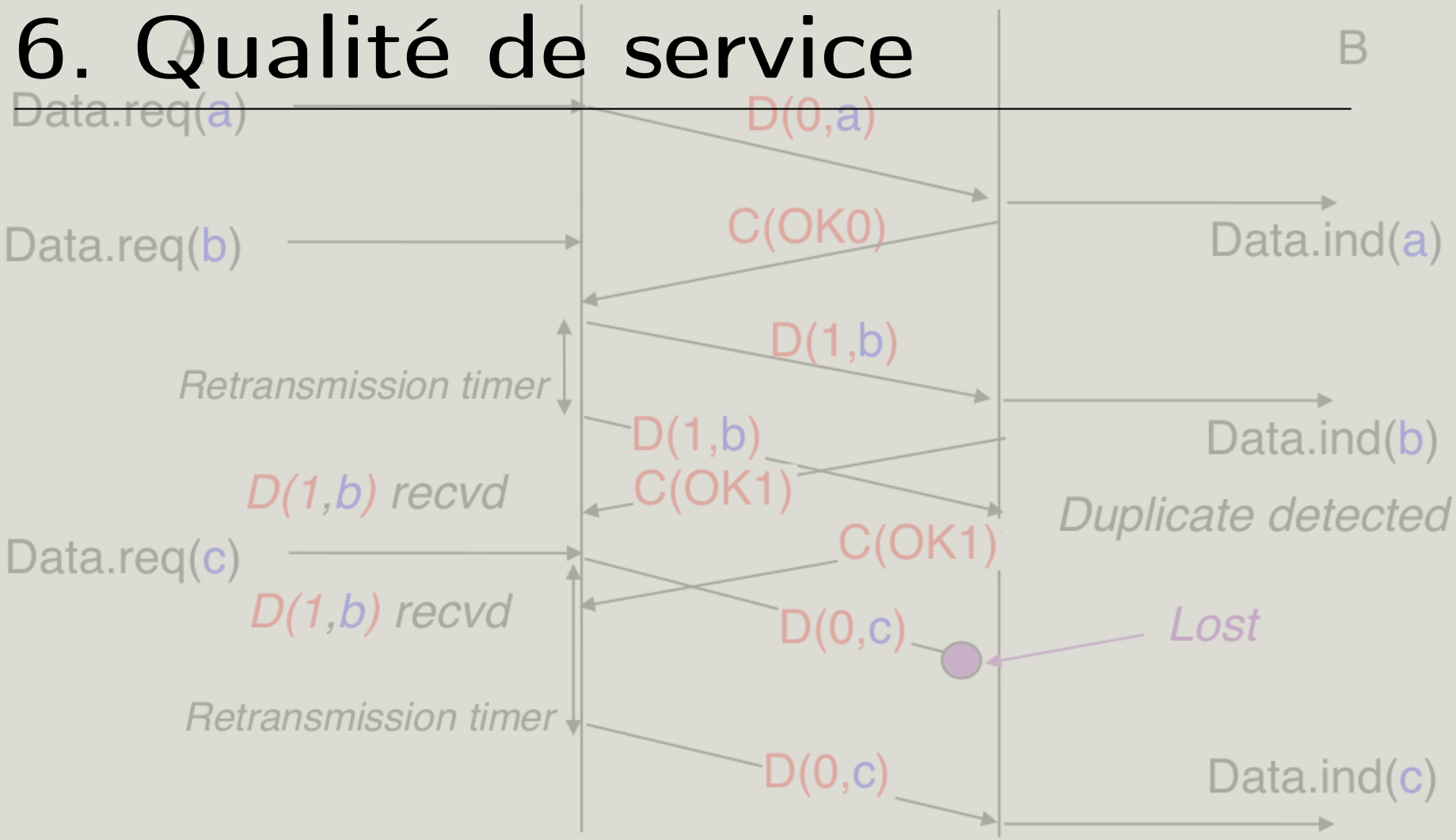
Eviter les petits paquets niveau émetteur (*Algorithme de Nagle*)

- Si beaucoup de données (plus que MSS), les envoyer
- S'il y a des données non acquittées, attendre
- Sinon envoyer

Attention

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

6. Qualité de service



Problématique

- Contrôler la congestion
- 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...

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 λ de congestion.
- Si $F < \lambda$
  - Au début, taille de la fenêtre de 2 MSS (*slow start*)
  - A chaque ACK, on augmente de 1 MSS
  - 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

Mes remerciements à Nicolas Ollinger (Université d'orléans).