COSC264
Introduction to Computer Networks and the Internet

Reliable data transfer: ARQ protocols
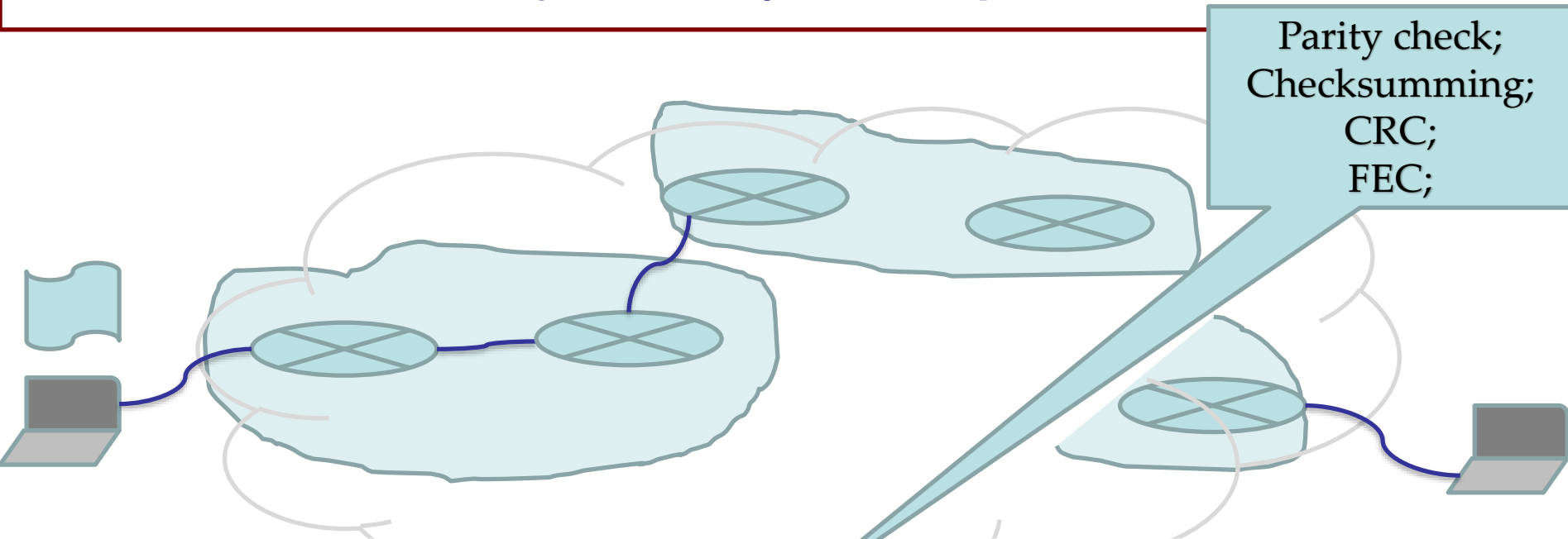
Dr. Barry Wu

Wireless Research Centre

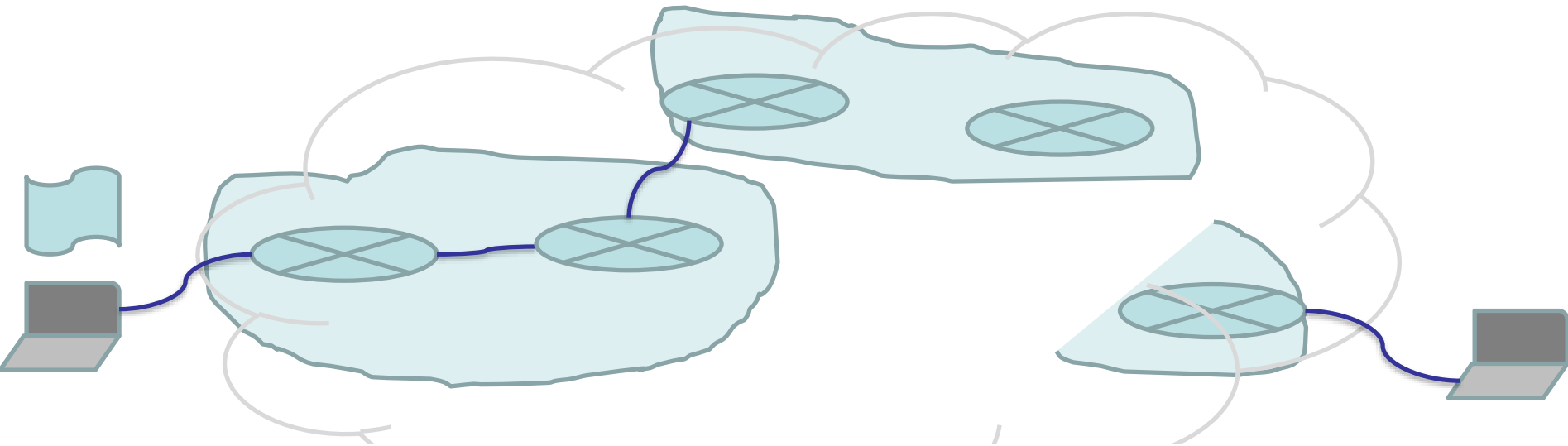University of Canterbury

barry.wu@canterbury.ac.nz

The journey of a packet

Parity check;
Checksumming;
CRC;
FEC;

Problems	Causes	Solutions
Bit error	e.g., signal attenuation/noise	Error detection and correction
Buffer overflow	e.g., Speed-mismatch; Too much traffic;	Flow control and congestion control
Lost packet	e.g., buffer overflow at host/router	Acknowledgement and retransmission (ARQ)
Out of order	e.g. an early packet gets lost and retransmitted; a later one arrives first.	Acknowledgement and retransmission (ARQ)

The journey of a packet

Problems	Causes	Solutions
Bit error	e.g., signal attenuation/noise	Error detection and correction
Buffer overflow	e.g., Speed-mismatch; Too much traffic;	Flow control and congestion control
Lost packet	e.g., buffer overflow at host/router	Acknowledgement and retransmission (ARQ)
Out of order	e.g. an early packet gets lost and retransmitted; a later one arrives first.	Acknowledgement and retransmission (ARQ)

Outline

- ARQ (Automatic Repeat reQuest) Protocols
  - Alternating Bit Protocol
  - Go-Back-N
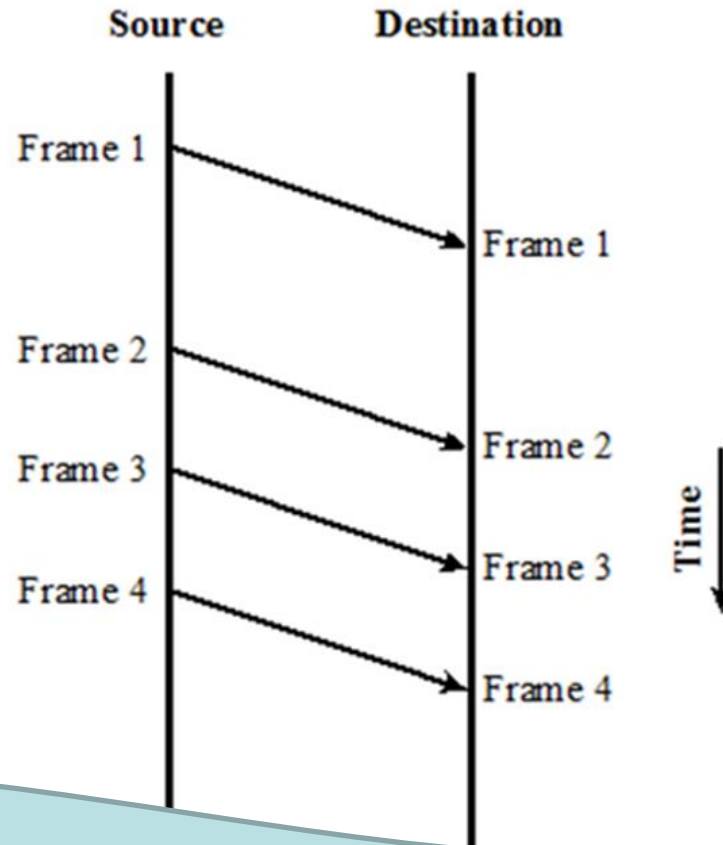 - Selective Repeat (Selective Reject)
- Summary

In a computer network setting, reliable data transfer protocols based on retransmission are known as ARQ (Automatic Repeat reQuest) protocols. [KR3]

Outline

- Reliable transfer protocols
  - rdt1.0: for a reliable channel
  - rdt2.0: for a channel with bit errors
  - rdt2.1: sender, handles garbled ACK/NAKs
  - rdt2.2: a NAK-free protocol
  - rdt3.0: channels with errors *and* loss
  - Pipelined protocols
    - Go-back-N
    - Selective repeat (Selective reject)

- A perfectly reliable channel;

  - no bit errors
  - no buffer overflow
  - no out-of-order
  - no packet loss

Source   Destination

Frame 1

Frame 1

Frame 2

Frame 2

Frame 3

Time

Frame 3

Frame 4

Frame 4

The sender sends data whenever data is available; the receiver receives data and delivers data to its upper-layer protocol;

6

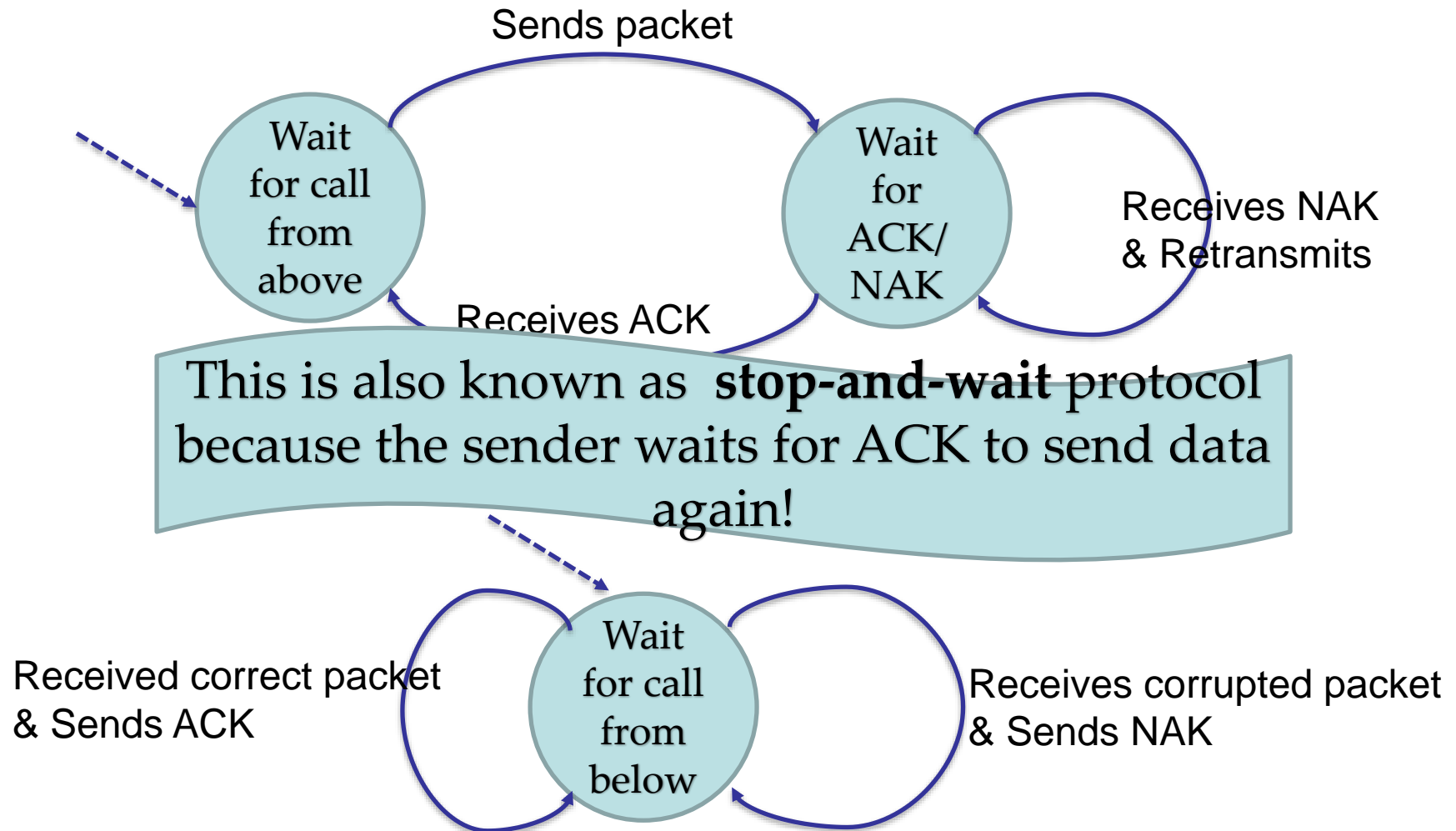- A channel with only bit errors;
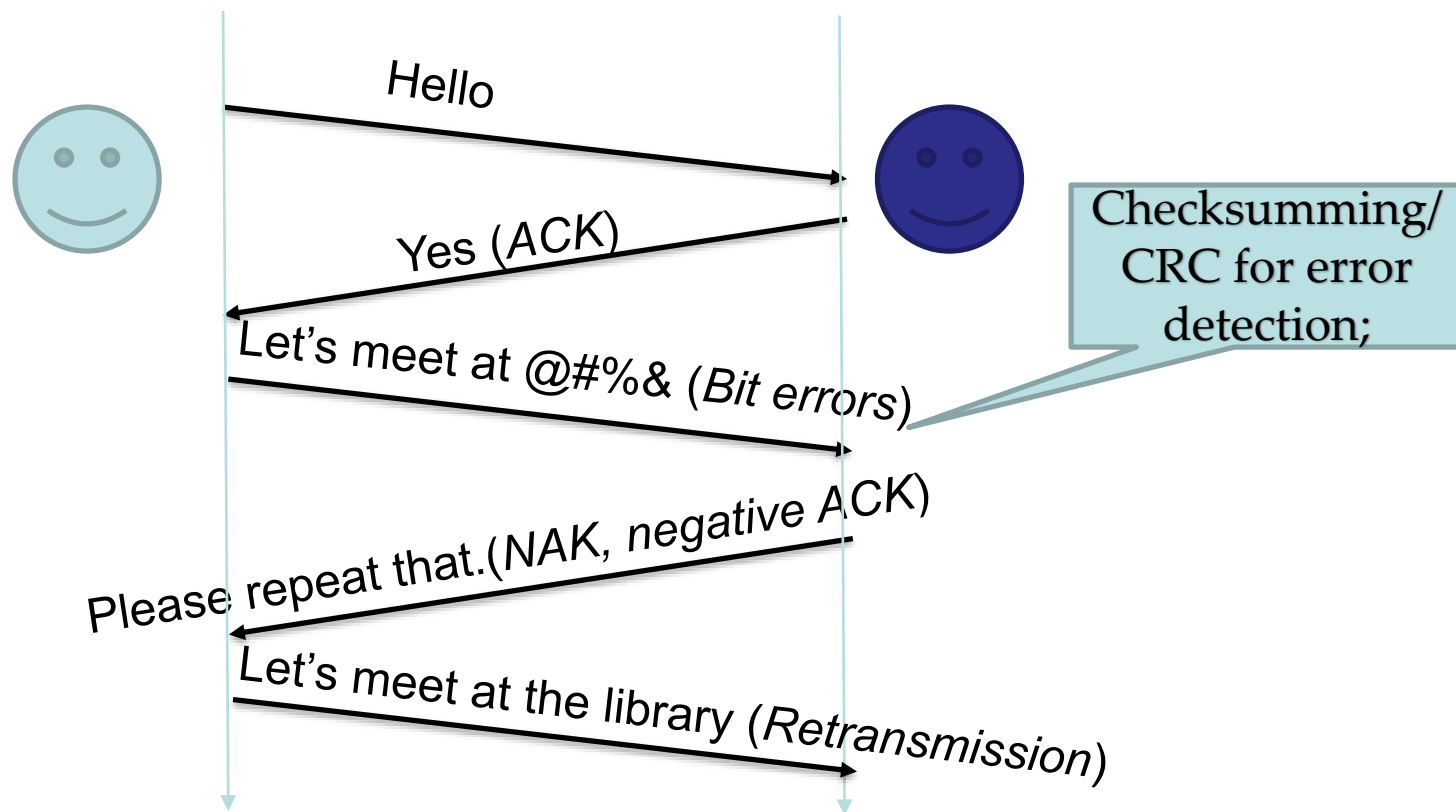
rdt 2.0 protocol

- Sender
  - When packet is ready, send it and wait for ACK/NAK;
  - If NAK is received, retransmits the last packet and wait for ACK/NAK;
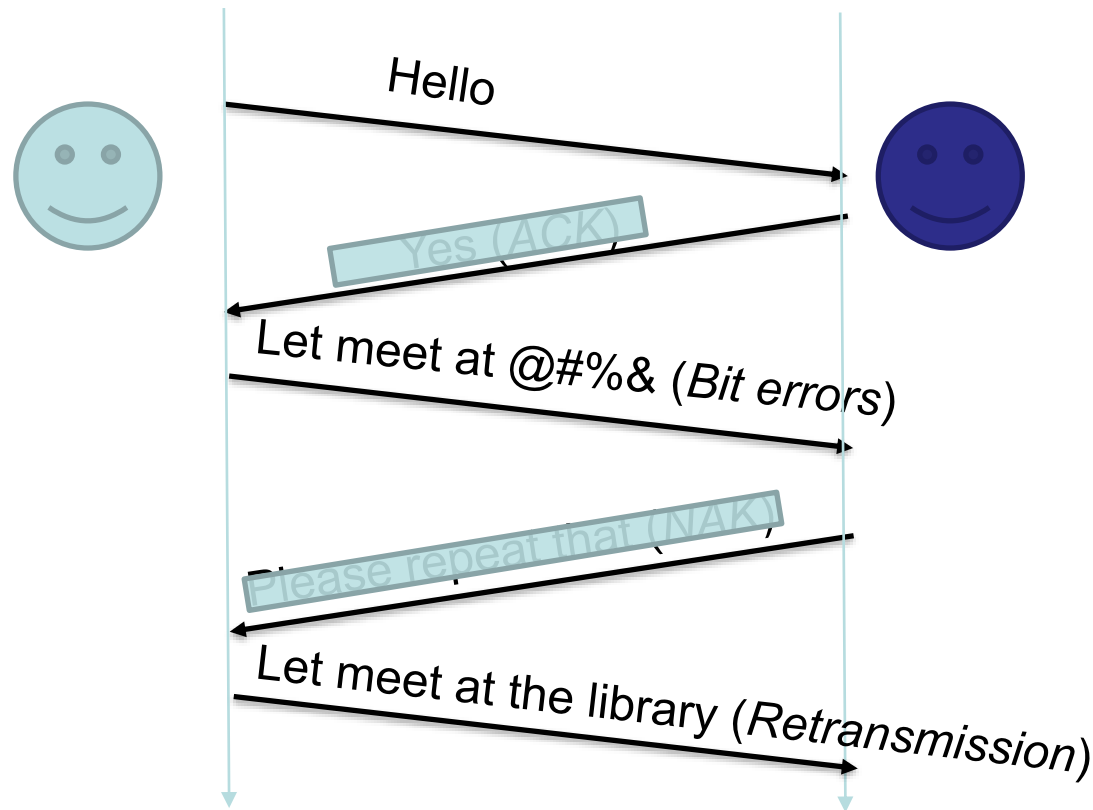 - If ACK is received, wait for next packet to be ready;
- Receiver
  - If packet is received without error, sends ACK;
  - If packet is received with error, sends NAK;
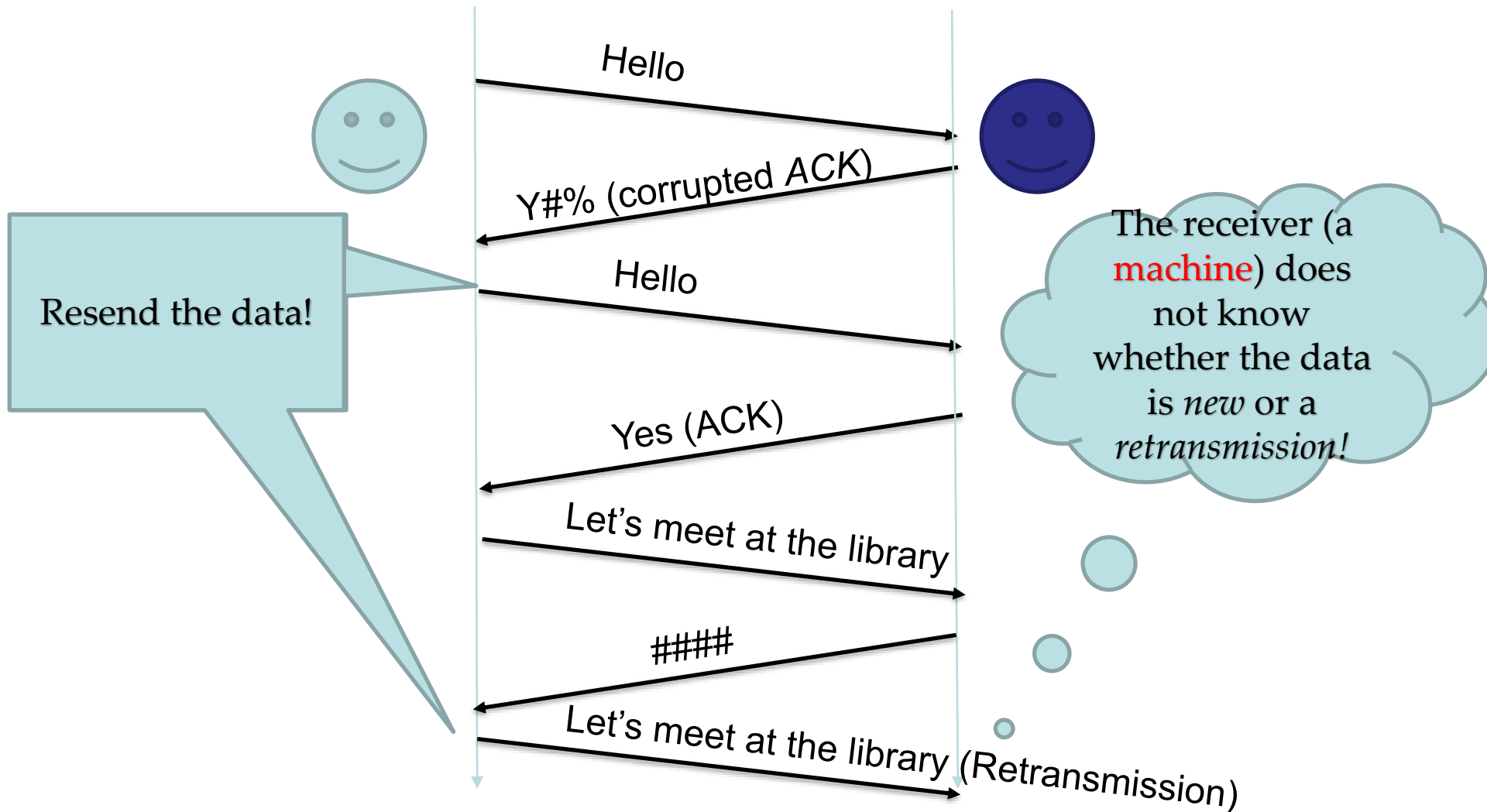
rdt 2.0: FSM (Finite-state machine)

Sends packet

Wait for call from above

Wait for ACK/ NAK

Receives NAK & Retransmits

Receives ACK

This is also known as **stop-and-wait** protocol because the sender waits for ACK to send data again!

Wait for call from below

Received correct packet & Sends ACK

Receives corrupted packet & Sends NAK

Hello

Yes (ACK)

Let meet at @#%& (Bit errors)

Please repeat that (NAK)

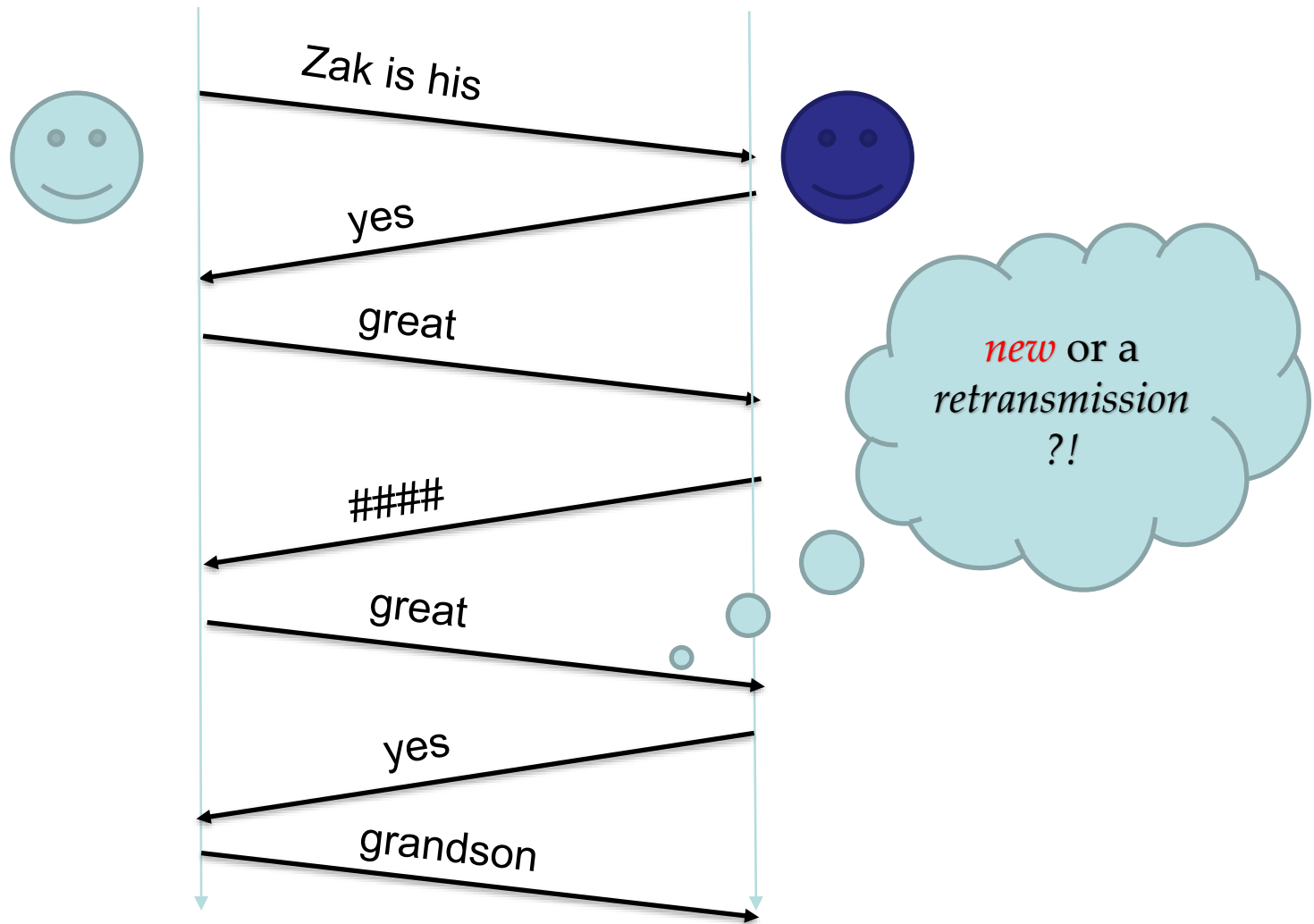Let meet at the library (Retransmission)

Again we can detect bit errors in ACK/NAK!
But we only know something is corrupted but do not know it is an ACK or NAK exactly;

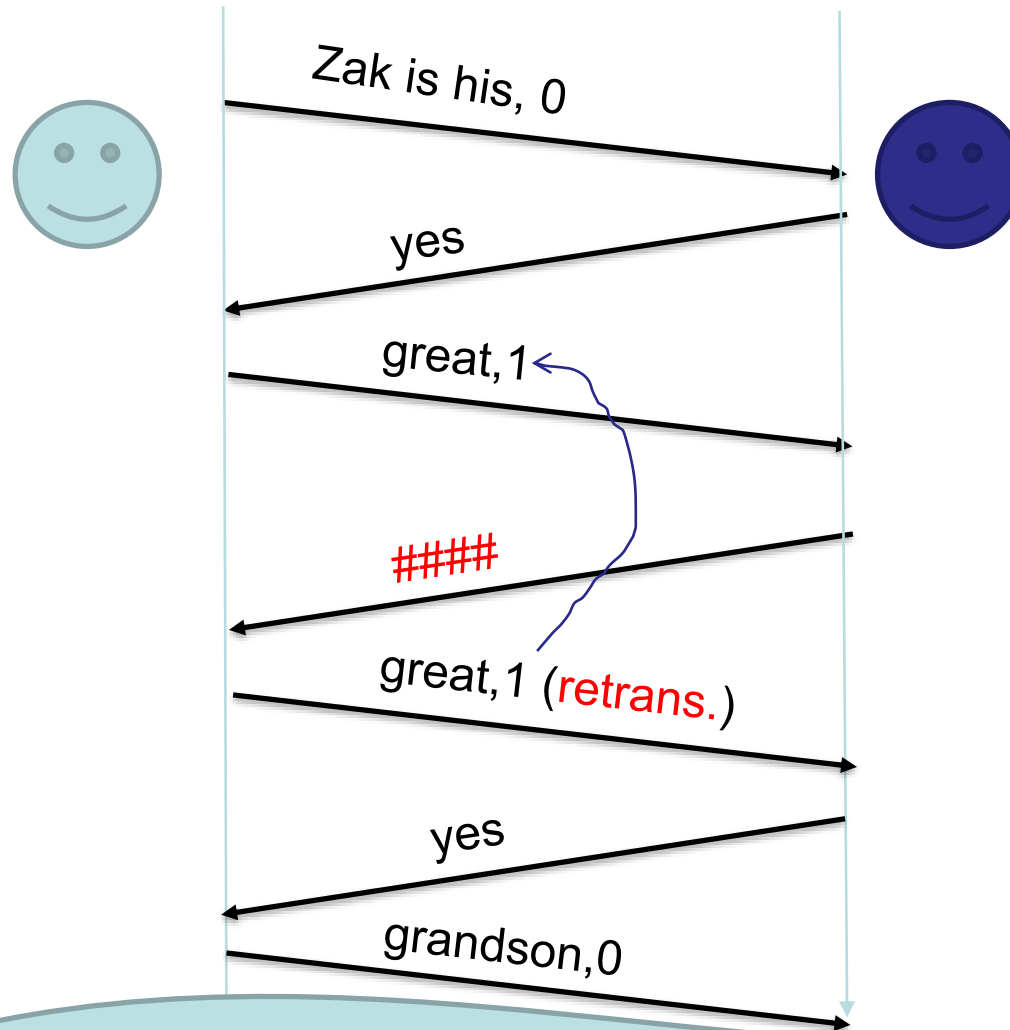# rdt 2.0: A fatal flaw-corrupted ACK/NAK

Hello

Y#% (corrupted ACK)

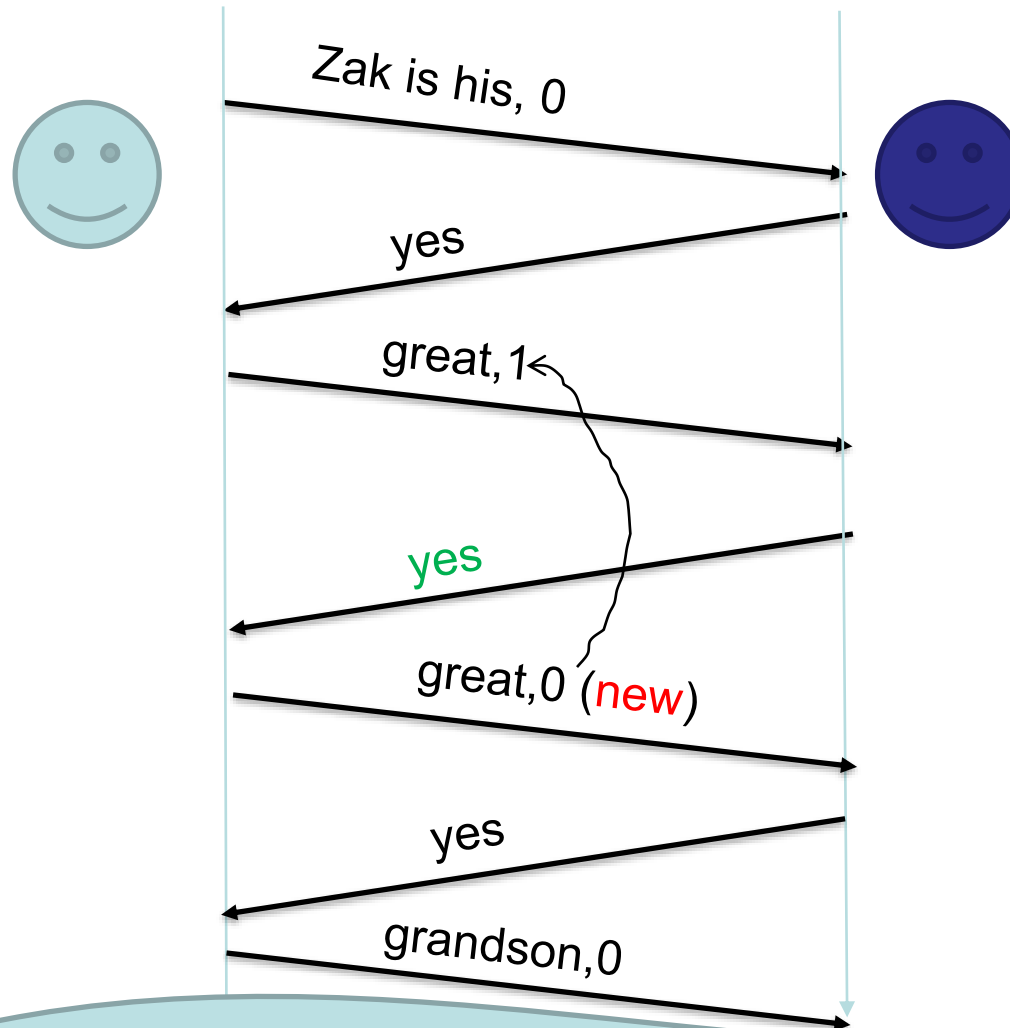Resend the data!

Hello

Yes (ACK)

Let's meet at the library

####

Let's meet at the library (Retransmission)

The receiver (a machine) does not know whether the data is *new* or a *retransmission!*

Zak is his, 0

yes

great,1

####

great,1 (retrans.)

yes

grandson,0

Zak is his great grandson; the receiver will not be confused!

rdt 2.0: A fatal flaw-corrupted ACK/NACK

Zak is his, 0

yes

great,1

yes

great,0 (new)

yes

grandson,0

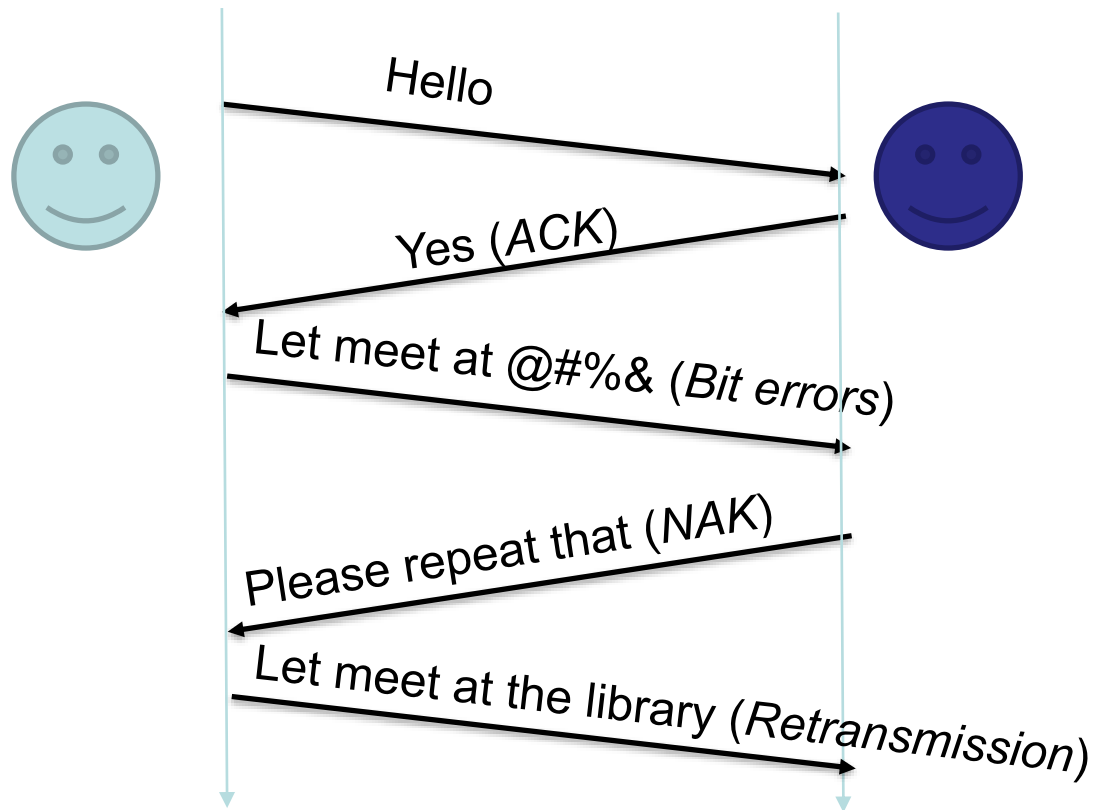Zak is his great great grandson; the receiver will not be confused!

rdt 2.1: A fatal flaw: corrupted ACK/NACK

- Data packet with a new field (1 bit sequence number);
  - retransmission (same as previous)
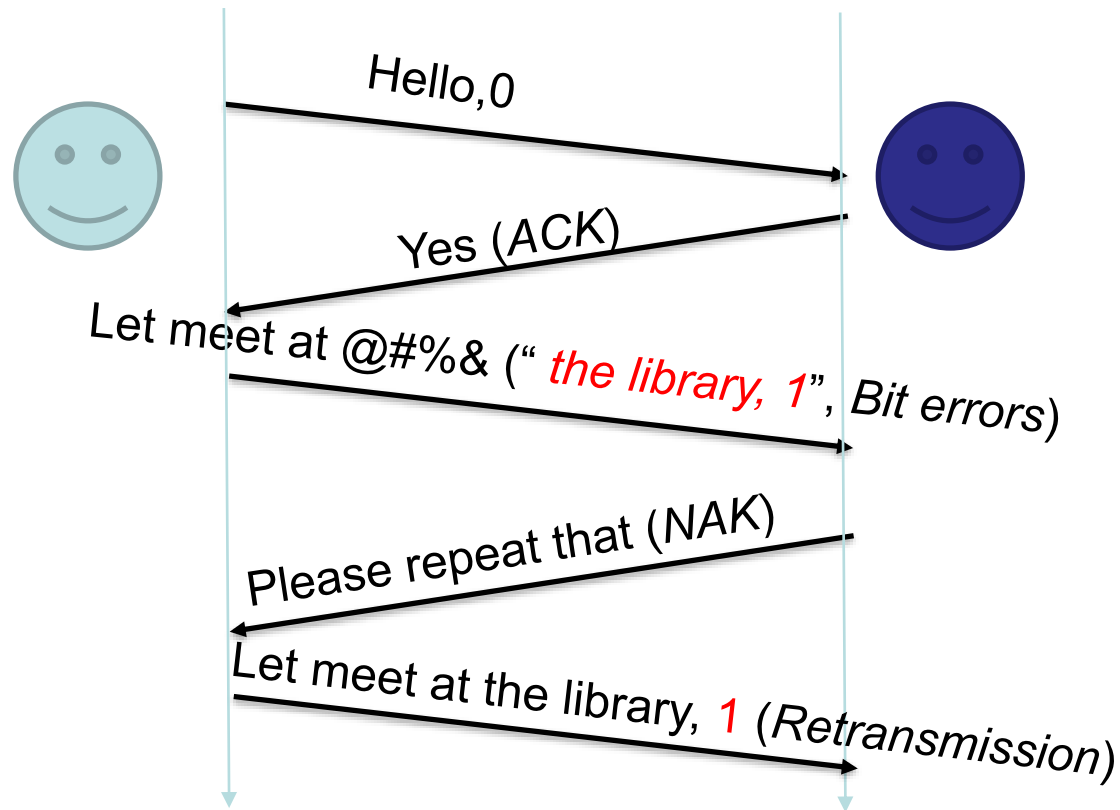 - new(different from previous)

Two cases: corrupted data (ACK/NAK); corrupted acknowledgements (1 bit seq #).
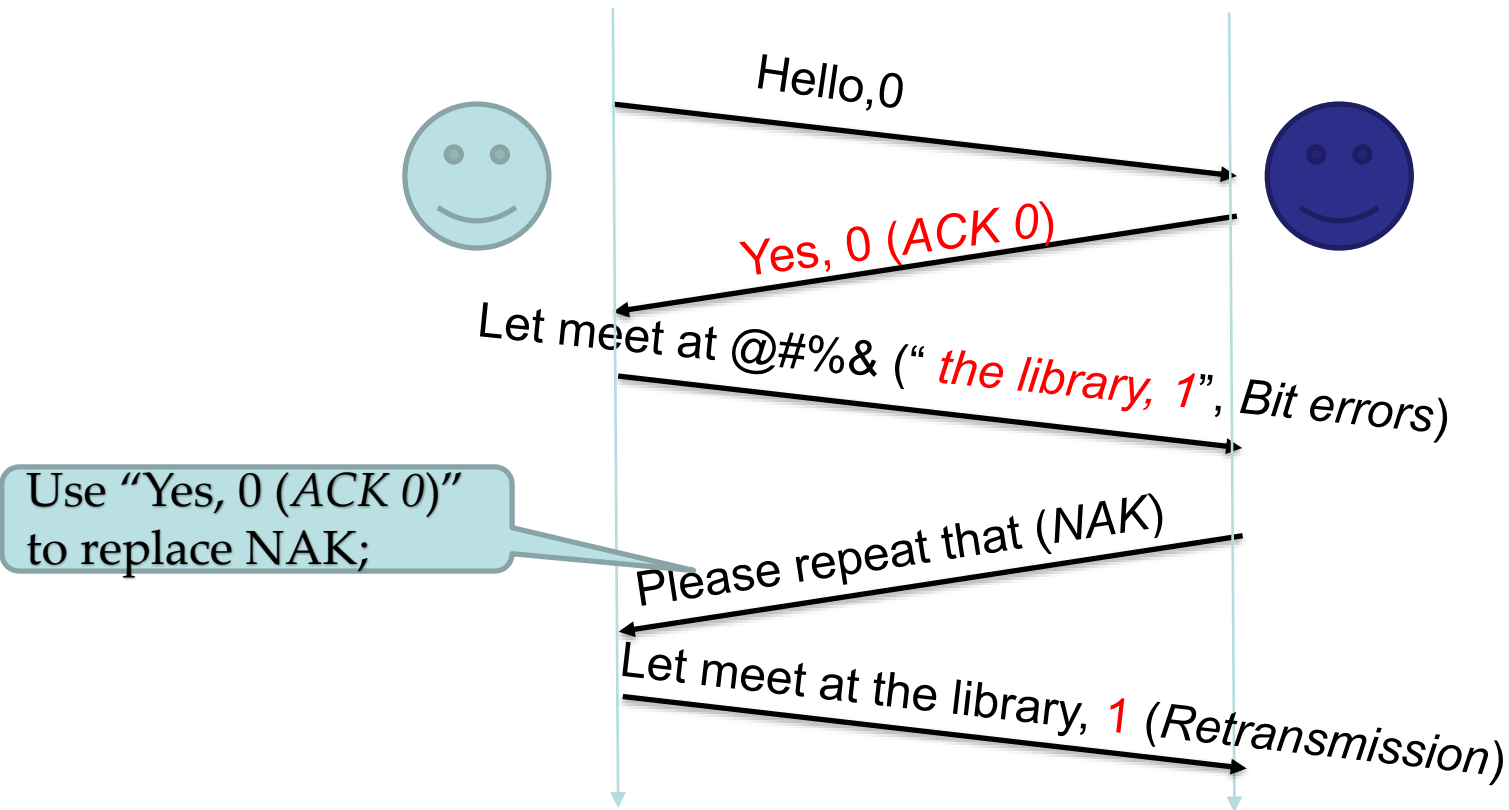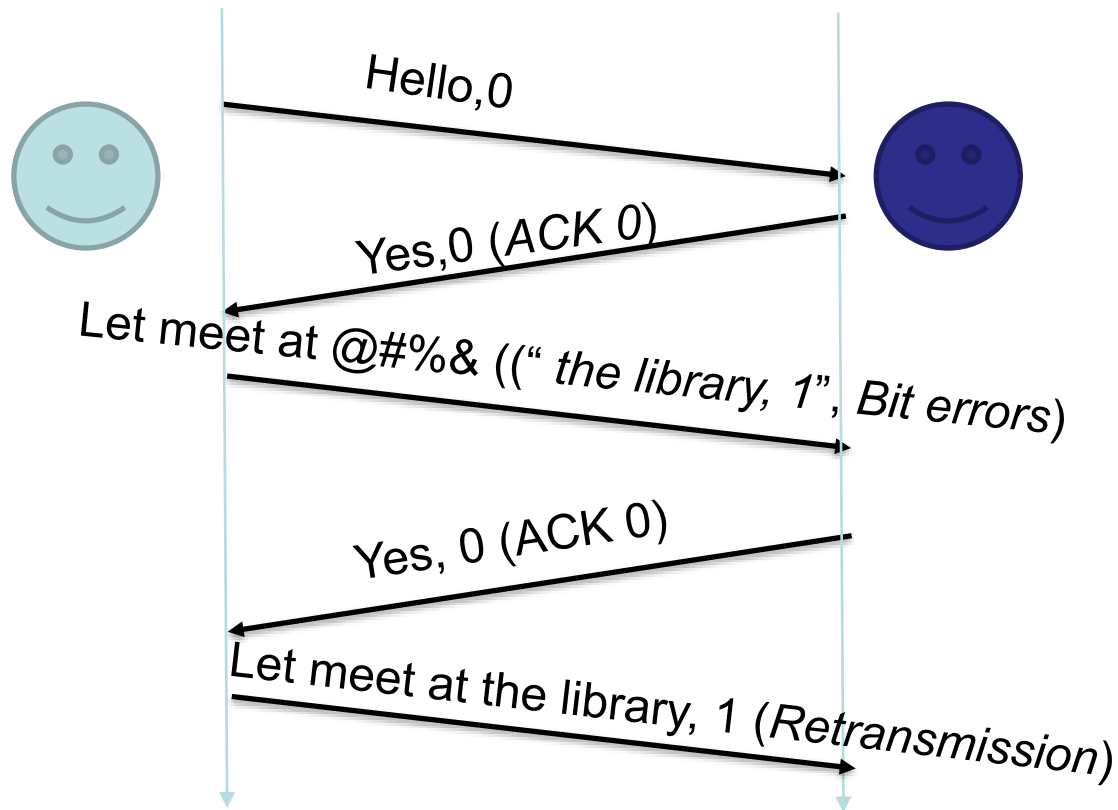Assumption: no packet loss;

rdt 2.2: a NAK-free protocol

Hello

Yes (ACK)

Let meet at @#%& (Bit errors)

Please repeat that (NAK)

Let meet at the library (Retransmission)

rdt 2.2: a NAK-free protocol

Hello,0

Yes (ACK)

Let meet at @#%& (" *the library, 1*", Bit errors)

Please repeat that (NAK)

Let meet at the library, 1 (Retransmission)

rdt 2.2: a NAK-free protocol

rdt 2.2: A NAK-free protocol

Hello,0

Yes,0 (ACK 0)

Let meet at @#%& ((" the library, 1", Bit errors)

Yes, 0 (ACK 0)

Let meet at the library, 1 (Retransmission)

We can use duplicate ACKs to replace NAK.

Two cases: corrupted data (ACK/NAK); corrupted acknowledgements (1 bit seq # in data).
NAK-free: use duplicate ACKs;
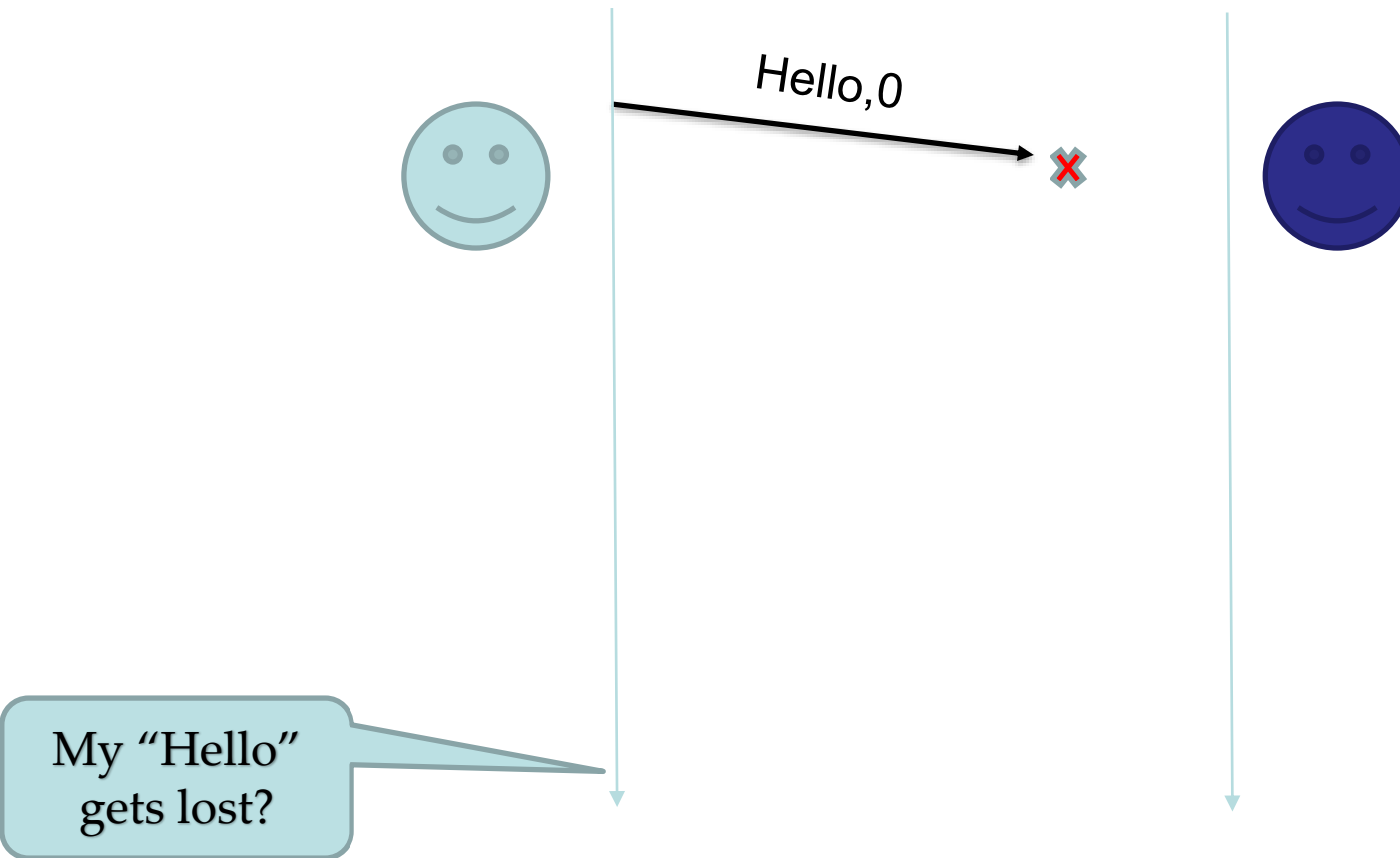*Assumption: no packet loss;*

rdt 3.0: Reliable data transfer over a *lossy* channel with bit errors

- Packets can get lost.
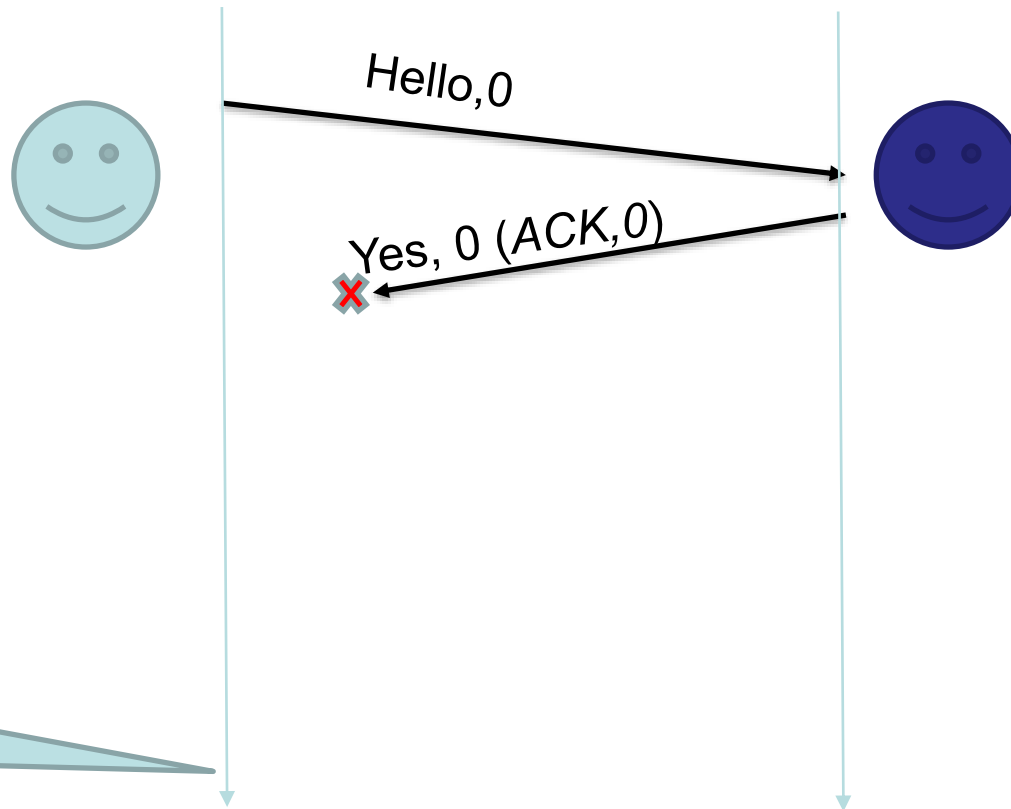 - How to detect packet loss?
  - What to do when packet loss occurs?

What to do when packet loss occurs

- Retransmission (A panacea!)
  - Data packet loss
  - ACK loss
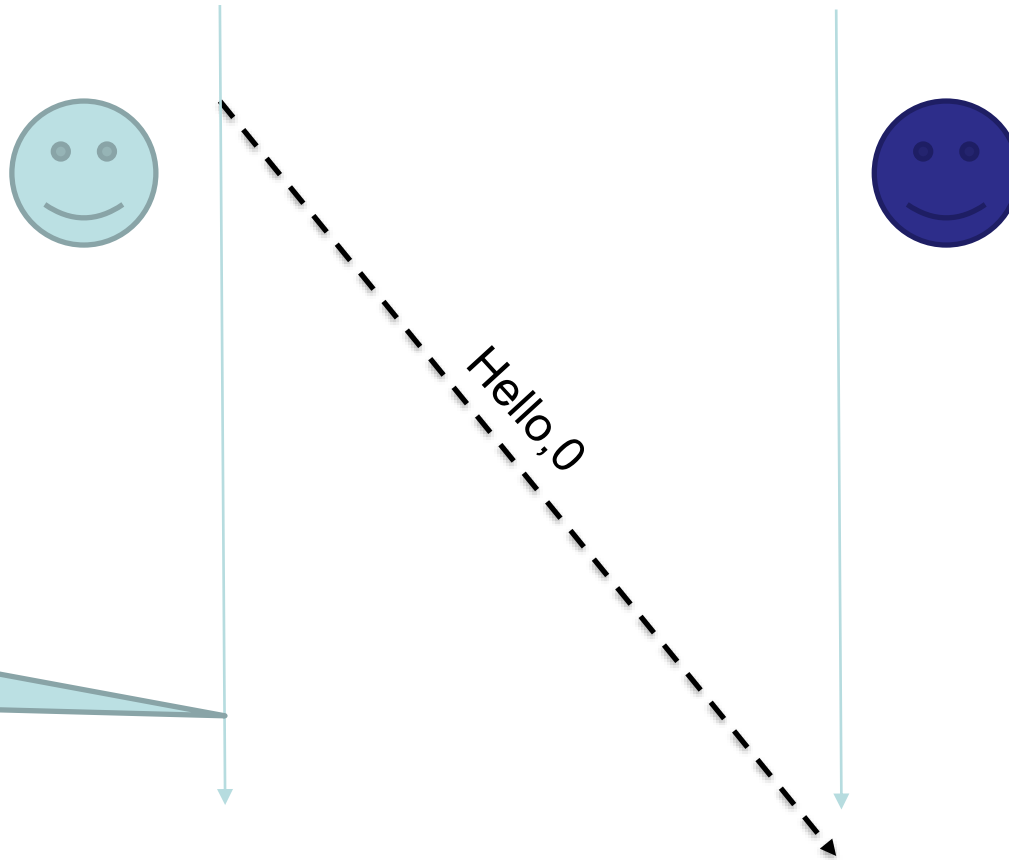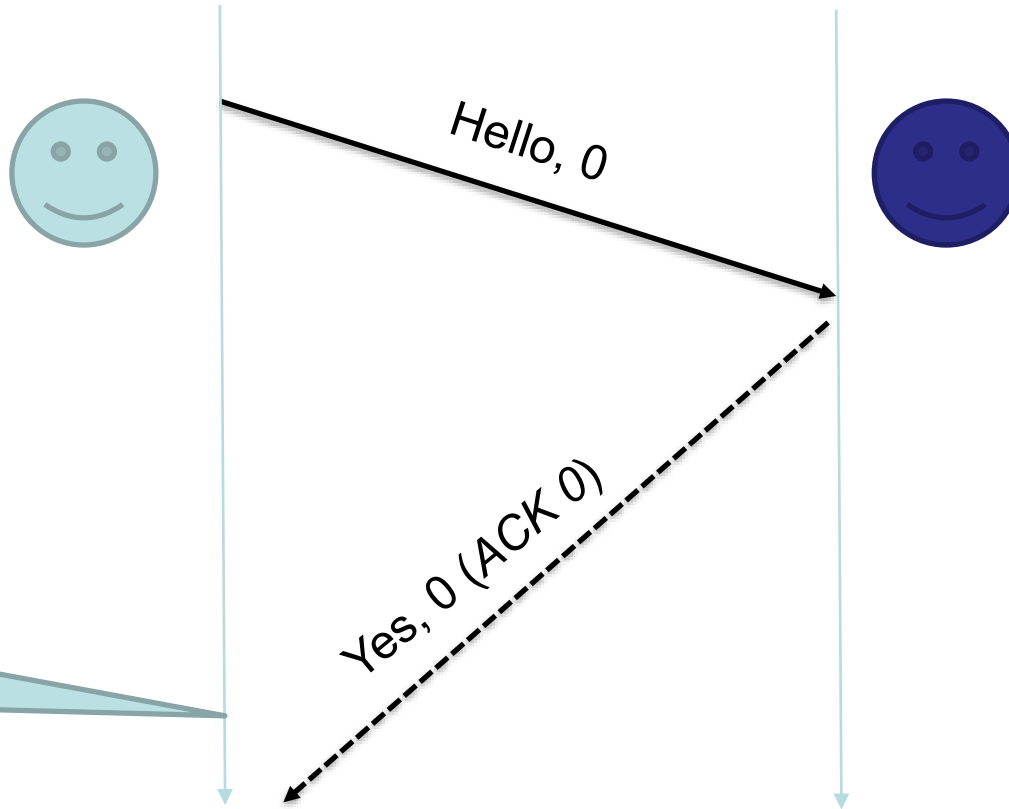 - Data/ACK delayed

Data loss

ACK loss

Delayed

Delayed
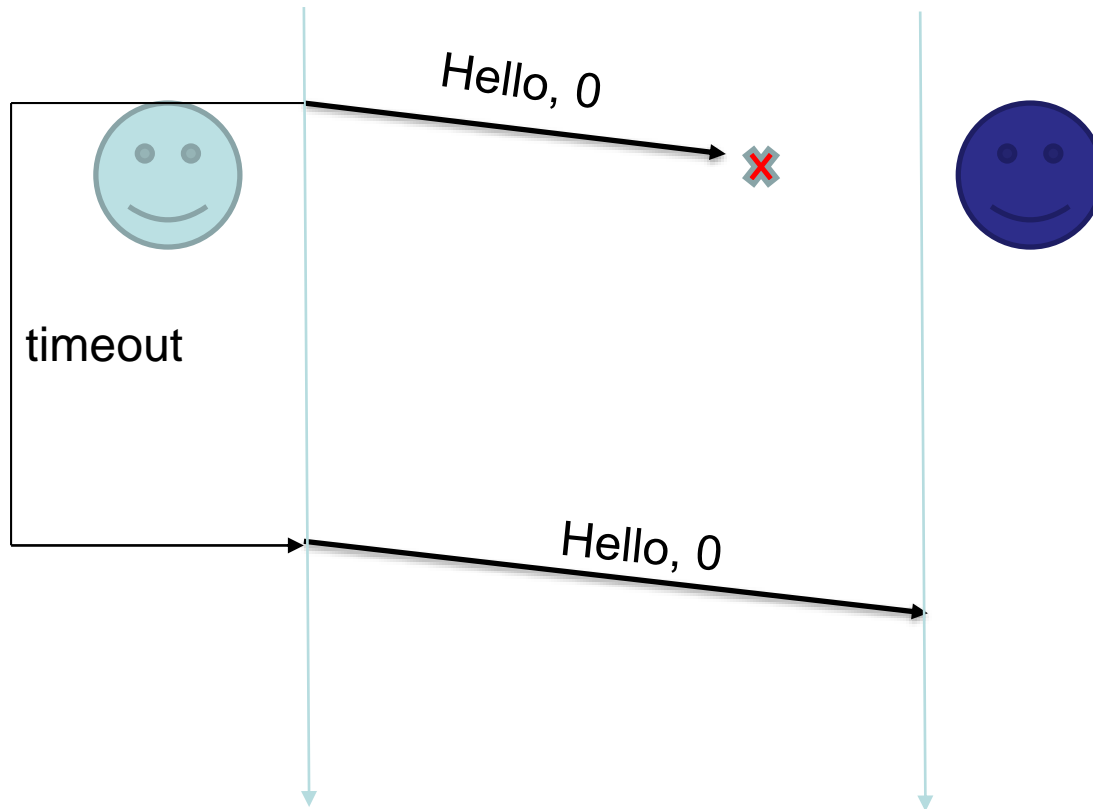
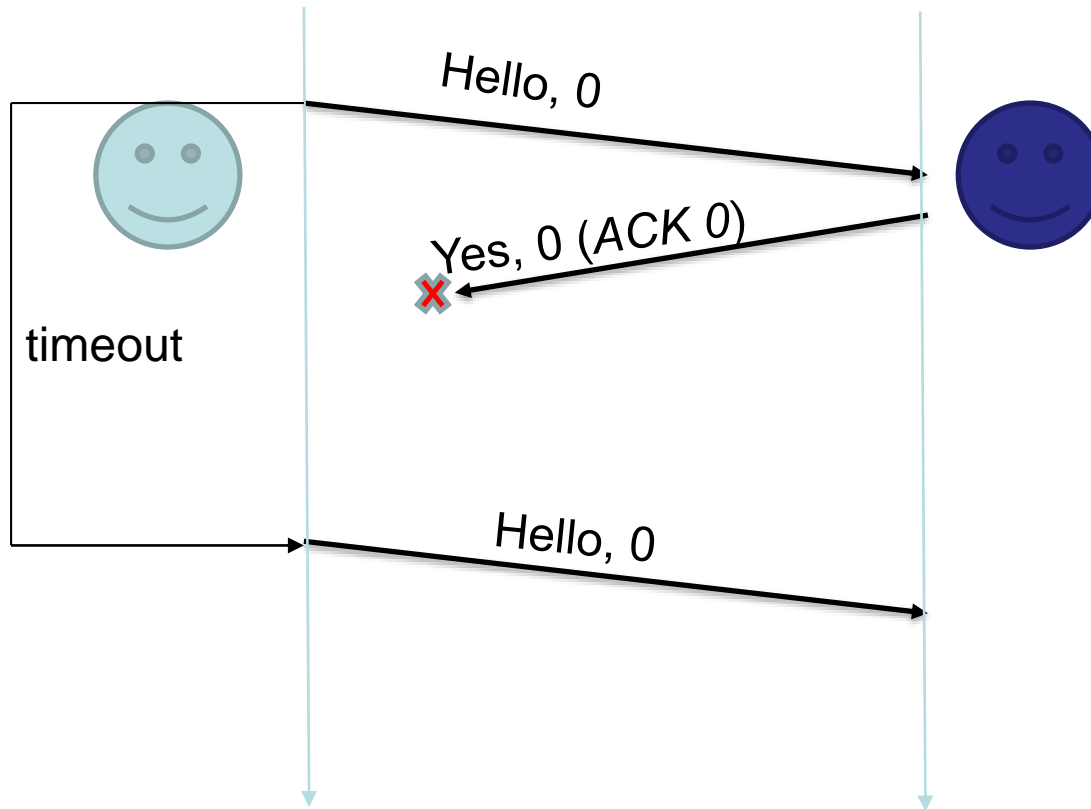# How to detect packet loss

- Wait, wait and wait!
- How long should the sender wait?
  - Too short → duplicate data packets
  - Too long → decreased performance
  - The solution: to judiciously choose a time value

We will use a countdown timer!

Data loss

Hello, 0

timeout

Hello, 0

Hello, 0

Yes, 0 (ACK 0)

timeout

Hello, 0

Delayed

Delayed

timeout

Hello, 0

Yes, 0 (ACK 0)

Hello, 0
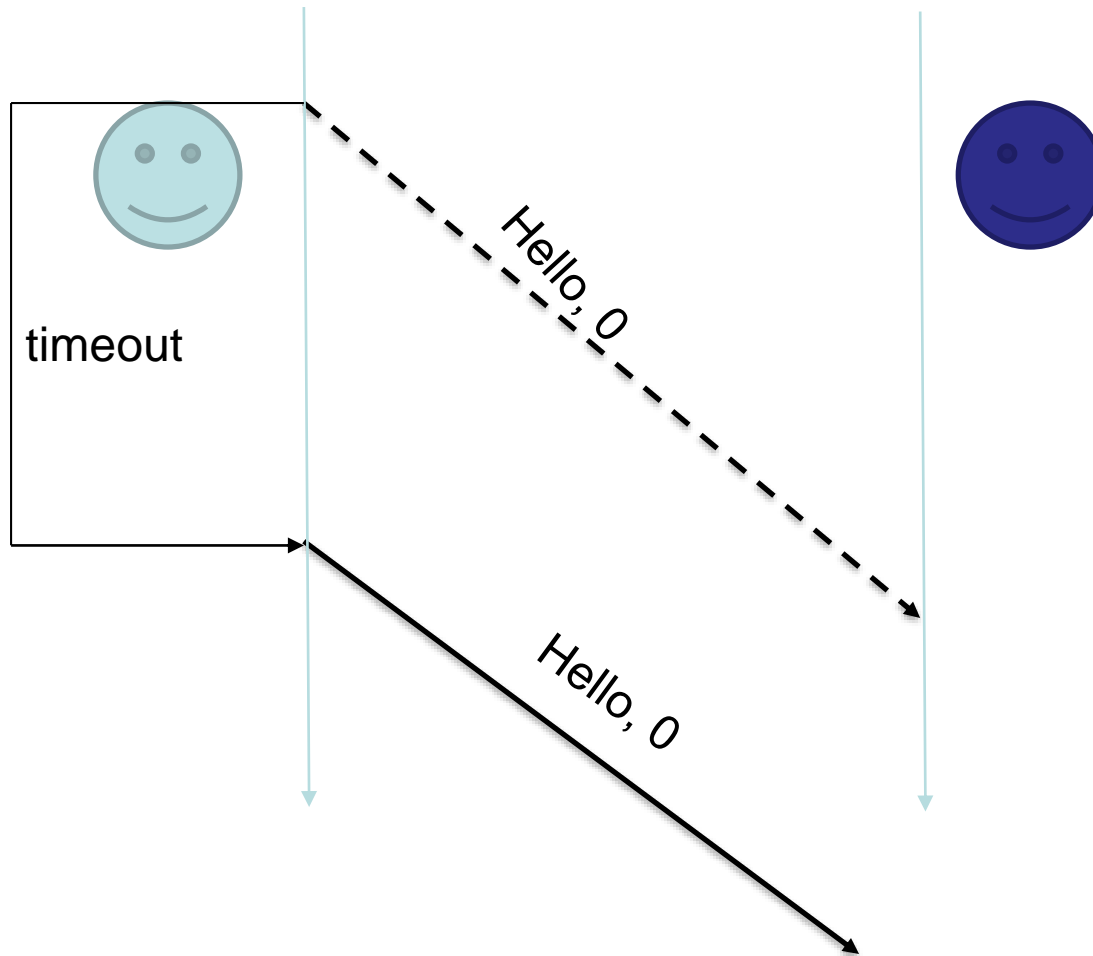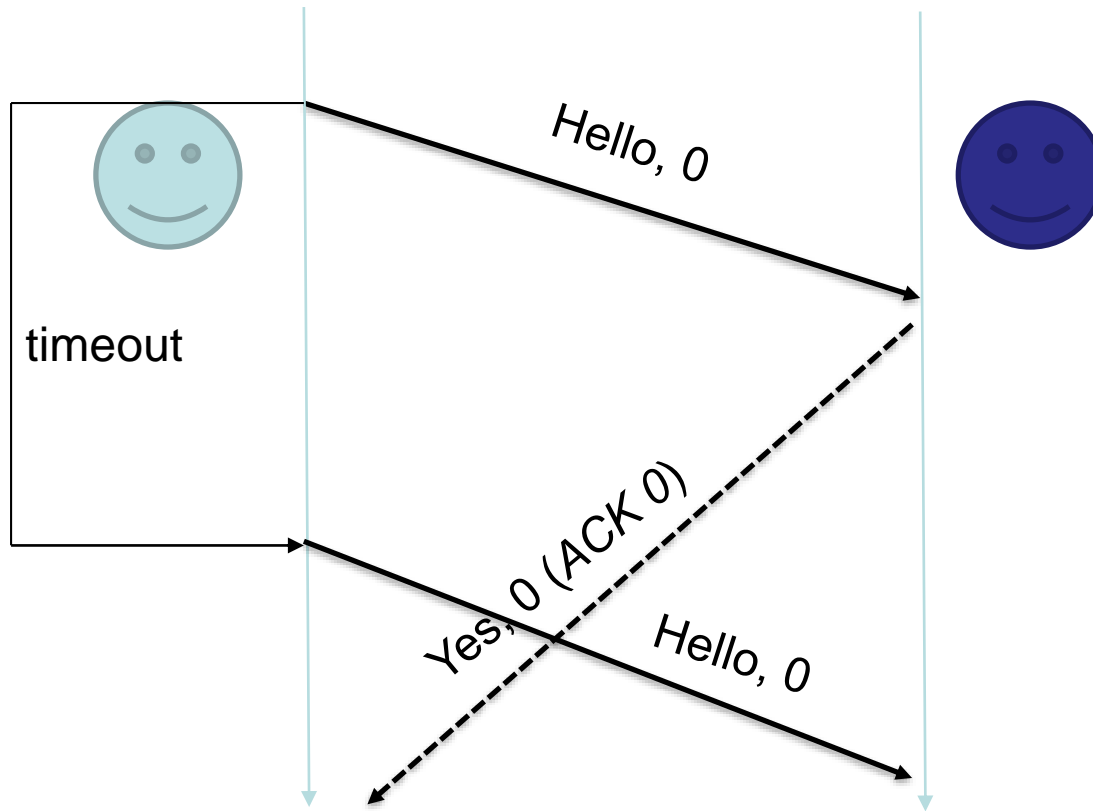
Use timeout to detect packet loss (no guarantee);
Use retransmission to recover from packet loss (duplicate data packets);

rdt 3.0: An illustration

(a) operation with no loss

(b) lost packet

Also known as alternating-bit protocol

rdt 3.0: An illustration

(c) lost ACK

(d) Premature timeout
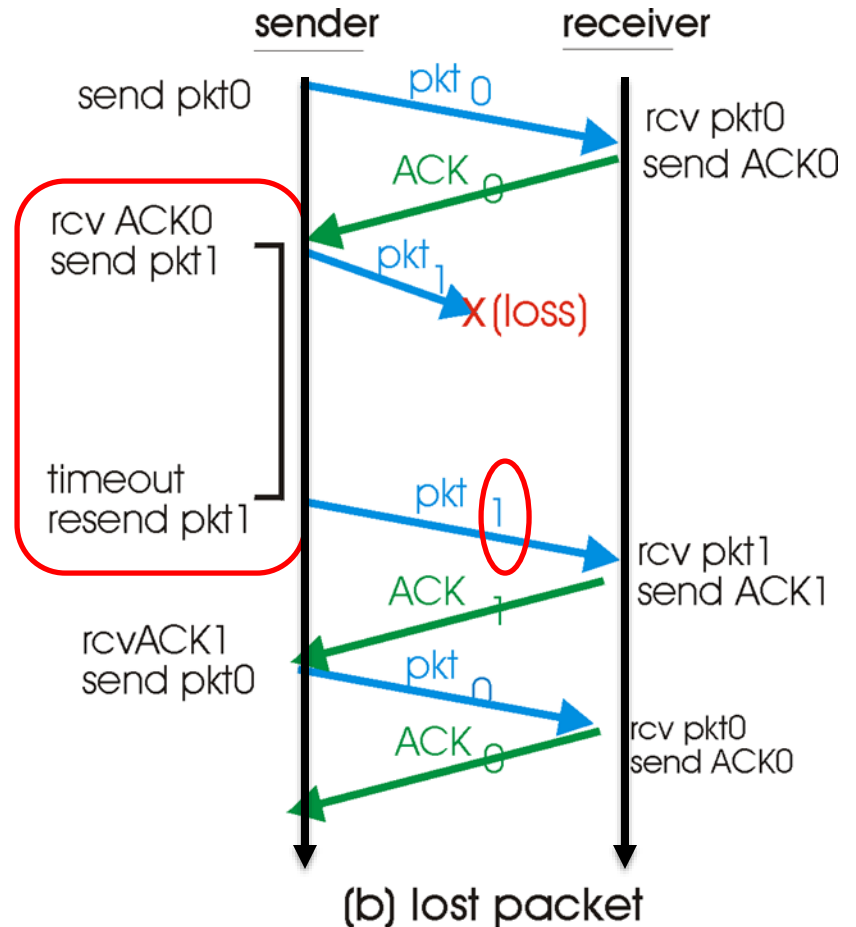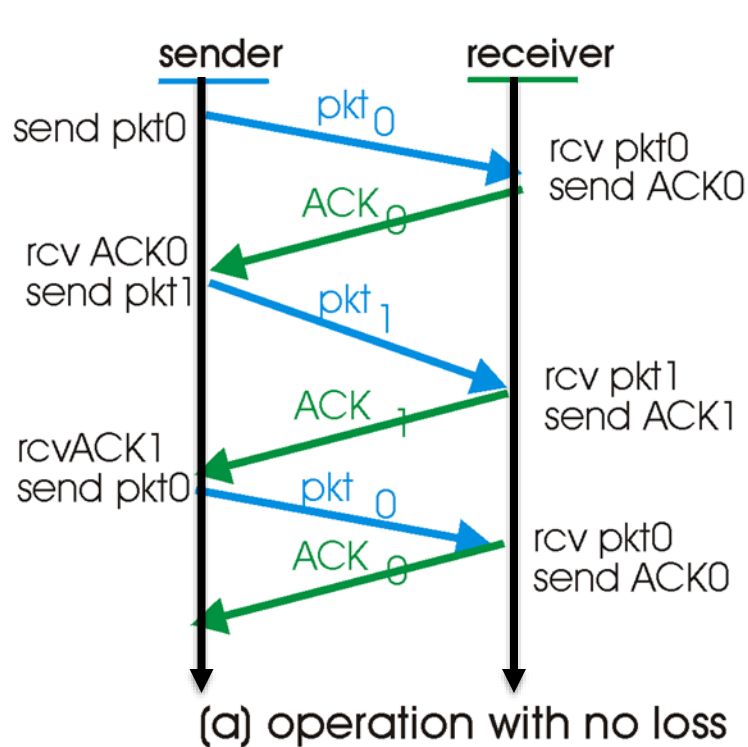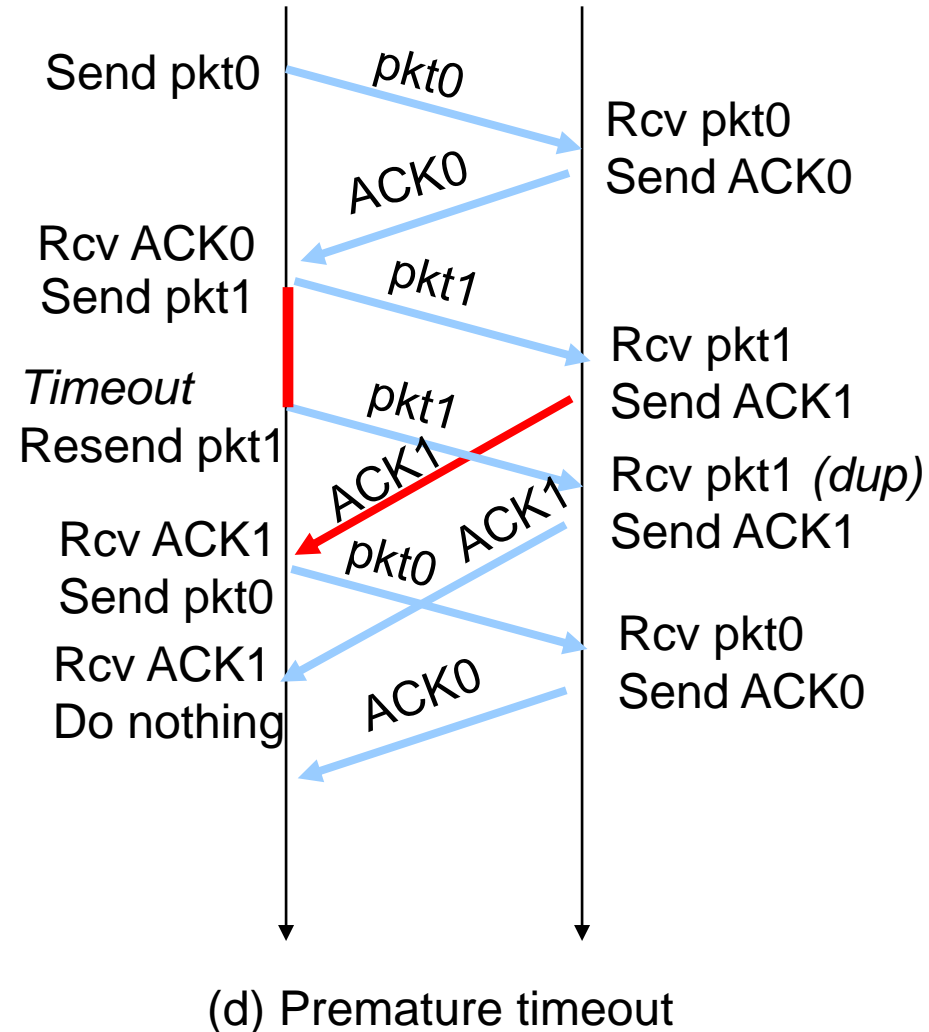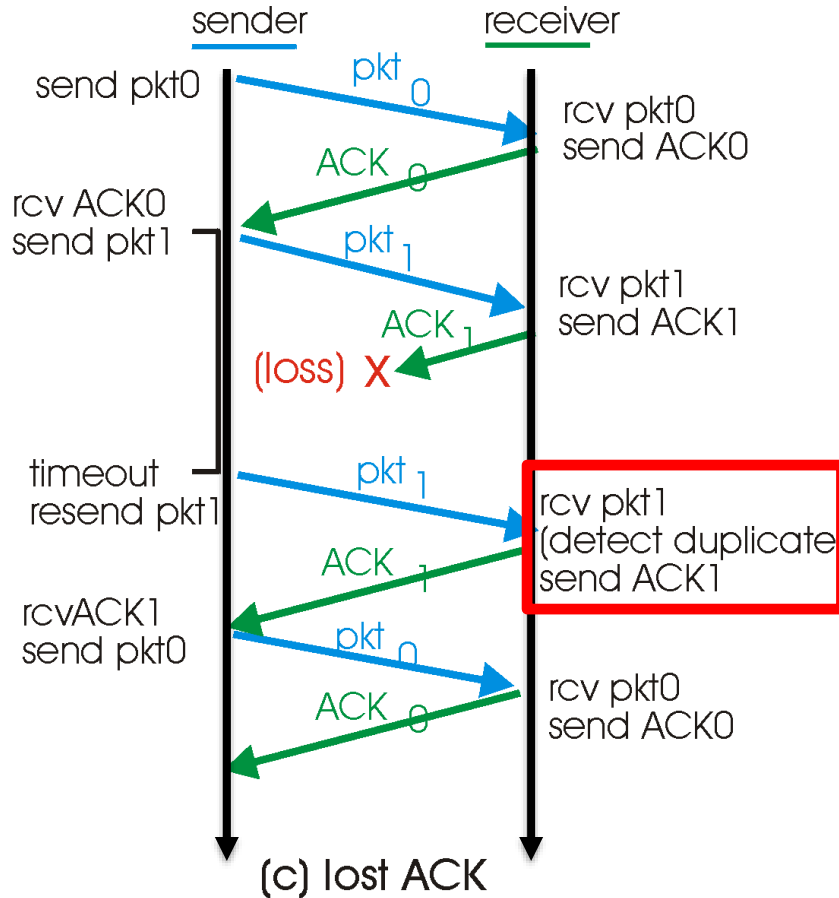
Mechanisms we have learned

- Checksum (to detect bit error)
- ACK/NAK (to notify the sender)
- Sequence number (duplicate data packets)
- Duplicate ACKs (NAK-free)
- Timer (data packet loss)
- Retransmission (a panacea)

Not the end!

rdt 3.0: An illustration - review

(a) operation with no loss

(b) lost packet

Also known as alternating-bit protocol

rdt 3.0: An illustration - review

(c) lost ACK

(d) Premature timeout

- It is essentially a stop-and-wait protocol.

First bit of first packet transmitted, $t = 0$

sender

receiver

Last bit of first packet transmitted, $t = L/R$

first packet bit arrives

RTT

RTT – Round Trip Time
L – Packet Size
R – Transmission Rate

ACK arrives, send next packet, $t = RTT + L/R$
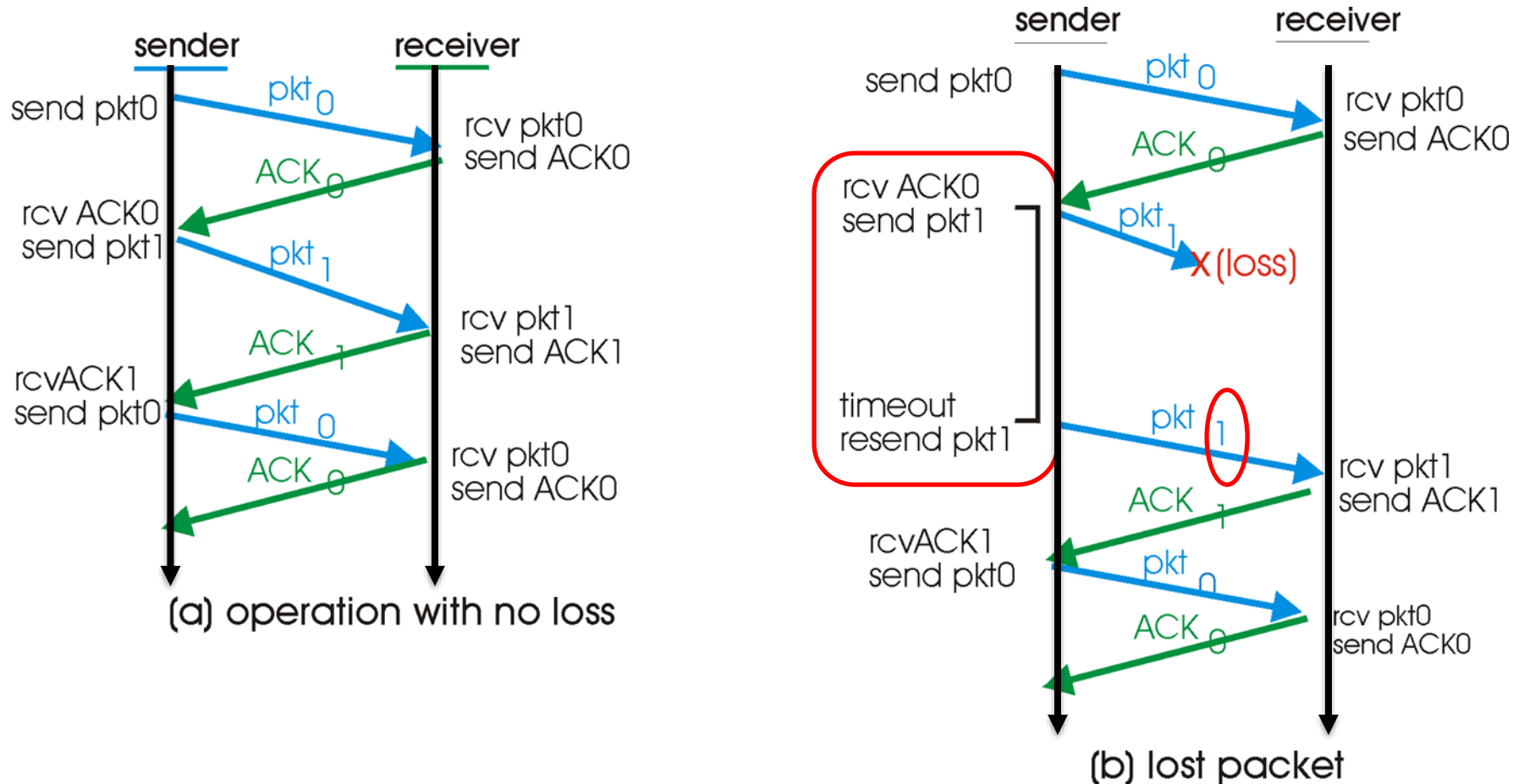
The Sender Utilisation: $U_{sender} = \dfrac{L/R}{RTT + L/R}$
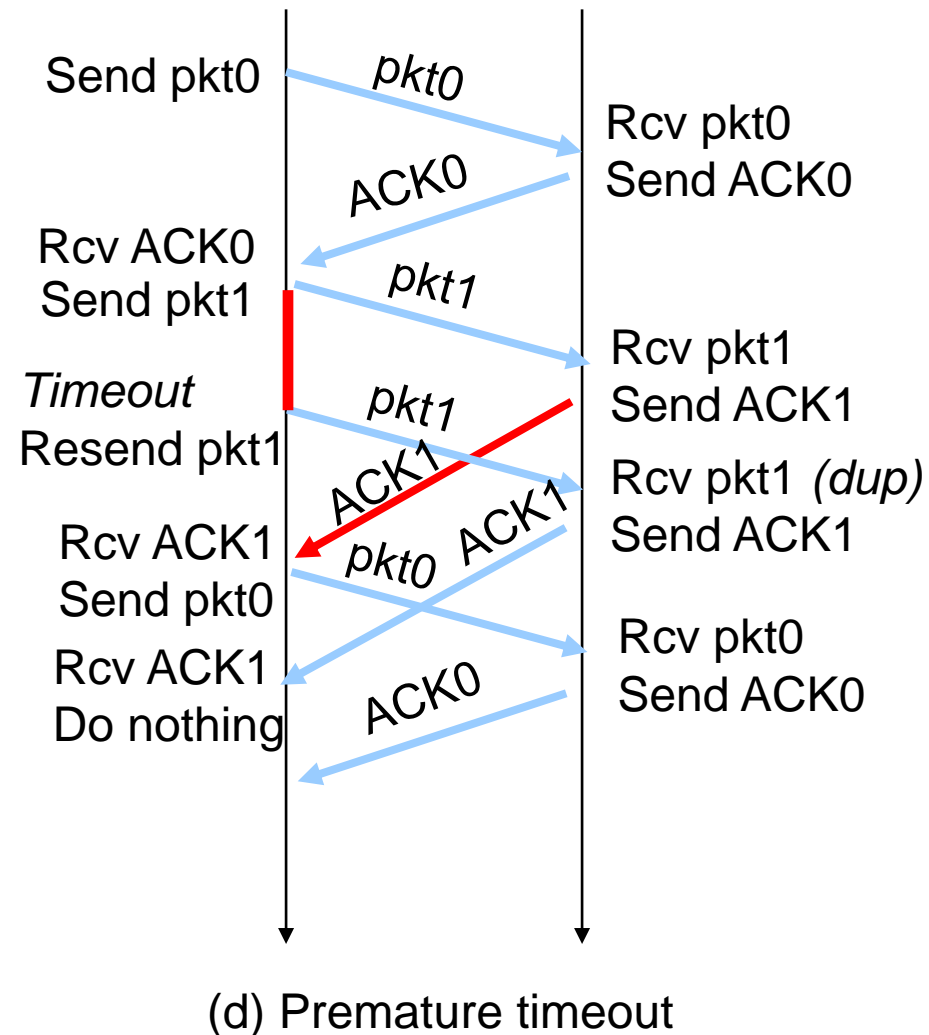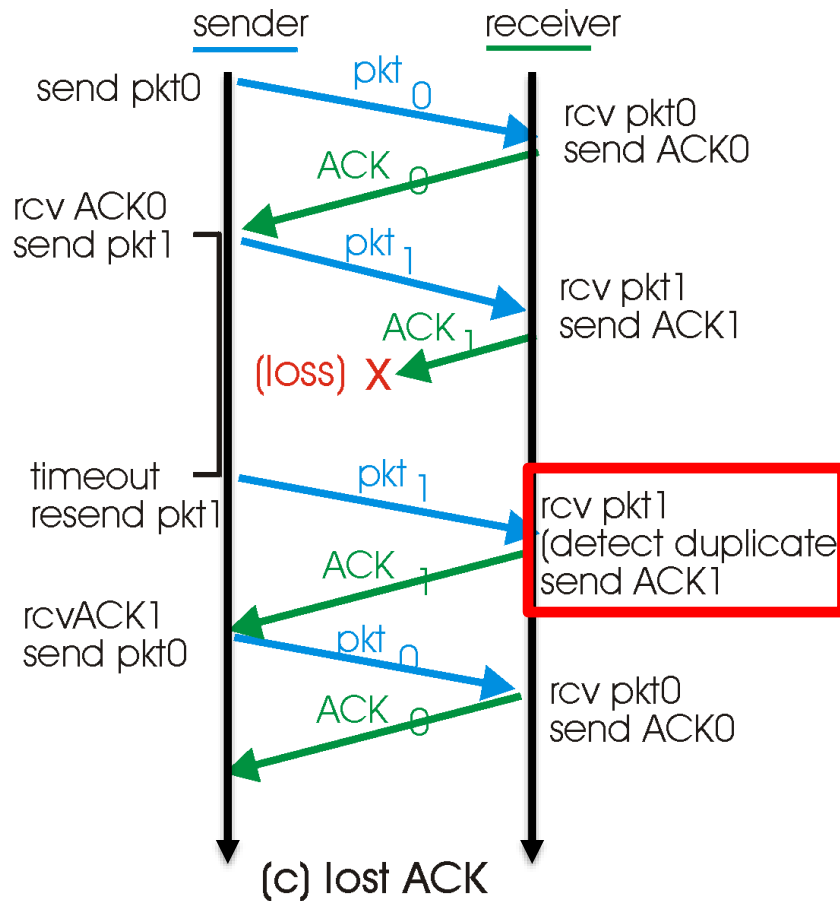
The performance problem of rdt 3.0

- An example: 1 Gbps link, 15 ms end-to-end delay (RTT, 30ms) , 1KB packet:

$$T_{transmit} = \frac{L \text{ (packet length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{8\text{kb/pkt}}{10^9 \text{ b/sec}} = 8 \text{ microsec}$$

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

The sender is waiting in most time!
Very low utilisation! Not mention we have neglected lower layer processing times and queuing delays.

Pipelining
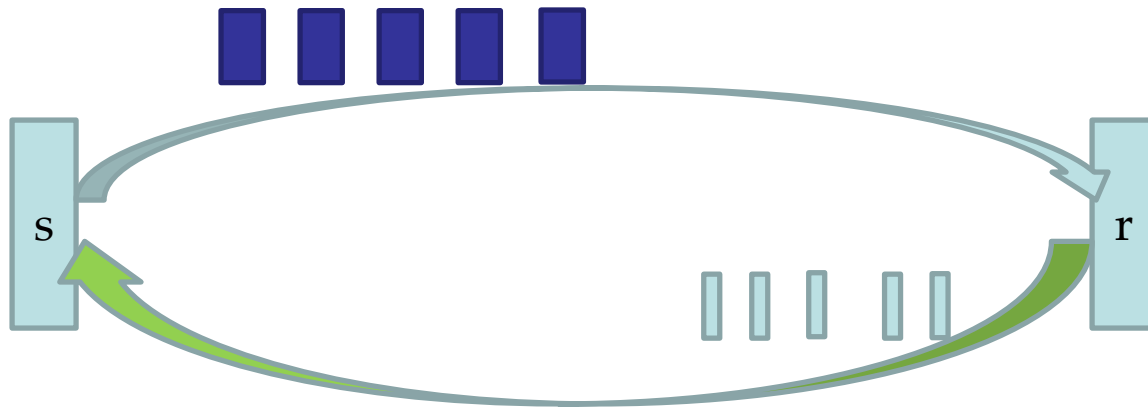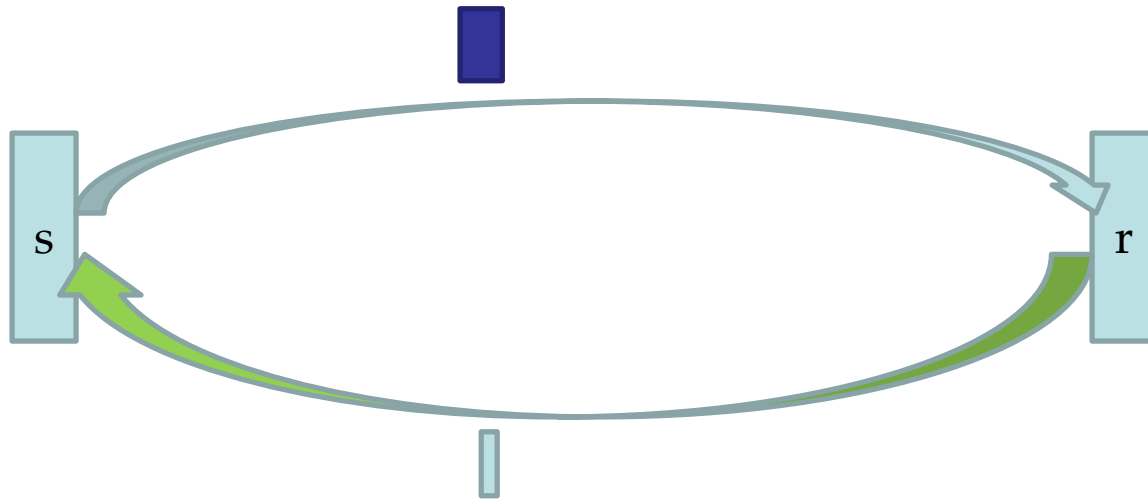
Pipelining
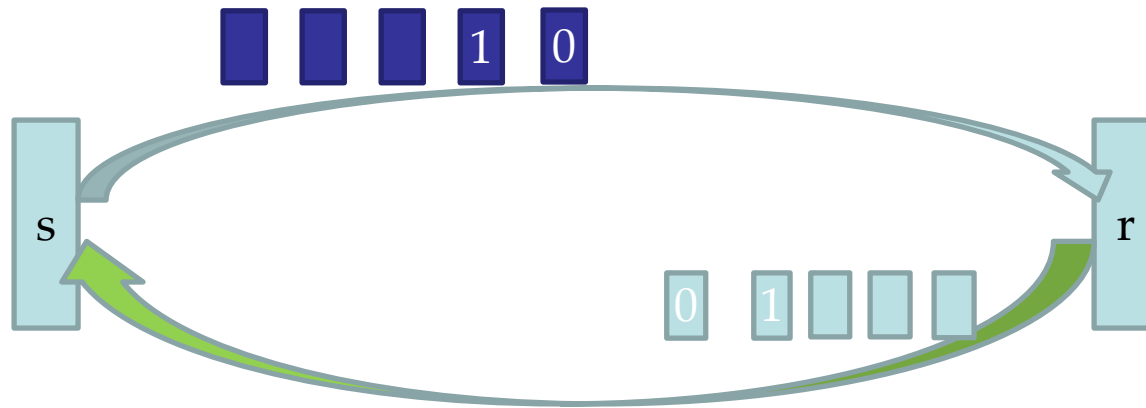
- Using Seq# [0,1] is insufficient;
  - In practice, seq # field occupies k bits, ranging from 0 to 2^k -1. (TCP uses 32 bits for seq#.)
- Sender and receiver may have to buffer more than one packet;
- Then we have two basic approaches based on pipelining.

Go-Back-N protocol

- The sender is allowed to send multiple packets without waiting for an acknowledgement, but is constrained to have *no more than N unacknowledged packets* in the pipeline.

Base: seq# of the oldest unACK'd pkt

Nextseqnum: the smallest unused seq#

The window size is related to flow control and congestion control. (utilisation, performance)

ACK'd seq#

Usable seq#, not yet sent

Window of size *N*

Sent seq#, not yet ACK'd

Not usable seq#
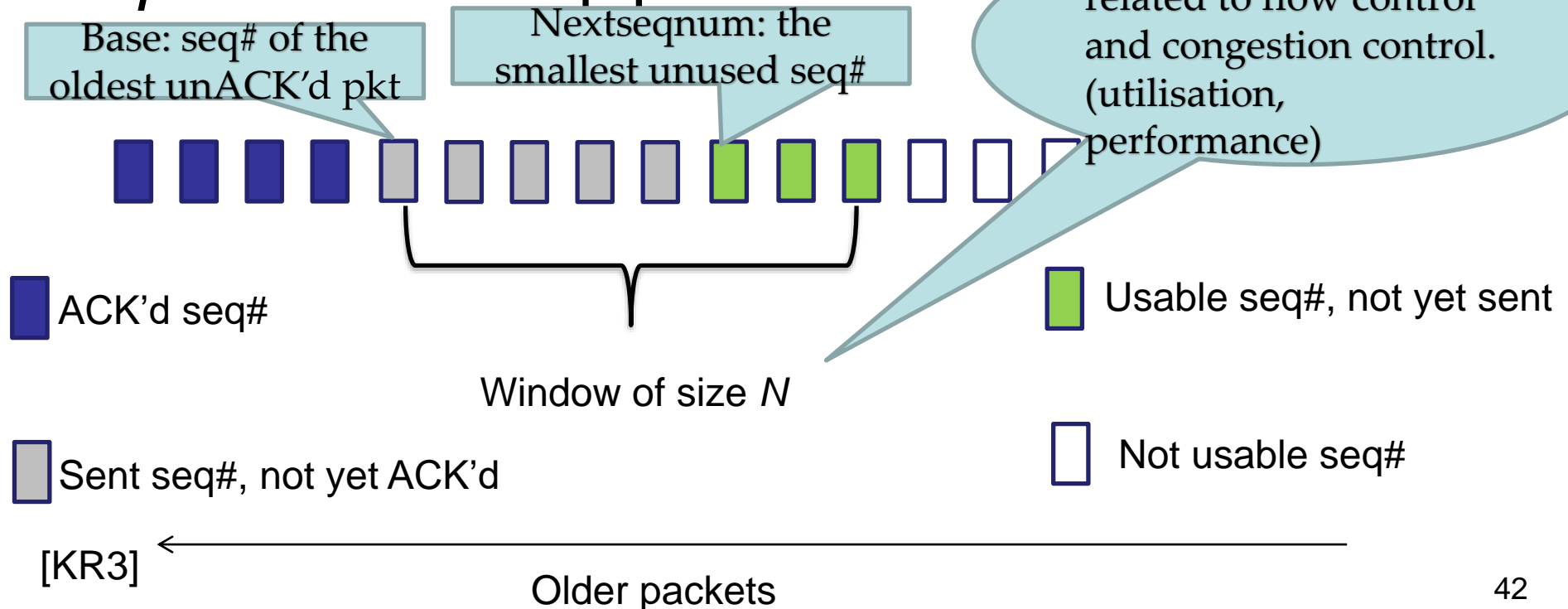
Go-Back-N – Initialisation & Invocation from Above (Sender)

Nextseqnum seq#

Base seq#

ACK'd seq#

Usable seq#, not yet sent

Window of size *N*

Not usable seq#

Sent seq#, not yet ACK'd

Older packets

[KR3]

43
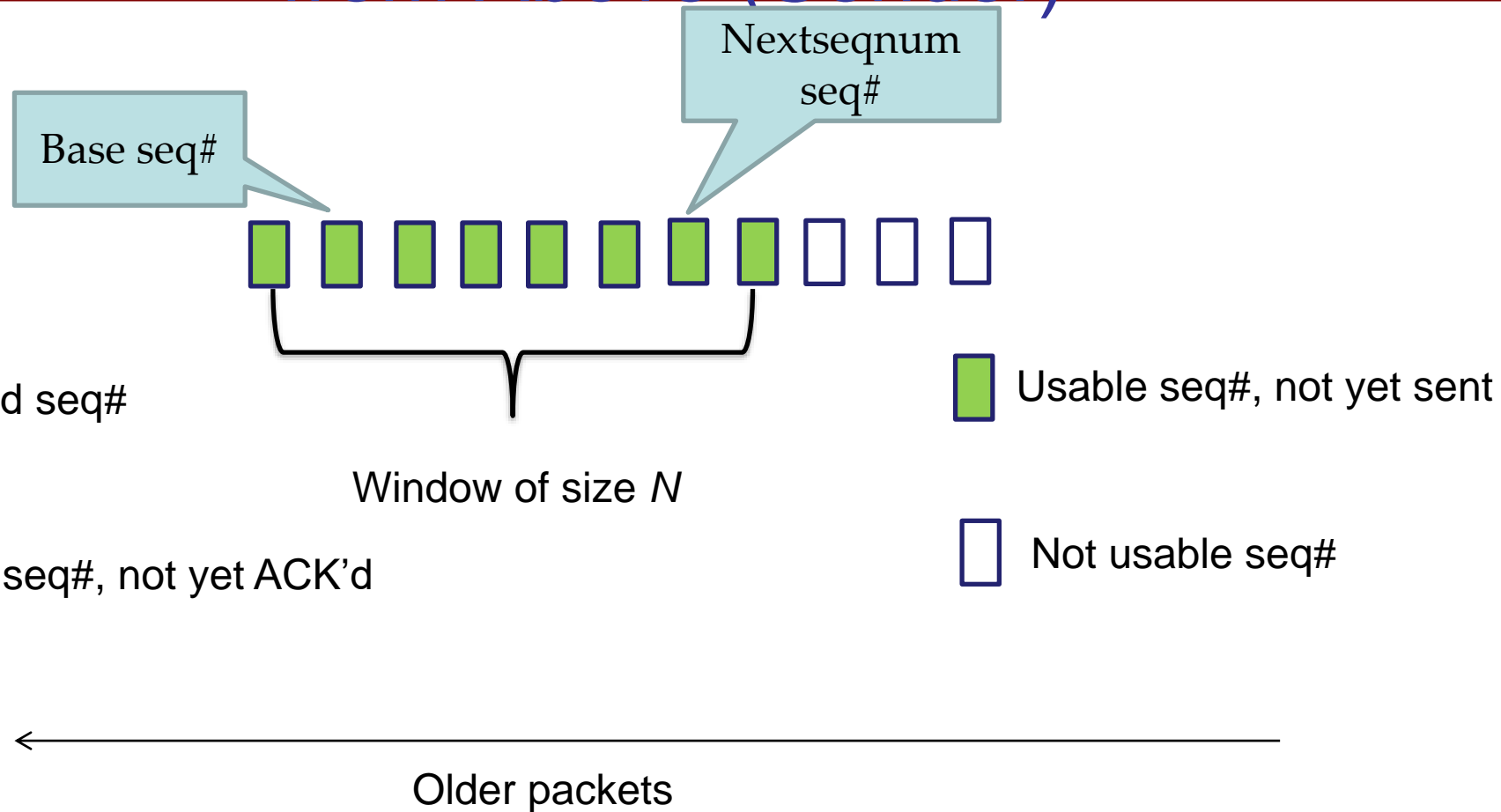
Go-Back-N – Receipt of an ACK (Sender)

Base seq#

Nextseqnum seq#

ACK'd seq#

Usable seq#, not yet sent

Window of size *N*

Sent seq#, not yet ACK'd

Not usable seq#

Older packets

[KR3]
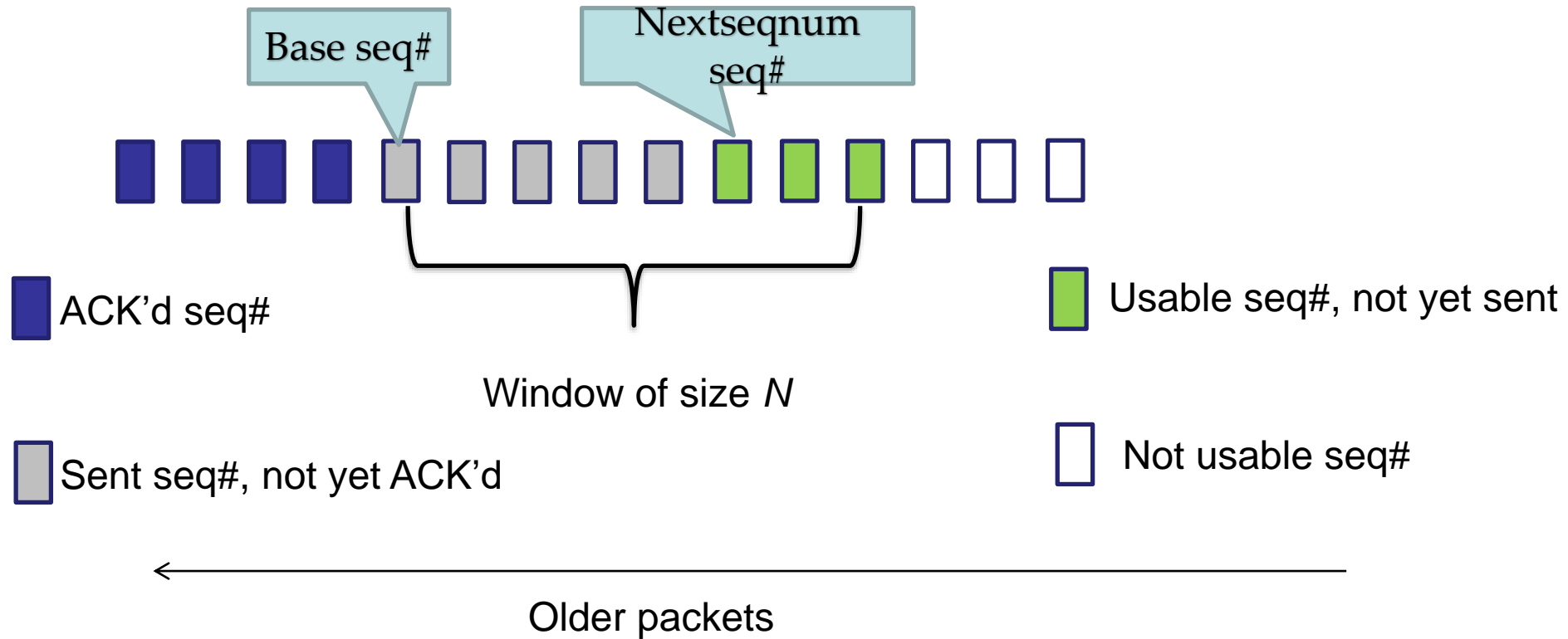
Go-Back-N- Cumulative Acknowledgment (Sender)
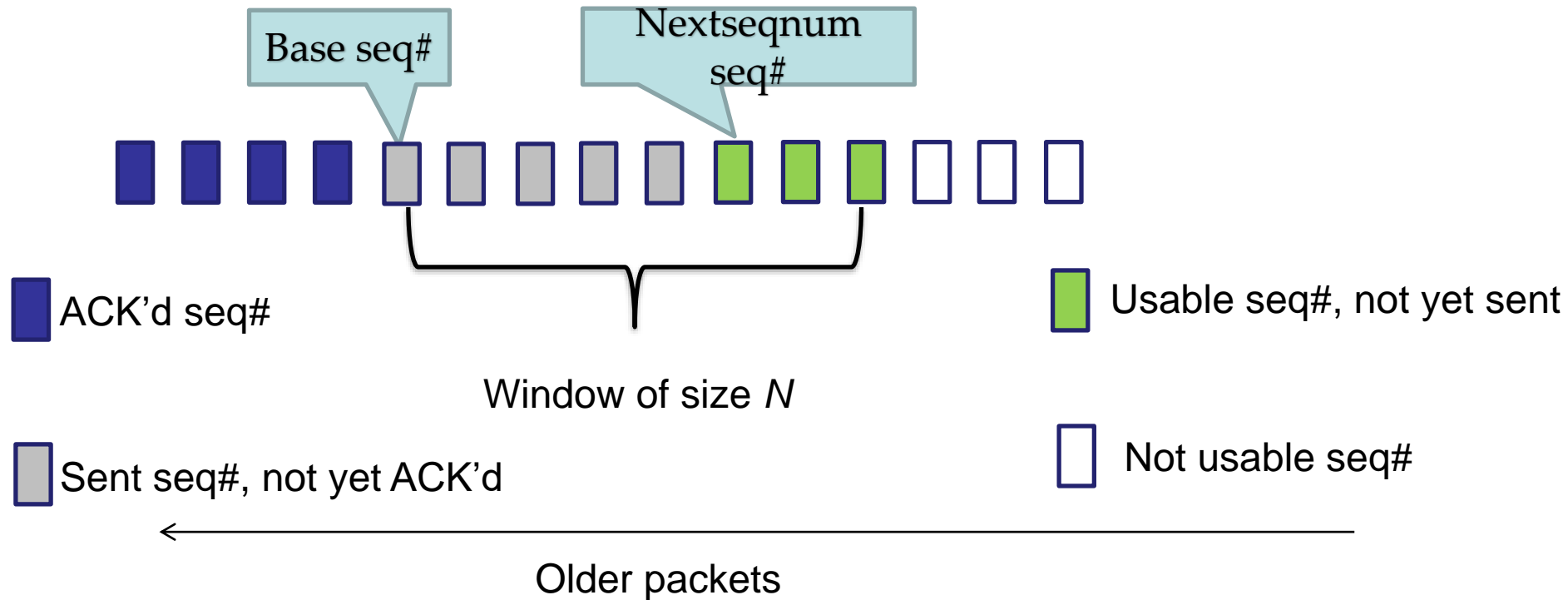
Base seq#

Nextseqnum seq#

ACK'd seq#

Usable seq#, not yet sent

Window of size *N*

Sent seq#, not yet ACK'd

Not usable seq#

Older packets

- An ACK with seq # *n* means that all packets with a seq # up to and including *n* have been correctly received at the receiver.

Data 1

Data 2

ACK 0

Data 3

ACK 0

I am expecting Data 1 and I will discard Data 2 & 3;

When there is a timeout, I will resend *Data 1, 2, 3;*

Receiver: My rule is always expecting the next in-order pkt; if unexpected pkt comes, I always discard them and resend an ACK for the most recently received in-order pkt.

- An ACK with seq # *n* means that all packets with a seq # up to and including *n* have been correctly received at the receiver.

Data 1

ACK 1

Data 2

ACK 2

Data 3

ACK 3

I know *Data 1, 2, and 3* are all received correctly☺
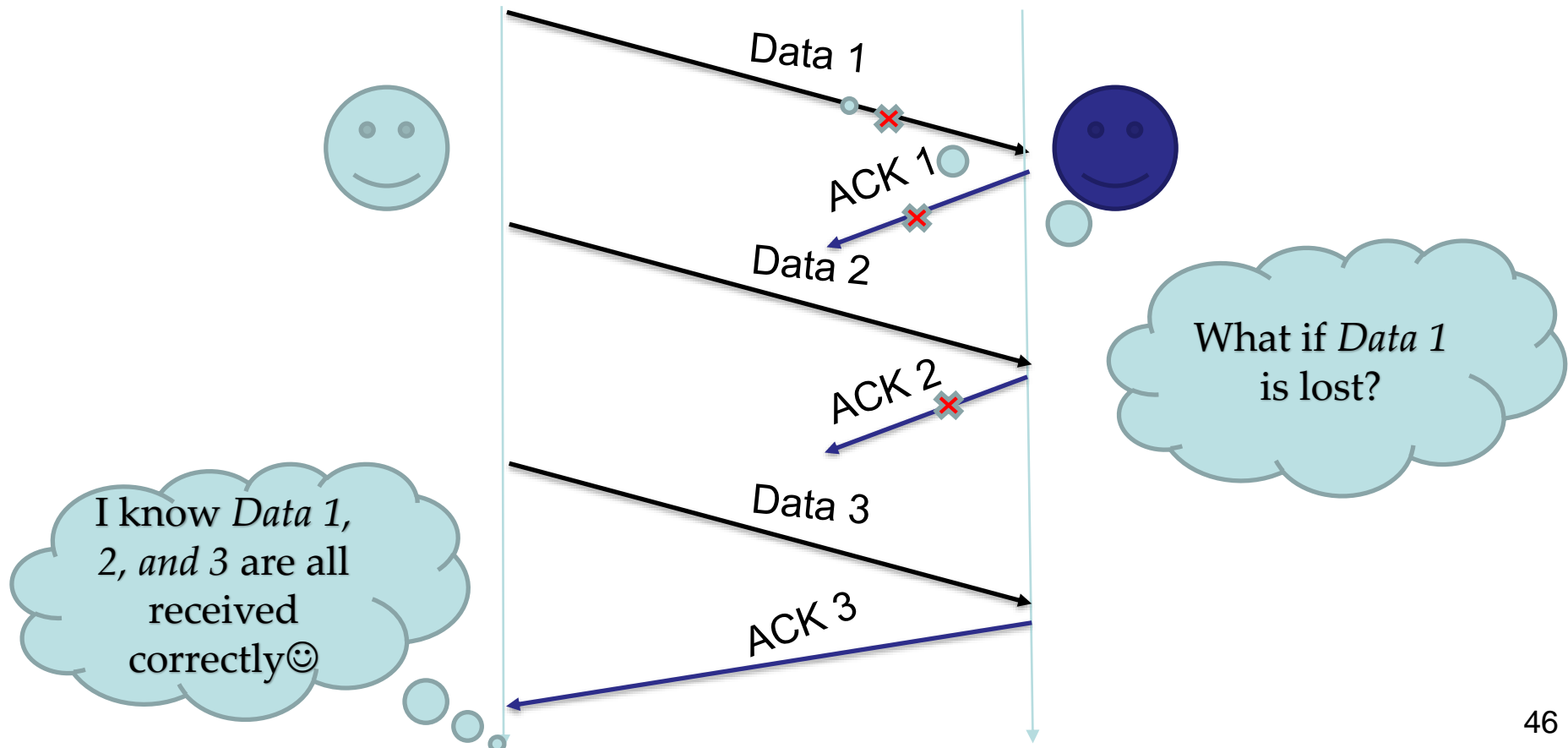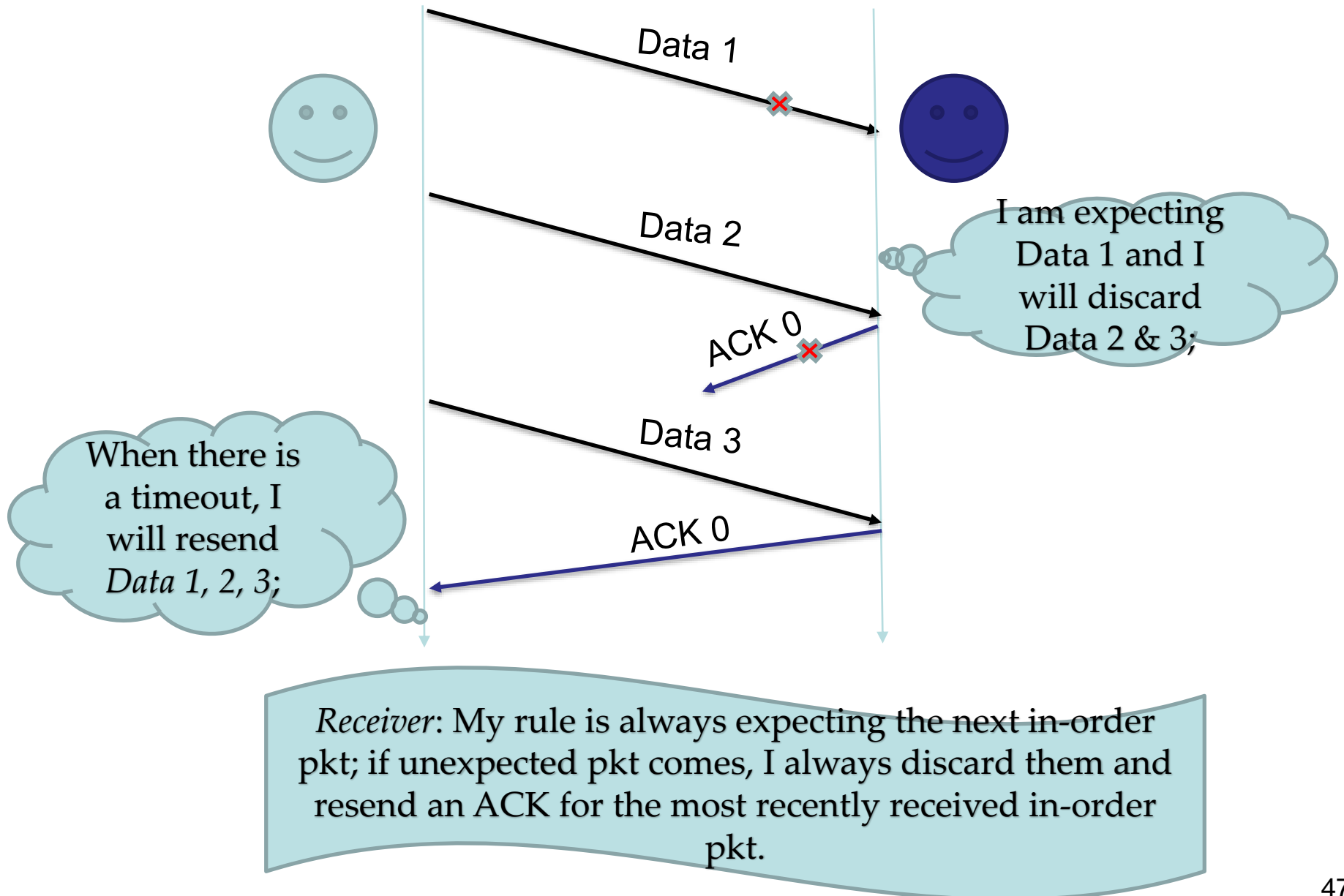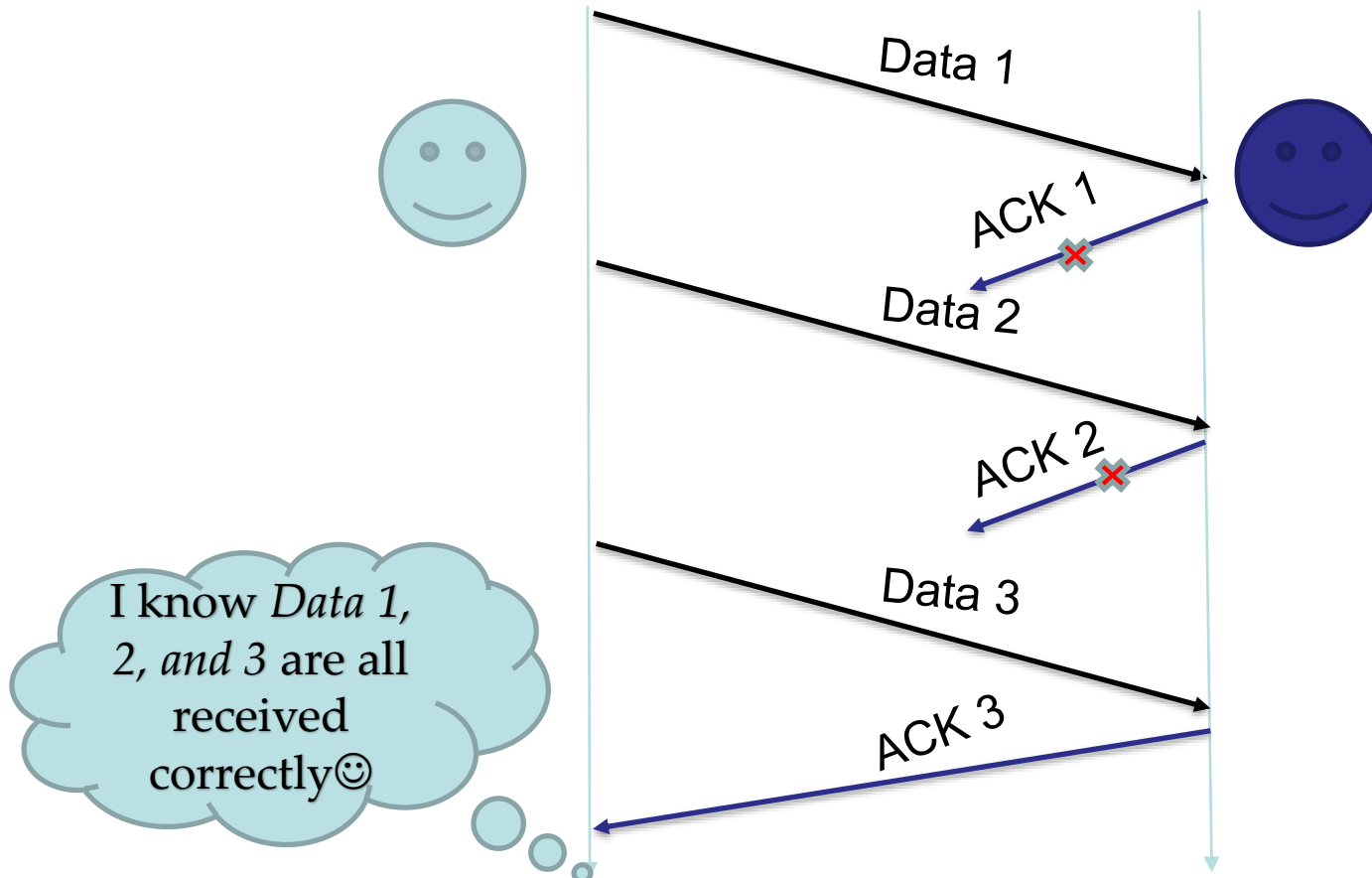
Go-Back-N -timeout

Where there is a timeout:
resend all not yet ACK'd pkts.
(*this could be a bad thing!*)

Base seq#

Nextseqnum seq#
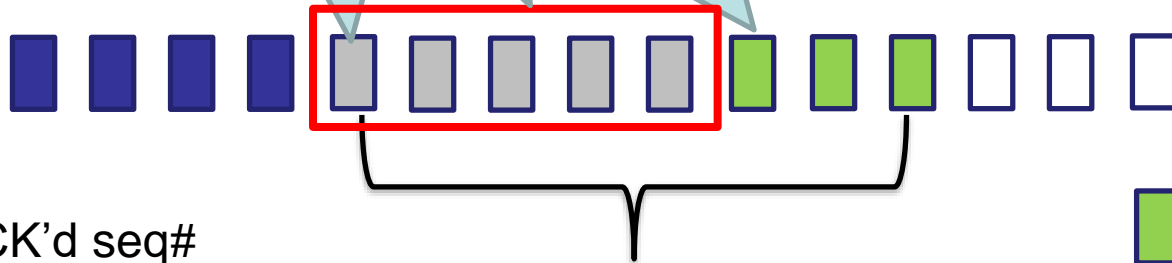
ACK'd seq#

Usable seq#, not yet sent

Window of size *N*

Sent seq#, not yet ACK'd

Not usable seq#

← Older packets

The Receiver Side

Initially, expecting packet 0

0	1	2				2^k -1

When packet 0 arrives uncorrupted,
Deliver data to above;
Send ACK 0;
Now expecting packet 1;

1	2	3			2^k -1	0

Then packet 2 (*out-of-order*)
arrives uncorrupted,
Discard it;
Still send ACK 0;
Still expecting packet 1;

1	2	3			2^k -1	0

The receiver could have buffered packet 2 for later delivery; but if pkt 1 is lost, pkts 1 & 2 will be retransmitted later. For simplicity!

Then packet 1 arrives uncorrupted,
Deliver data to above;
Send ACK 1;
Now expecting packet 2;

2	3	4		2^k -1	0	1

The receiver only needs to keep track of *expectedseqnum* (a variable).

[AK05, KR3]

Ressend all unACK'd pkts

↑

Timeout(the oldest pkt is lost)

↑

Receiver will discard all following less older pkts if it receives some;

Cumulative ACK(n) (Sender assures itself all unACK'ed pkts with seq# no bigger than n have arrived without corruption.)

↑

Receiver always sends back ACK for the most recently arrived in-order pkt;

↑

Receiver receives an in-order pkt; sends back ACK for this pkt;

↑

Receiver receives an out-of-order pkt; sends back ACK for the most recently arrived in-order pkt;

Go-Back-N: sender extended FSM

rdt_send(data)
_____

if (nextseqnum < base+N) { **/*If we are allowed to send packets*/**
   sndpkt[nextseqnum] = make_pkt(nextseqnum,data,chksum)
   udt_send(sndpkt[nextseqnum])
   if (base == nextseqnum) **/*If there are no previous packets in flight*/**
     start_timer   **/*timer for the just sent packet*/**
   nextseqnum++ **/*Move to the next "smallest unused sequence #"*/**
  }
else
 refuse_data(data)

Λ
_____

base=1
nextseqnum=1

Wait

rdt_rcv(rcvpkt)
  && corrupt(rcvpkt)
_____

Λ

timeout
_____

start_timer **/*timer for the to-be-sent packet*/**
**/*Resend all previously send ack packets that
have not been ACKed*/**
udt_send(sndpkt[base])
udt_send(sndpkt[base+1])
…
udt_send(sndpkt[nextseqnum-1])
                 Go back

rdt_rcv(rcvpkt) &&
  notcorrupt(rcvpkt)
_____

base = getacknum(rcvpkt)+1 **/*slide the window to its right*/**
If (base == nextseqnum) **/*If there are no packets in flight*/**
  stop_timer **/* stop timer because there are not outstanding pkt*/**
 else
  start_timer **/*timer for the current oldest packet*/**

[AK05, KR3]

53

Go-Back-N: Receiver extended FSM

default /*other cases other than receiving in-order packets*/
udt_send(sndpkt) /*send an ACK for the most recently received in-order packet*/

Λ

expectedseqnum=1
sndpkt =
 make_pkt(expectedseqnum,**ACK**,chksum)

Wait

rdt_rcv(rcvpkt) /*receive in-order packets*/
 && notcurrupt(rcvpkt)
 && hasseqnum(rcvpkt,expectedseqnum)

extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(expectedseqnum,**ACK**,chksum)
udt_send(sndpkt)
expectedseqnum++

[AK05, KR3]

Frames buffered until acknowledged

Frames already transmitted

Window of frames that may be transmitted

· · · | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | · · · ·

Frame sequence number

Last frame acknowledged

Last frame transmitted

Window shrinks from trailing edge as frames are sent

Window expands from leading edge as ACKs are received

(a) Sender's perspective

Frames already received

Window of frames that may be accepted

· · · | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | · · · ·

Last frame acknowledged

Last frame received

Window shrinks from trailing edge as frames are received

Window expands from leading edge as ACKs are sent

(b) Receiver's perspective

This part grows when new ACKs arrive and shrinks when new data is sent.

Base seq#

Nextseqnum seq#

ACK'd seq#

Usable seq#, not yet sent

Window of size *N*

Sent seq#, not yet ACK'd
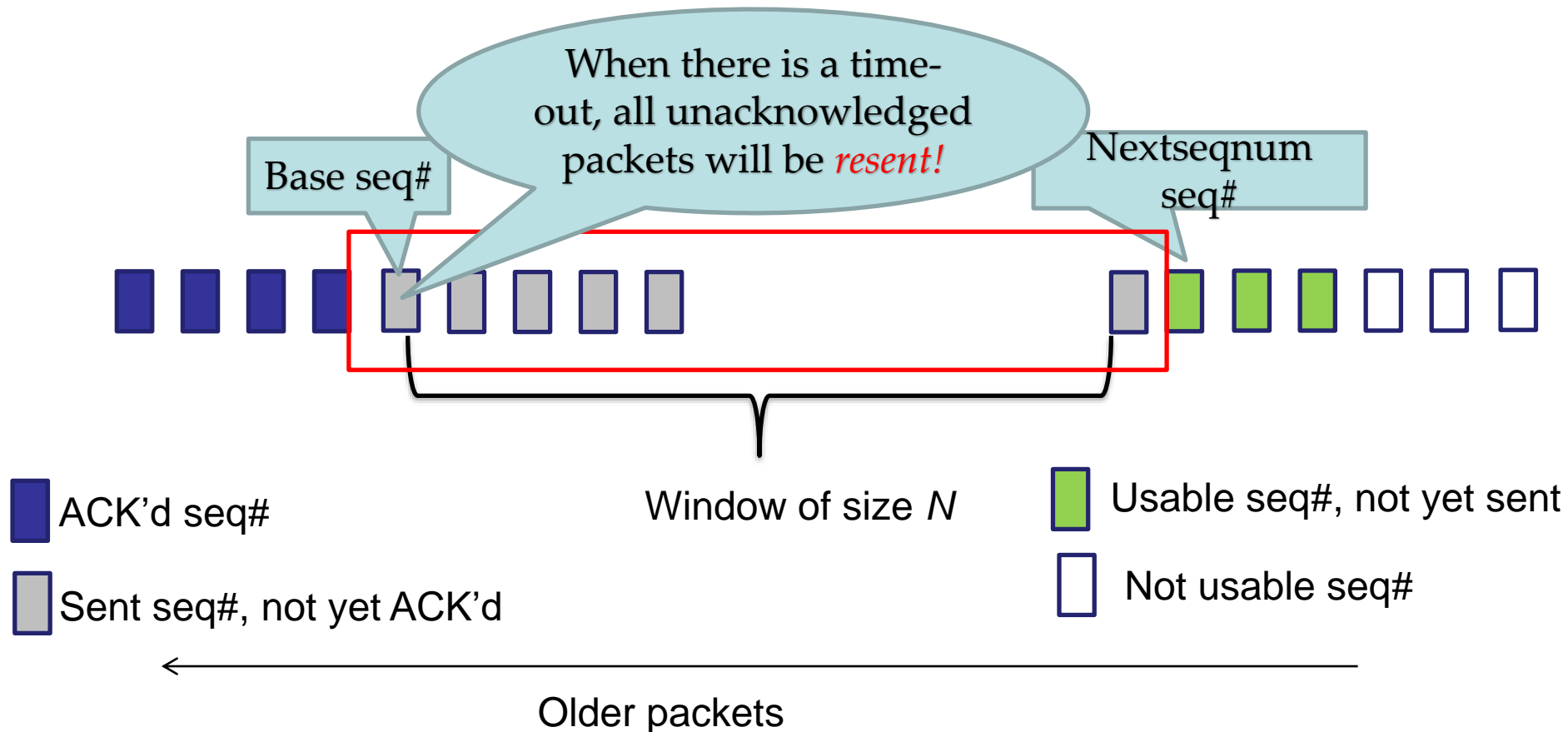
Not usable seq#

Older packets

Tricks so far

- Checksum (bit errors)
- Seq # (new or duplicate)
- Duplicate ACK (NAK-free)
- Timer (to detect packet loss)
- Retransmission (to recover from transmission errors)
- Cumulative ACK (ACK all packets with no bigger seq#, in Go-Back-N)
- Window, pipelining (to improve sender utilisation, flow control, congestion control)

Go-Back-N is great!... But it has its own performance problem☹

Go-Back-N: Many packets in pipeline

When there is a time-out, all unacknowledged packets will be *resent!*

Base seq#

Nextseqnum seq#

ACK'd seq#

Sent seq#, not yet ACK'd

Window of size *N*

Usable seq#, not yet sent

Not usable seq#

Older packets

Selective Repeat/Reject (SR)

- receiver *individually* acknowledges all correctly received pkts
  - buffers pkts, as needed, for eventual in-order delivery to upper layer
- sender only resends pkts for which ACK not received
  - sender timer for each unACKed pkt
- sender window
  - N consecutive seq #'s
  - again limits seq #s of sent, unACKed pkts

[AK05]

SR: Sender window

send_base

nextseqnum

Window of size *N*

ACK'd seq#

Sent seq#, not yet ACK'd

Usable seq#, not yet sent

Not usable seq#

← Older packets

SR: Receiver window

Rcv_base

Rcv_base + N-1

Window of size N

ACK'd seq#; out-of-order Buffered;

Expected seq#, not yet received

Acceptable seq#

Not usable seq#

Older packets
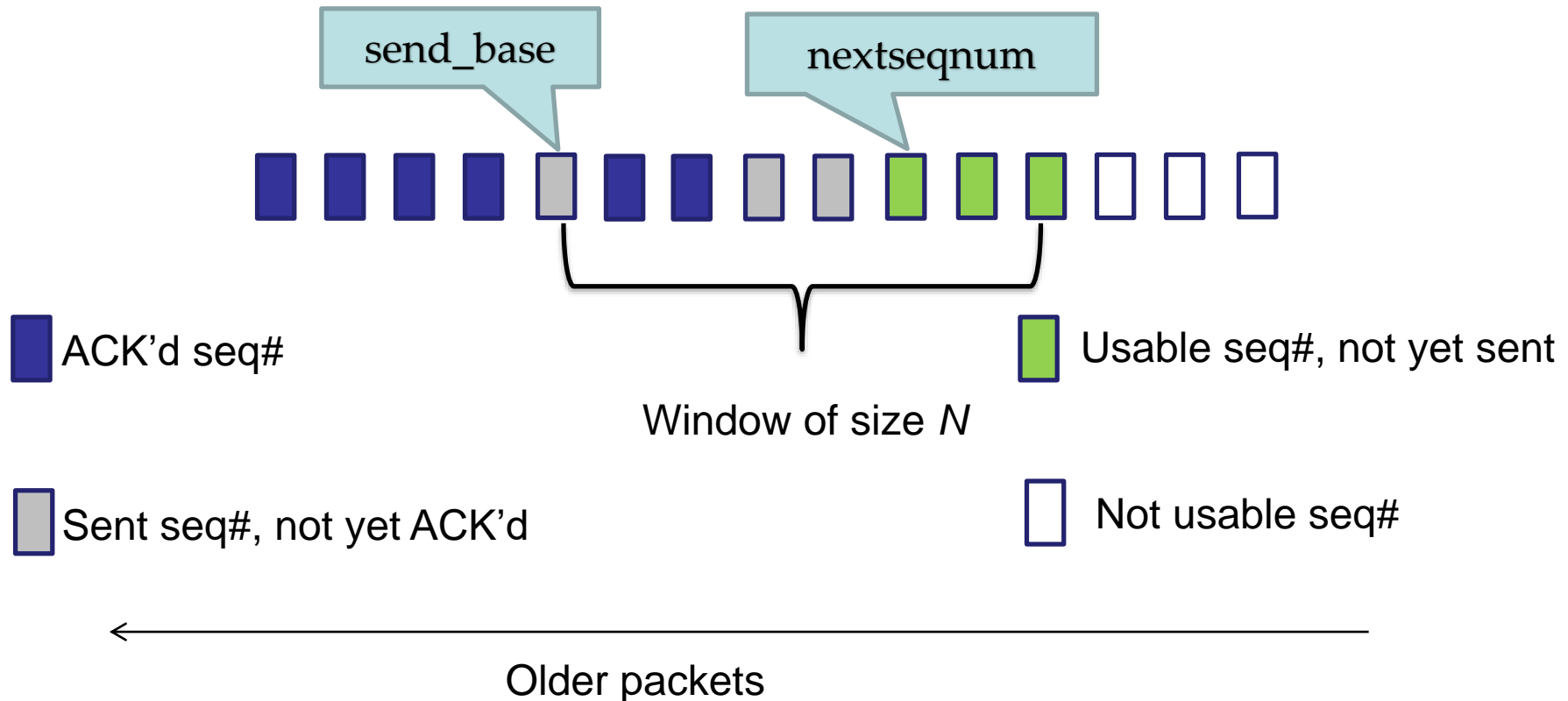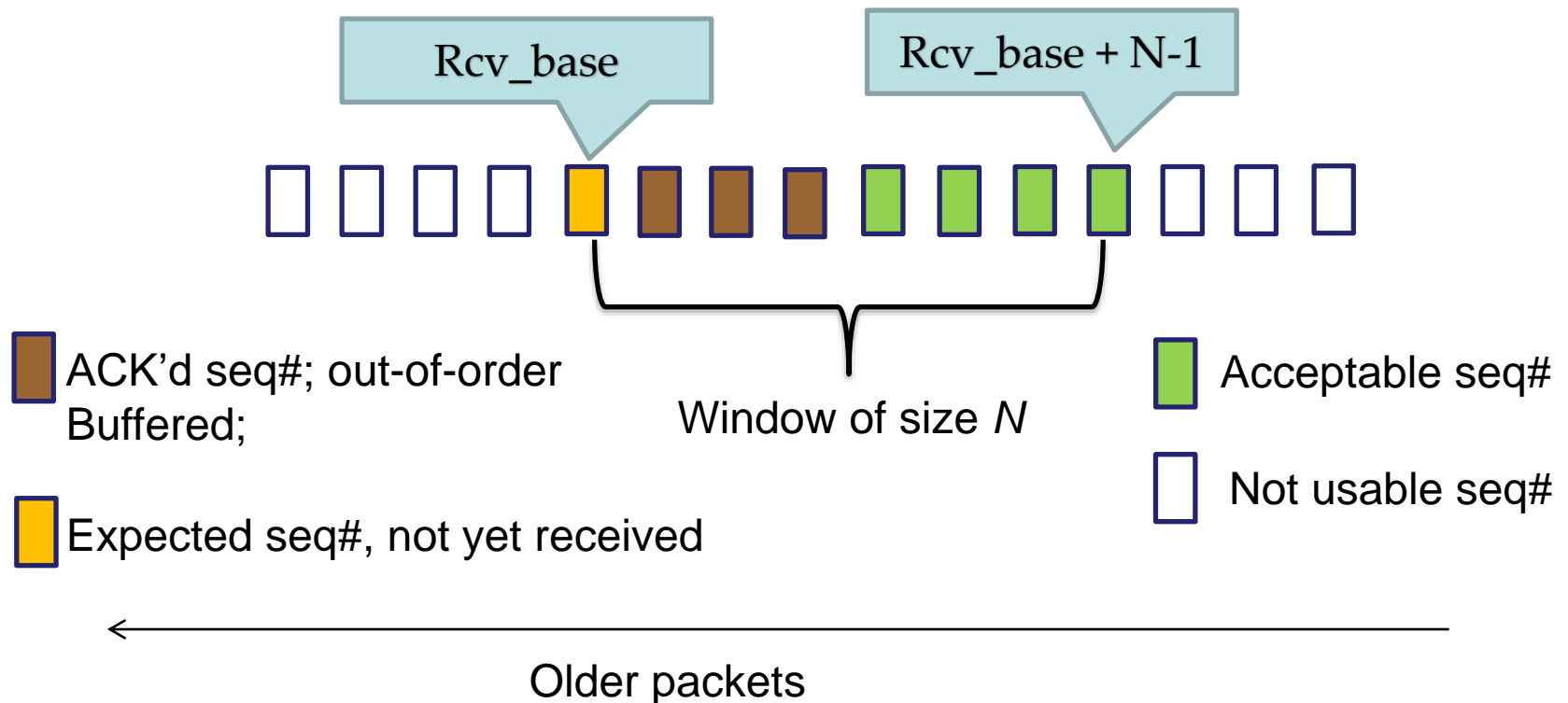
SR: Sender window – receipt of an ACK(1)

sendbase

nextseqnum

ACK'd

Window of size *N*

ACK'd seq#

Usable seq#, not yet sent

Sent seq#, not yet ACK'd

Not usable seq#

sendbase

nextseqnum

ACK'd

Window of size *N*
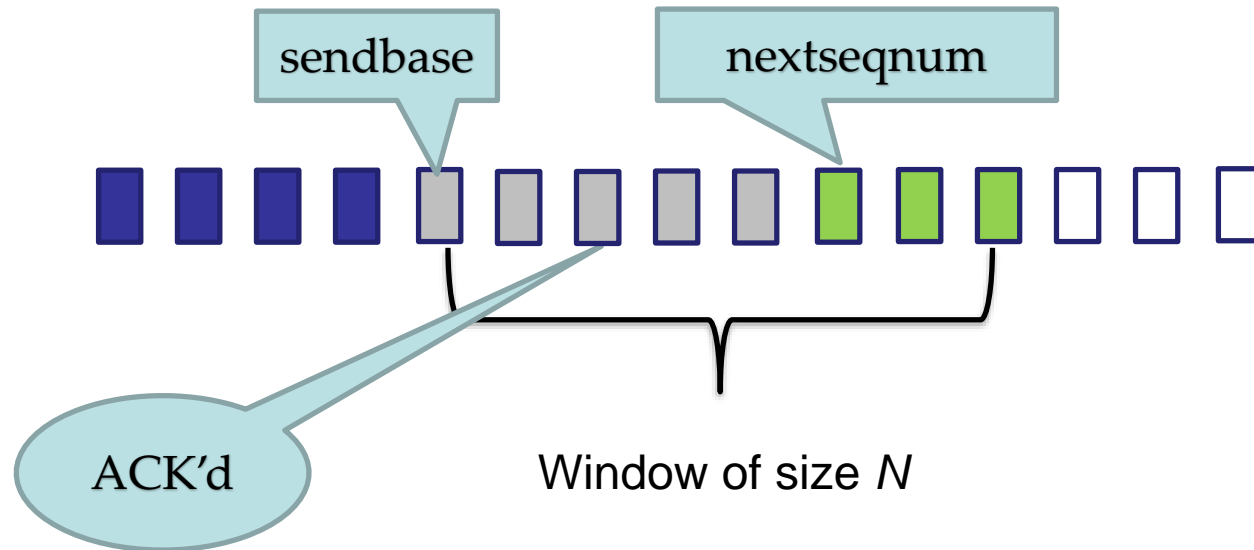
ACK'd seq#

Usable seq#, not yet sent

Sent seq#, not yet ACK'd

Not usable seq#
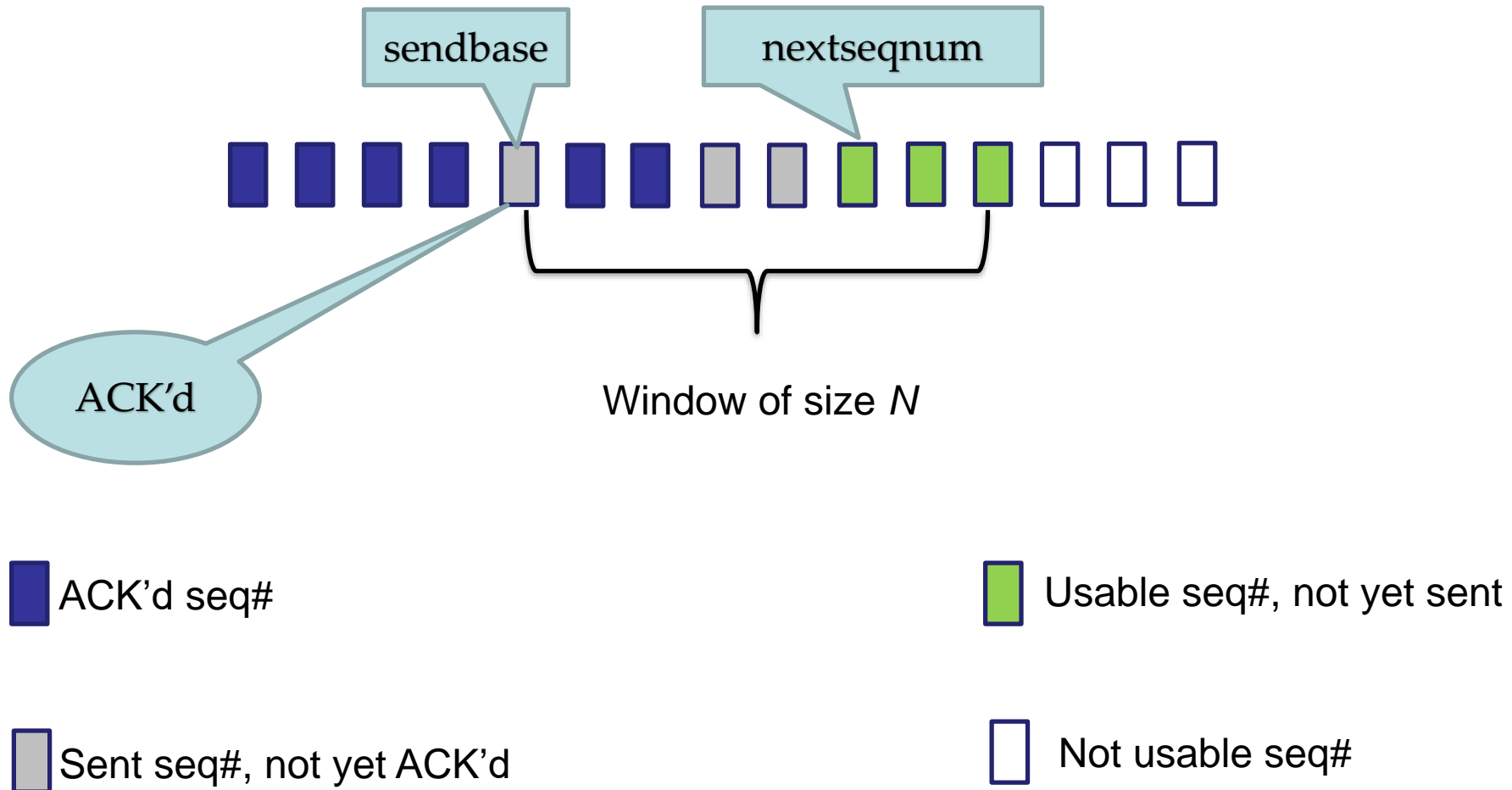
SR: Sender window – timeout

sendbase

nextseqnum

Timeout for pkt with this seq#; resend this pkt *only*

Window of size *N*

ACK'd seq#

Usable seq#, not yet sent

Sent seq#, not yet ACK'd

Not usable seq#

Older packets
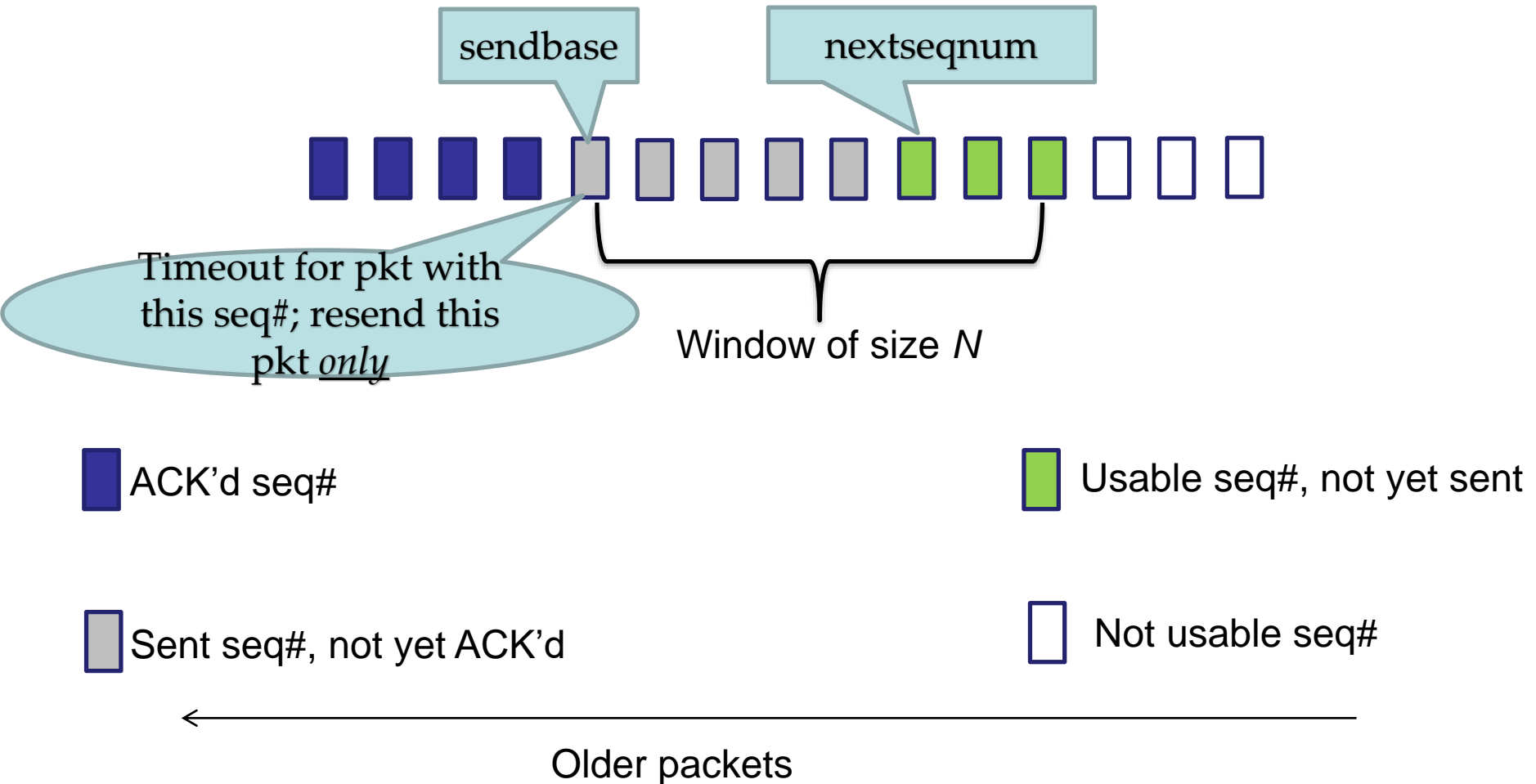
SR: sender side

sender

data from above :

- if next available seq # in window, send pkt

timeout(n):

- resend pkt n, restart timer

ACK(n) in [sendbase,sendbase+N-1] (*more precisely [sendbase, nextseqnum-1]*):

- mark pkt n as received
- if n smallest unACKed pkt, advance window base to next unACKed seq #

The only case when sender window moves.

SR: Receiver window (1) – new pkt not the expected

Receiver receives new packet; buffer it and send a *selective* ACK;

Rcv_base

Window of size *N*
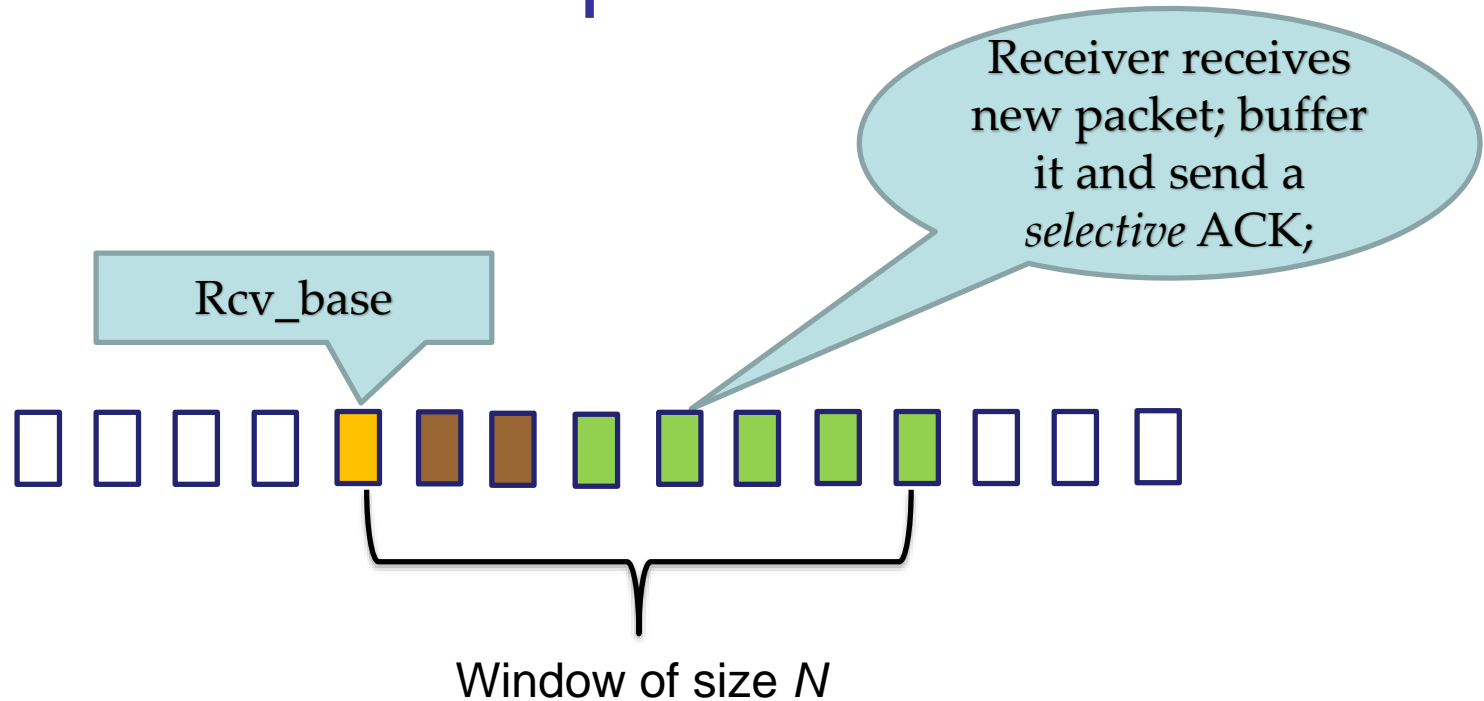
ACK'd seq#; out-of-order Buffered;

expected seq#, not yet received

acceptable seq#
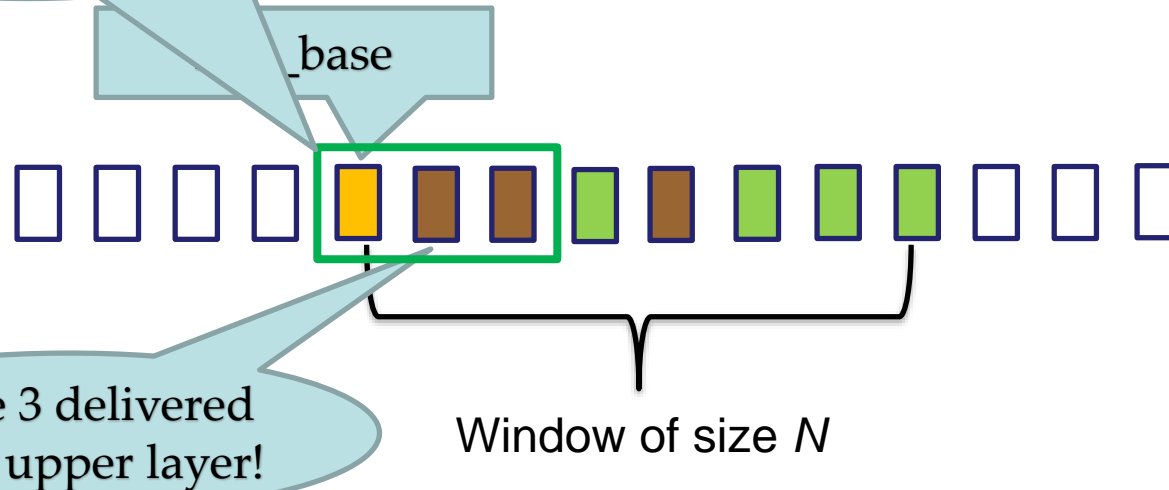
Not usable seq#

SR: Receiver window (2) – old pkt

Receiver receives an old packet (retransmission); ignore it and send a *selective* ACK again;

Rcv_base

Window of size *N*

ACK'd seq#; out-of-order Buffered;

acceptable seq#

Not usable seq#

expected seq#, not yet received

Receiver receives an expected pkt; it will send a *selective* ACK; deliver buffered and consecutively numbered pkts to above;

_base

These 3 delivered to the upper layer!

Window of size *N*

ACK'd seq#; out-of-order Buffered;

expected seq#, not yet received

acceptable seq#

Not usable seq#

Rcv_base

Window of size *N*

This pkt delivered to the upper layer!

ACK'd seq#; out-of-order Buffered;

expected seq#, not yet received

acceptable seq#

Not usable seq#

Rcv_base

Receiver receives a delivered pkt; it will send an ACK (sender might be waiting for this ACK).

Window of size *N*

ACK'd seq#; out-of-order Buffered;

acceptable seq#

Not usable seq#

expected seq#, not yet received

SR: sender and receiver protocols

receiver

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

- ❑ send ACK(n)
- ❑ out-of-order: buffer
- ❑ in-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt
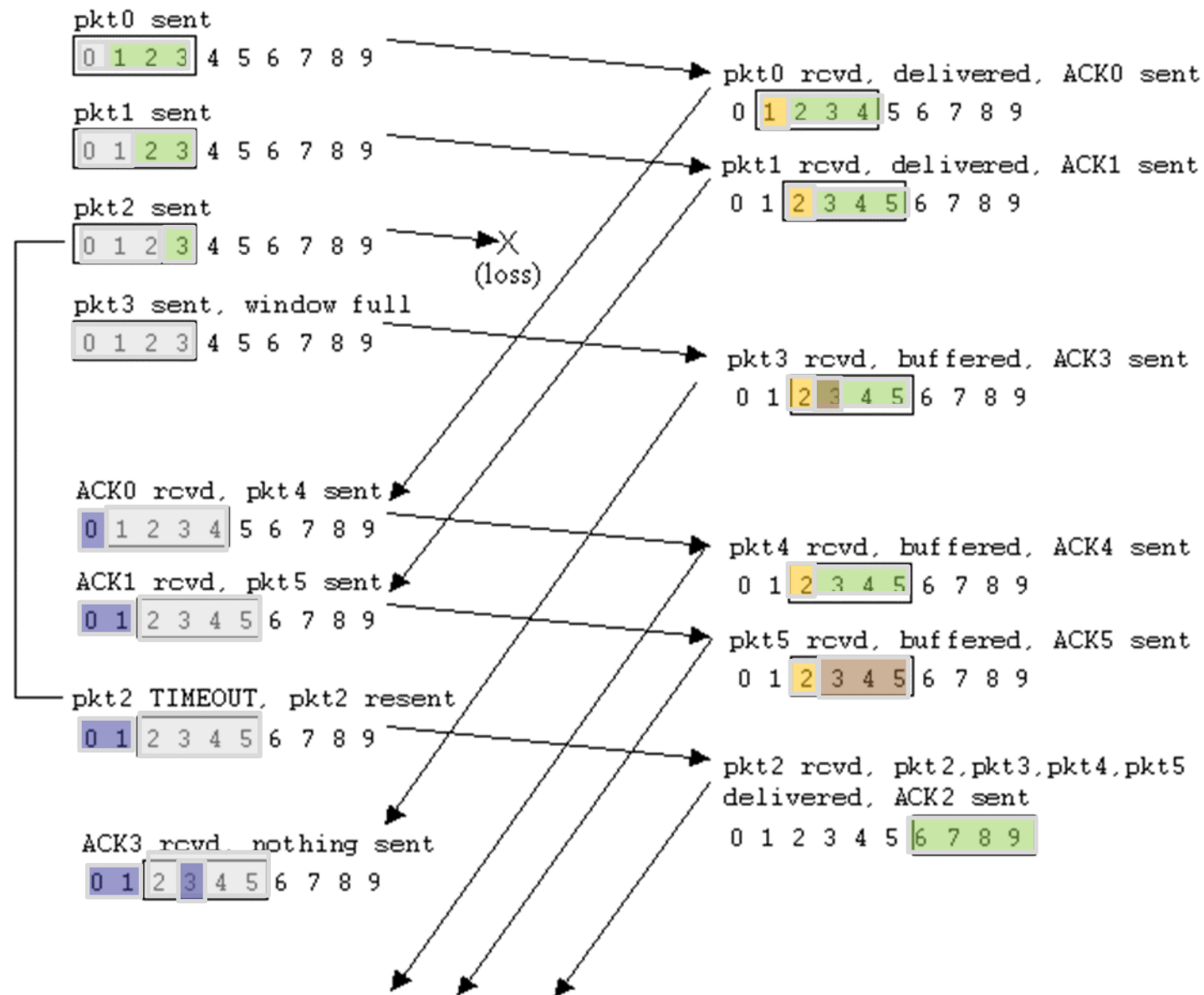
pkt n in [rcvbase-N,rcvbase-1]

- ❑ ACK(n) (*Sender is retransmitting n – it has not received ACK(n); resend it to make sender's window move;*)

otherwise:

- ❑ ignore

SR: receiver dilemma with too large window

Example:

- seq #'s: 0, 1, 2, 3

- window size=3

  > A retransmission with seq# 0

- receiver sees no difference in two scenarios!

> Q: what relationship between seq # space and window size?

(a)

(b)

> A new transmission with seq# 0

[AK05, KR3]

74
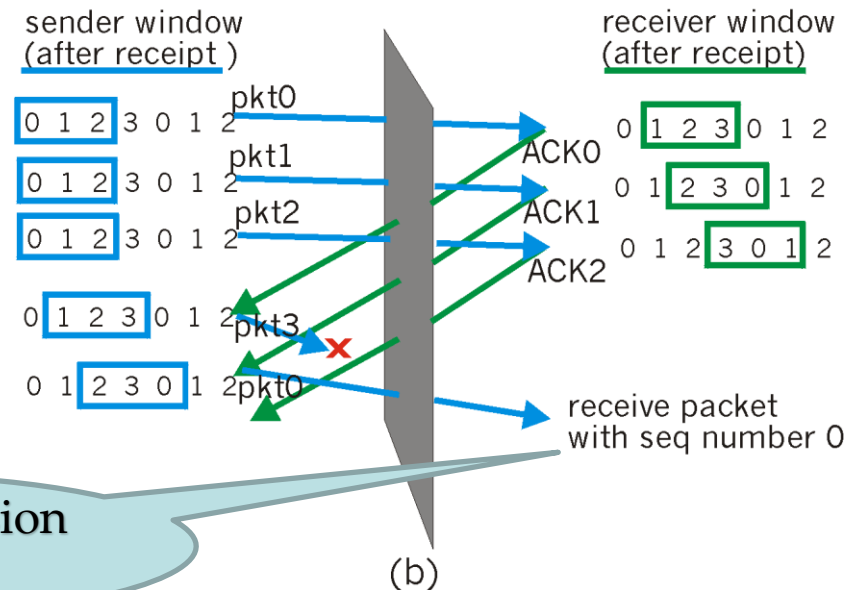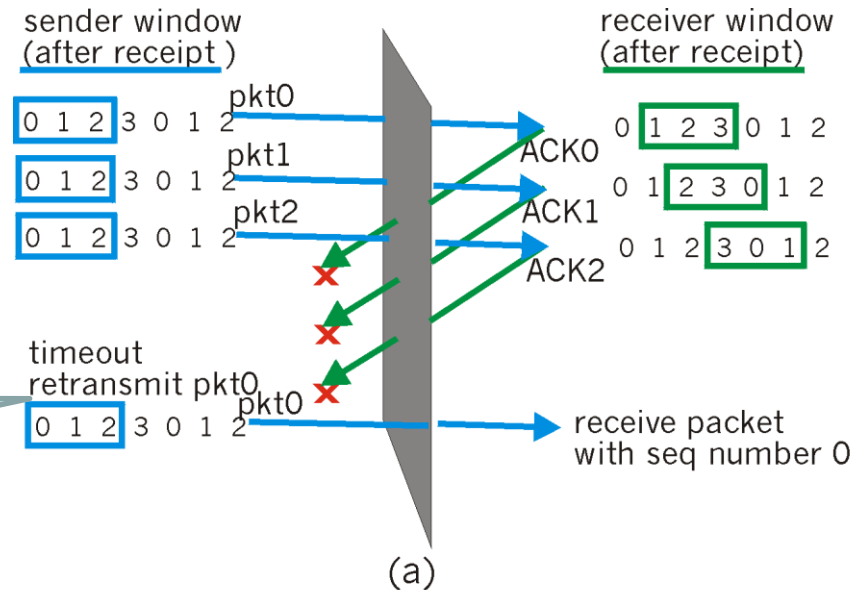
Window size vs seq# space

Q: what relationship between seq # space and window size?

A: window size $\leq \frac{1}{2}$ (size of *seq# space*).

e.g., we use an 8-bit field for seq#; the window size should be no greater than $2^8/2 = 2^7 = 128$;

On the other hand, if we use a window of size 4, we should at least use a 3-bit seq# field.

A hidden assumption

- We assume a packet will NOT be reordered within the channel between the sender and the receiver;
  - Reasonable for a single link;
  - Reordering could happen when the sender and receiver is connected by a multi-hop path;

- Old copies of a packet or an ACK with seq# *x* can appear. Neither sender's window or receiver's window contains *x*;
  - Refer to the "ignore" case in SR Receiver side protocol;
  - Give a limited "live" time to every packet in the network;

The tricks so far

Mechanism	Use, Comments
Checksum	To detect bit errors
Timer	To timeout/retransmit a packet (lost/premature packet)
Sequence number	To detect a lost packet (gap in seq#) To detect a duplicate packet (duplicate seq#)
Acknowledgement	To notify successful reception (individual/cumulative)
Negative ACK	To notify unsuccessful reception
Window, pipelining	To exercise flow control, congestion control and improve sender utilisation

Summary

- **Reliable transfer protocols**
  - rdt1.0: for a reliable channel
  - rdt2.0: for a channel with bit errors
  - rdt2.1: sender, handles garbled ACK/NAKs
  - rdt2.2: a NAK-free protocol
  - rdt3.0: channels with errors *and* loss
  - Pipelined protocols
    - o Go-back-N
    - o Selective repeat

References

- [KR3] James F. Kurose, Keith W. Ross, *Computer networking: a top-down approach featuring the Internet*, 3rd edition.

- [PD5] Larry L. Peterson, Bruce S. Davie, *Computer networks: a systems approach*, 5th edition

- [TW5] Andrew S. Tanenbaum, David J. Wetherall, *Computer network*, 5th edition

- [LHBi]Y-D. Lin, R-H. Hwang, F. Baker, *Computer network: an open source approach*, International edition

- [JT] https://www.electronicdesign.com/test-measurement/what-s-difference-between-jitter-and-noise

Acknowledgements

- All slides are developed based on slides from the following two sources:
  - Dr DongSeong Kim's slides for COSC264, University of Canterbury;
  - Prof Aleksandar Kuzmanovic's lecture notes for CS340,Northwestern University, https://users.cs.northwestern.edu/~akuzma/classes/CS340-w05/lecture_notes.htm