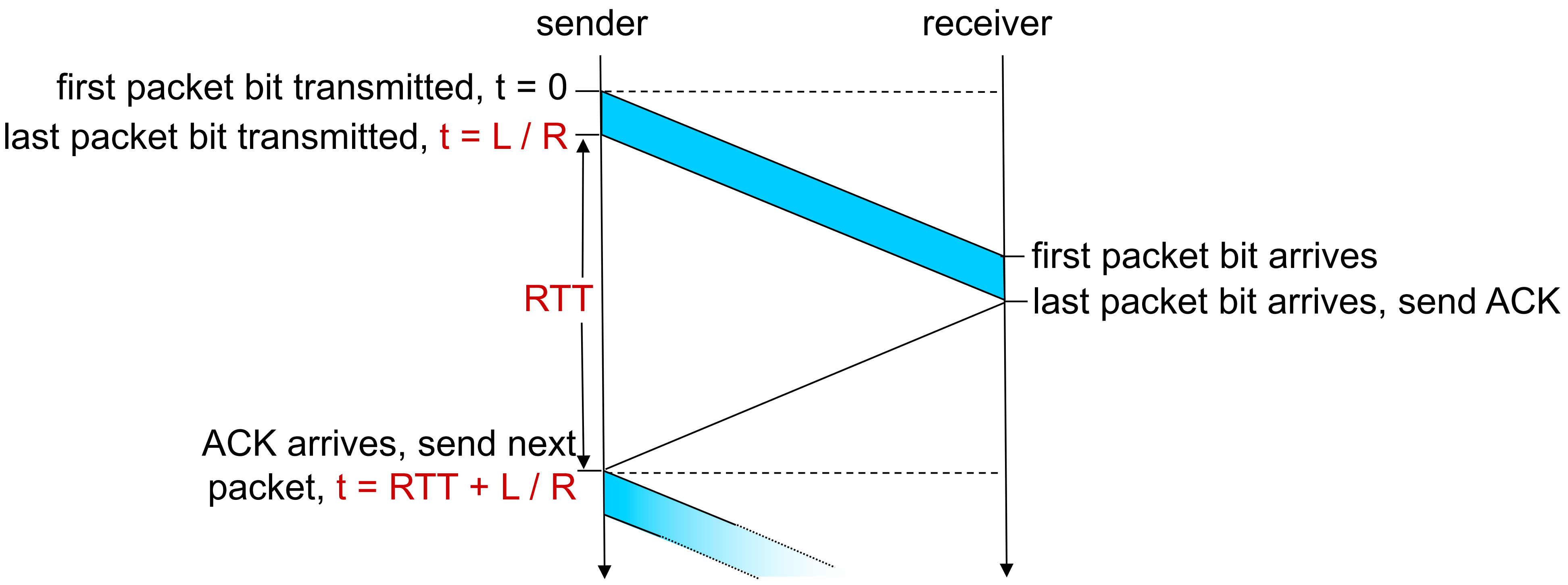
rdt3.0: stop-and-wait operation


sender
receiver
first packet bit transmitted, t = 0
last packet bit transmitted, t = L / R
RTT
first packet bit arrives
last packet bit arrives, send ACK
ACK arrives, send next
packet, t = RTT + L / R

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

rdt3.0: stop-and-wait performance

- rdt3.0 is correct, but performance stinks
- e.g.: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

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

- U $_{\text{sender}}$: *utilization* (or "throughput" in your HW1): fraction of time sender busy sending

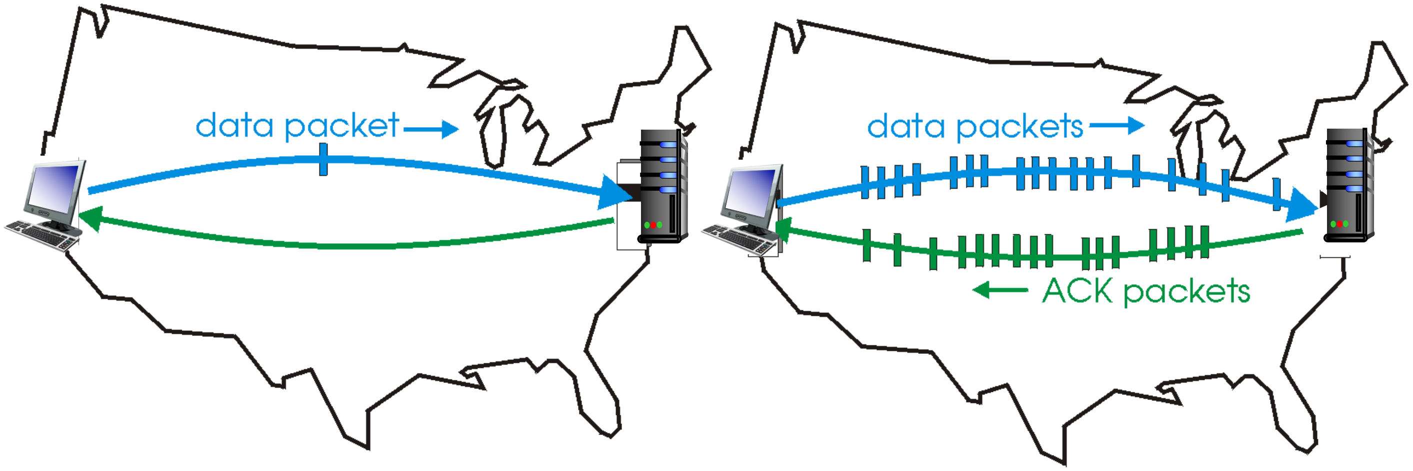
$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

- if RTT=30 msec, 1KB pkt every 30 msec: 33kB/sec throughput over 1 Gbps link
  - Example of a (bad) network protocol limits use of physical resources (fast link)!

Pipelined protocols

pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged pkts

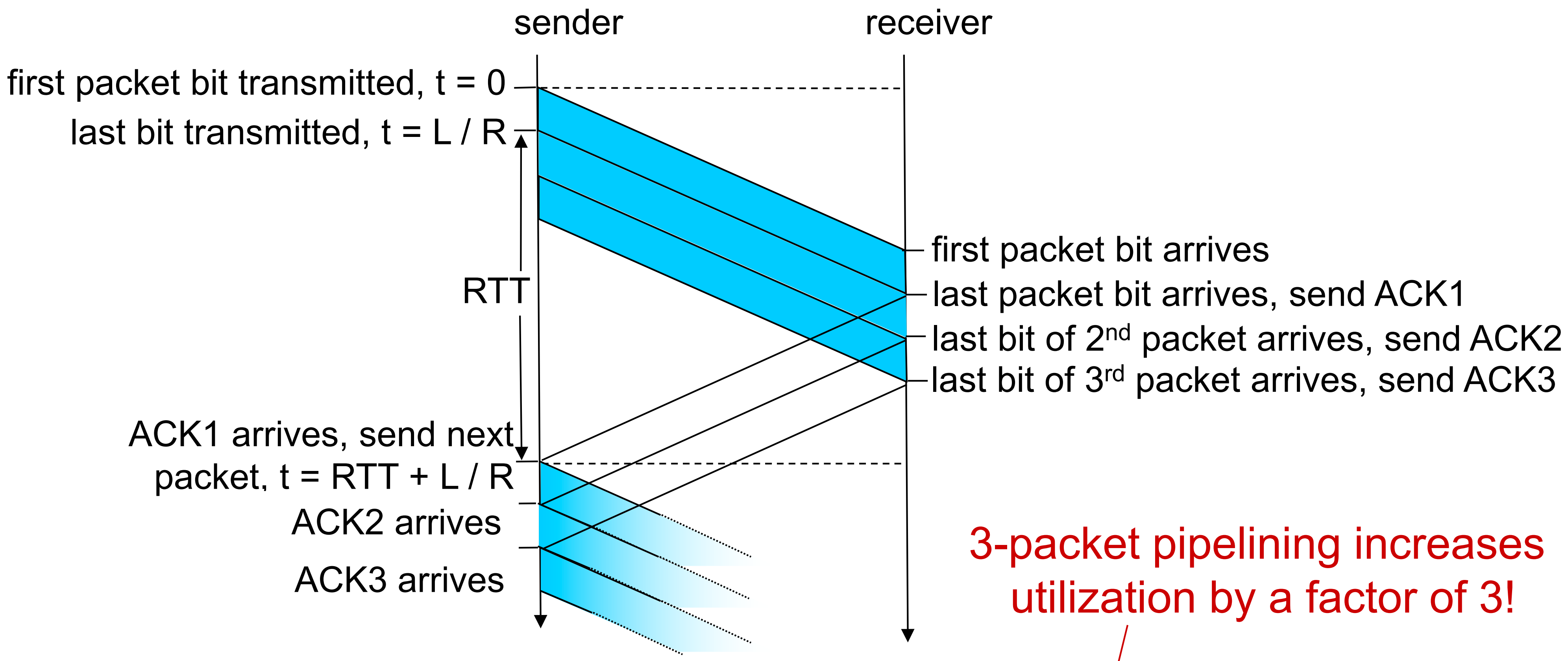
- range of sequence numbers must be increased
- buffering at sender and/or receiver
- [note: different than pipelining in HW1]



(a) a stop-and-wait protocol in operation (b) a pipelined protocol in operation

- two generic forms of pipelined protocols:
  - *go-Back-N*
  - *selective repeat*

Pipelining: increased utilization


sender
receiver
first packet bit transmitted, t = 0
last bit transmitted, t = L / R
RTT
first packet bit arrives
last packet bit arrives, send ACK1
last bit of 2nd packet arrives, send ACK2
last bit of 3rd packet arrives, send ACK3
ACK1 arrives, send next packet, t = RTT + L / R
ACK2 arrives
ACK3 arrives

3-packet pipelining increases utilization by a factor of 3!

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

Q: What is the best choice of the "window" of pipelined messages?

A: RTT/L/R+1 keeps the channel always busy

Pipelined protocols: overview

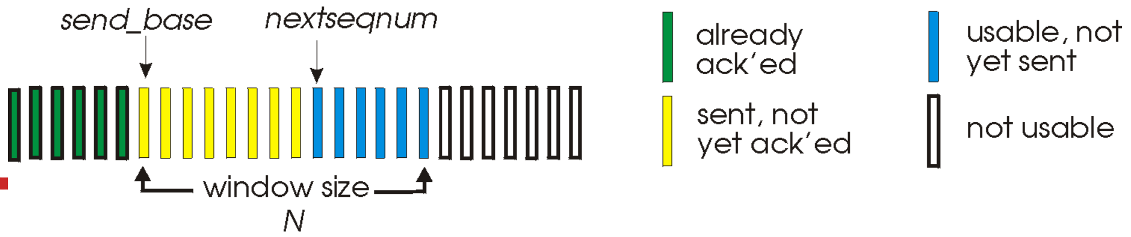
Go-back-N:

- sender can have up to N unacked packets in pipeline (window)
- receiver only sends *cumulative ack*
  - doesn't ack packet if there is a gap
- sender has timer for oldest unacked packet
  - when timer expires, retransmit *all* unacked packets

Selective Repeat:

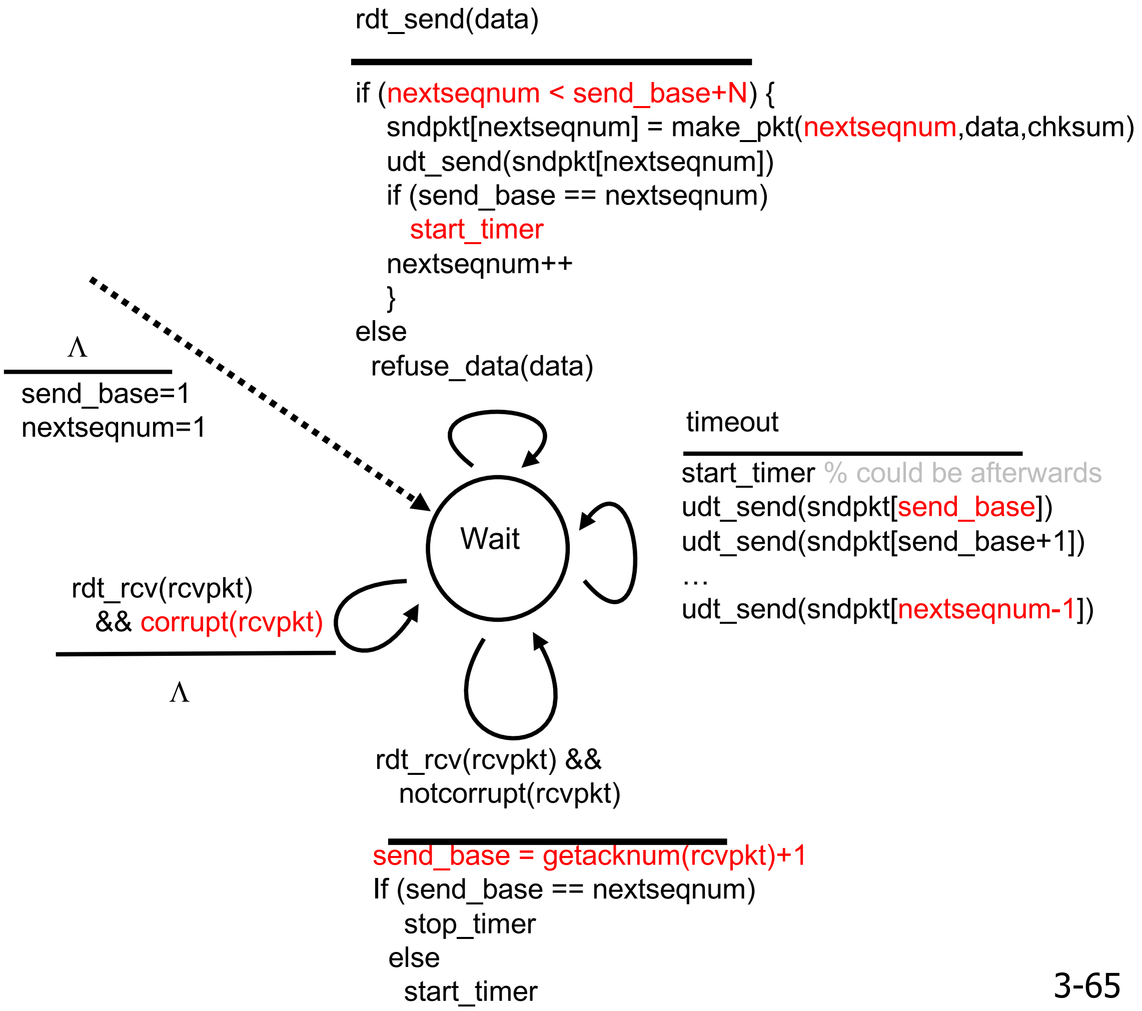
- sender can have up to N unack'ed packets in pipeline (window)
- rcvr sends *individual ack* for each packet
- sender maintains timer for each unacked packet
  - when timer expires, retransmit only that unacked packet

GBN Sender



- k-bit sequence # in pkt header
- sliding window of up to N, consecutive unack' ed pkts allowed

- Send_base:
  - oldest in-flight packet (n)
  - nextsequm: next to send
- ACK(m): ACKs all pkts up to, including seq # m - *"cumulative ACK"*
  - may receive duplicate ACKs (see receiver)
- *timeout(n)*: keep timer for oldest unack-ed packet (send_base); retransmit all sent but unacked pkts in window – i.e., all yellow packets



Λ

```
send_base=1
nextseqnum=1
```

rdt_send(data)

```
if (nextseqnum < send_base+N) {
    sndpkt[nextseqnum] = make_pkt(nextseqnum,data,chksum)
    udt_send(sndpkt[nextseqnum])
    if (send_base == nextseqnum)
      start_timer
    nextseqnum++
    }
else
  refuse_data(data)
```

timeout

```
start_timer % could be afterwards
udt_send(sndpkt[send_base])
udt_send(sndpkt[send_base+1])
…
udt_send(sndpkt[nextseqnum-1])
```

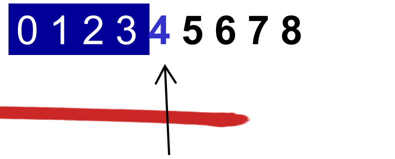
rdt_rcv(rcvpkt)
&& corrupt(rcvpkt)

Λ

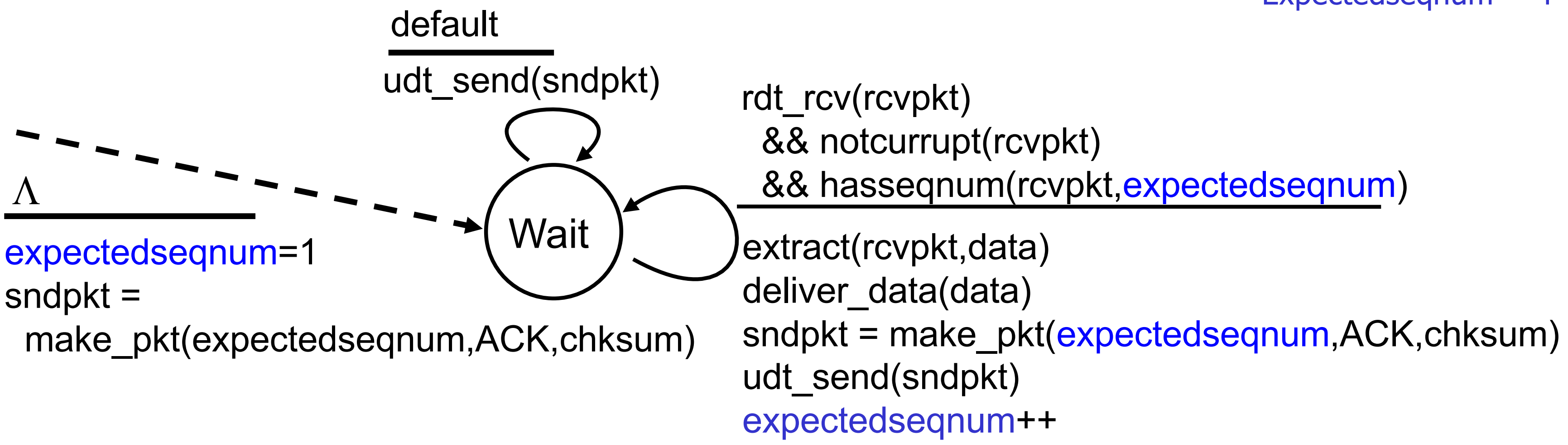
rdt_rcv(rcvpkt) &&
notcorrupt(rcvpkt)

```
send_base = getacknum(rcvpkt)+1
If (send_base == nextseqnum)
   stop_timer
 else
   start_timer
```

GBN Receiver



Expectedseqnum = 4



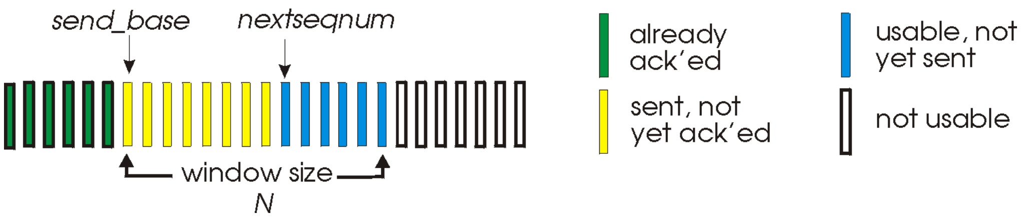
default
udt_send(sndpkt)

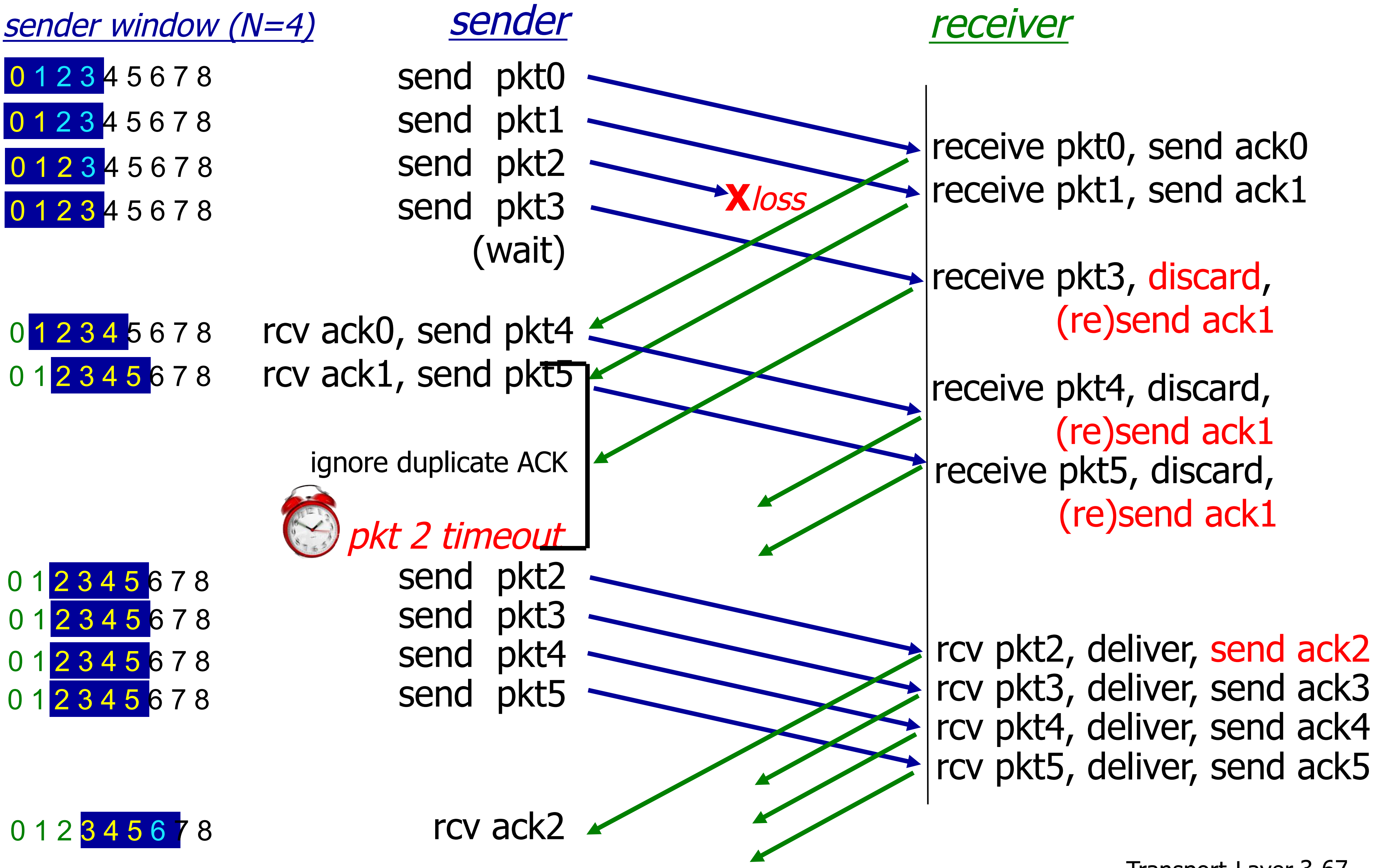
Λ
expectedseqnum=1
sndpkt =
  make_pkt(expectedseqnum,ACK,chksum)

rdt_rcv(rcvpkt)
  && notcurrupt(rcvpkt)
  && hasseqnum(rcvpkt,expectedseqnum)
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(expectedseqnum,ACK,chksum)
udt_send(sndpkt)
expectedseqnum++

- Simple state: need only remember **expectedseqnum**
  - This is the next seq (in order) we expect to receive
  - Everything before, is already received
- ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #
  - may generate duplicate ACKs
- If it receives out-of-order pkt:
  - Discard, *no receiver buffering!*
  - re-ACK pkt with highest in-order seq #

GBN in action





sender window (N=4)

sender

receiver

0 1 2 3 4 5 6 7 8 send pkt0
0 1 2 3 4 5 6 7 8 send pkt1
0 1 2 3 4 5 6 7 8 send pkt2
0 1 2 3 4 5 6 7 8 send pkt3
(wait)

X loss

receive pkt0, send ack0
receive pkt1, send ack1

receive pkt3, discard,
(re)send ack1

0 1 2 3 4 5 6 7 8 rcv ack0, send pkt4
0 1 2 3 4 5 6 7 8 rcv ack1, send pkt5

receive pkt4, discard,
(re)send ack1
receive pkt5, discard,
(re)send ack1

ignore duplicate ACK

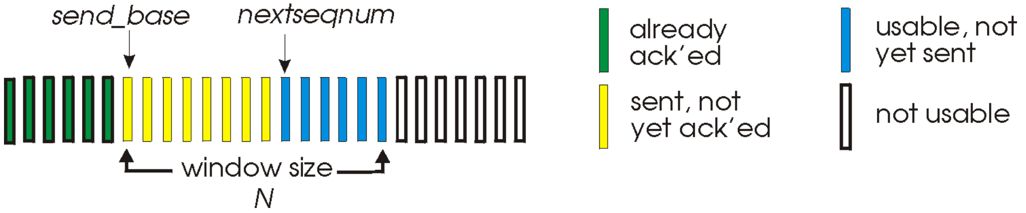
pkt 2 timeout

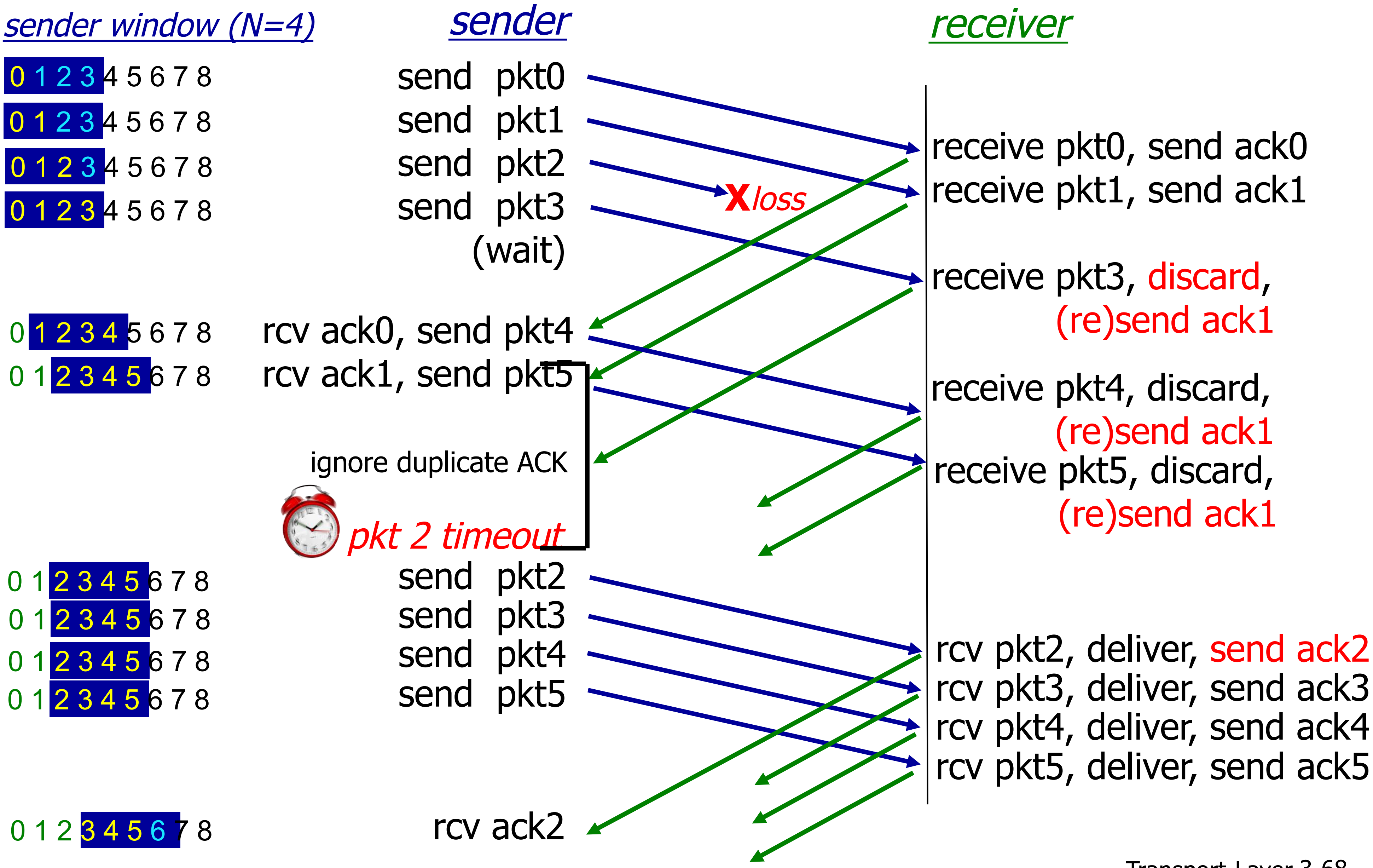
0 1 2 3 4 5 6 7 8 send pkt2
0 1 2 3 4 5 6 7 8 send pkt3
0 1 2 3 4 5 6 7 8 send pkt4
0 1 2 3 4 5 6 7 8 send pkt5

rcv pkt2, deliver, send ack2
rcv pkt3, deliver, send ack3
rcv pkt4, deliver, send ack4
rcv pkt5, deliver, send ack5

0 1 2 3 4 5 6 7 8 rcv ack2

GBN in action





sender window (N=4)

0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

sender

send pkt0
send pkt1
send pkt2
send pkt3
(wait)

rcv ack0, send pkt4
rcv ack1, send pkt5

ignore duplicate ACK

pkt 2 timeout

send pkt2
send pkt3
send pkt4
send pkt5

rcv ack2

X *loss*

receiver

receive pkt0, send ack0
receive pkt1, send ack1

receive pkt3, discard,
(re)send ack1

receive pkt4, discard,
(re)send ack1
receive pkt5, discard,
(re)send ack1

rcv pkt2, deliver, send ack2
rcv pkt3, deliver, send ack3
rcv pkt4, deliver, send ack4
rcv pkt5, deliver, send ack5

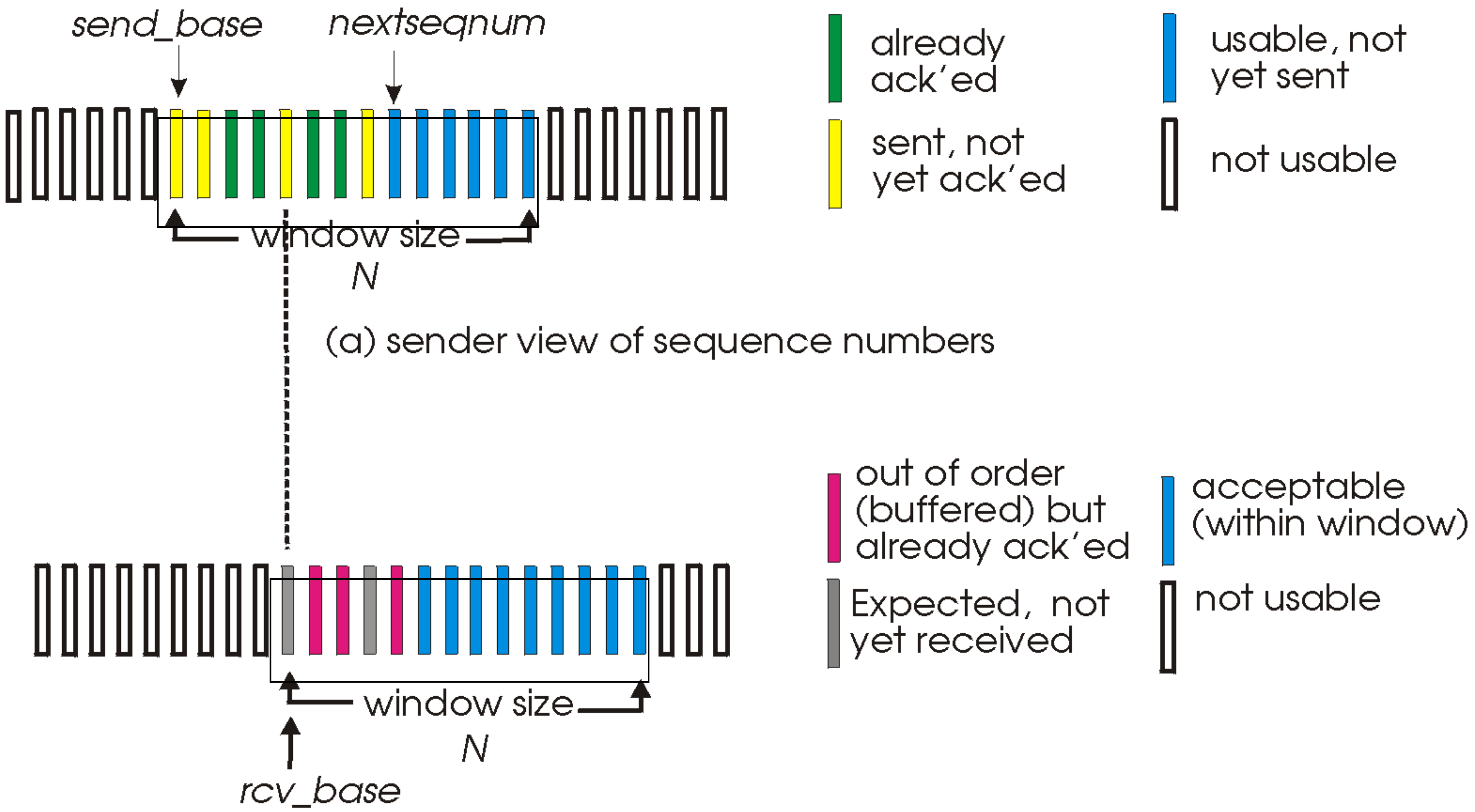
Selective Repeat (SR)

- Receiver *individually* acknowledges all correctly received pkts
  - whether they are received in-order or out-of-order
  - buffers pkts, as needed, for eventual in-order delivery to upper layer

- Sender only retransmits un-ACKed pkts
  - sender maintains timer for each unACKed pkt

- Both sender and receiver maintain windows
  - Sender window
    - *N* consecutive seq #'s
    - limits seq #s of sent, unACKed pkts
  - Receiver window
    - *N* consecutive seq #'s
    - Limits seq#s of buffered out-of-order packets

Selective Repeat: sender, receiver windows



(a) sender view of sequence numbers

(b) receiver view of sequence numbers

Selective Repeat (FSM not shown)

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]:

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

receiver

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

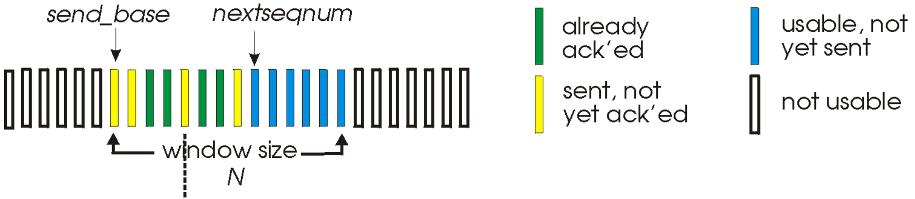
- send ACK(n)
- out-of-order: buffer
- in-order: deliver (also deliver all buffered, in-order pkts), advance window to next not-yet-received pkt

pkt n in [rcvbase-N,rcvbase-1] % duplicate

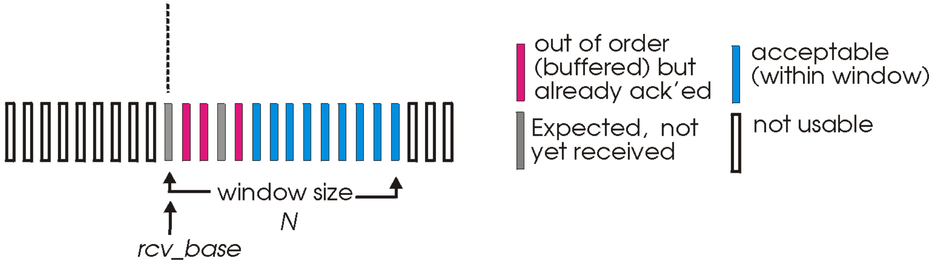
- ACK(n) [Why not ignore?]

otherwise:

- ignore

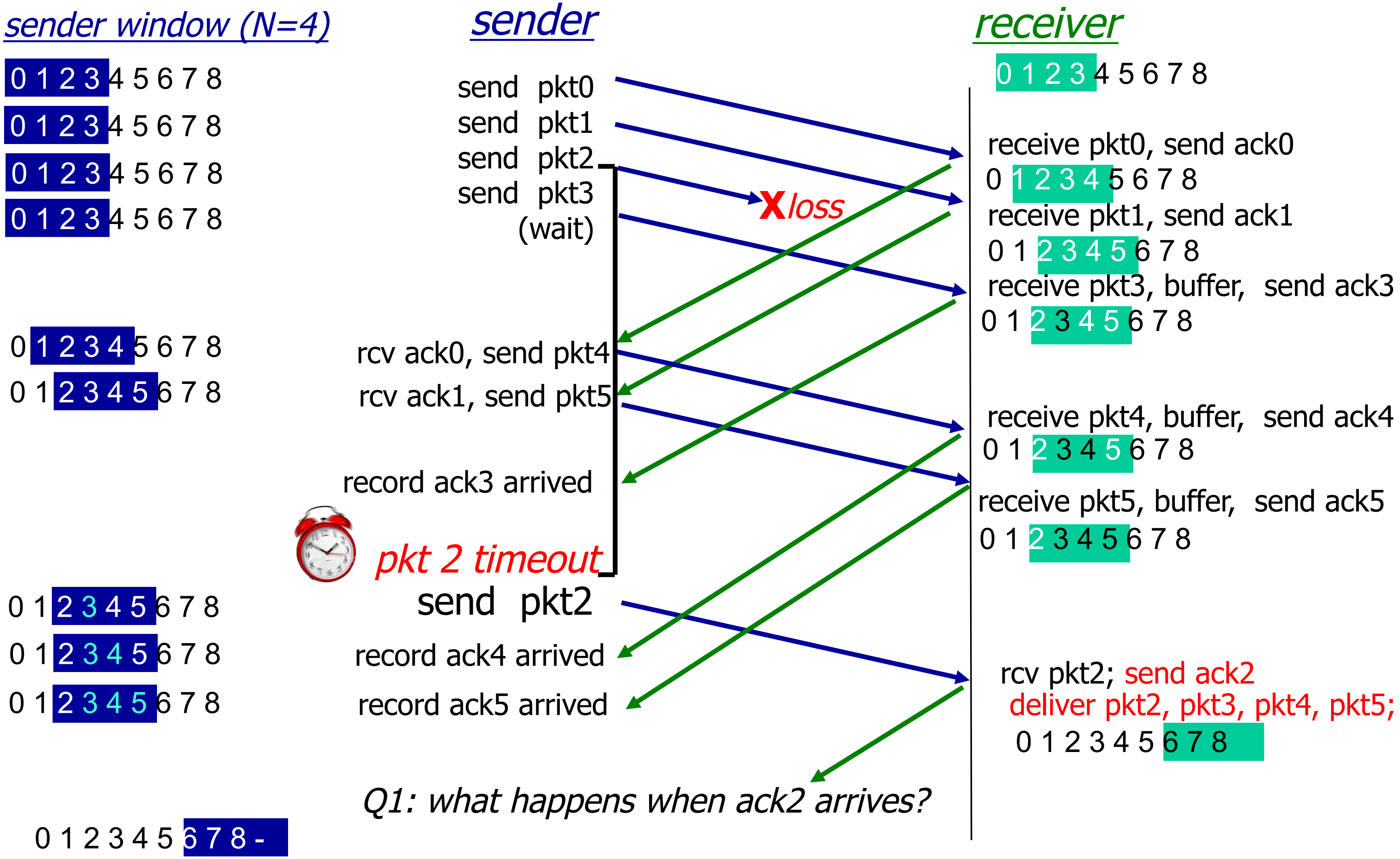


(a) sender view of sequence numbers


out of order (buffered) but already ack'ed
acceptable (within window)
Expected, not yet received
not usable
window size
N
rcv_base

(b) receiver view of sequence numbers

Selective repeat in action



sender window (N=4)

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8 -

sender

send pkt0
send pkt1
send pkt2
send pkt3
(wait)

X *loss*

rcv ack0, send pkt4
rcv ack1, send pkt5

record ack3 arrived

pkt 2 timeout

send pkt2

record ack4 arrived

record ack5 arrived

receiver

0 1 2 3 4 5 6 7 8

receive pkt0, send ack0
0 1 2 3 4 5 6 7 8

receive pkt1, send ack1
0 1 2 3 4 5 6 7 8

receive pkt3, buffer, send ack3
0 1 2 3 4 5 6 7 8

receive pkt4, buffer, send ack4
0 1 2 3 4 5 6 7 8

receive pkt5, buffer, send ack5
0 1 2 3 4 5 6 7 8

rcv pkt2; send ack2
deliver pkt2, pkt3, pkt4, pkt5;
0 1 2 3 4 5 6 7 8

Q1: what happens when ack2 arrives?

Lack of sync + finite seq#: ambiguity

Example:

- seq #' s: 0, 1, 2, 3
- window size=3

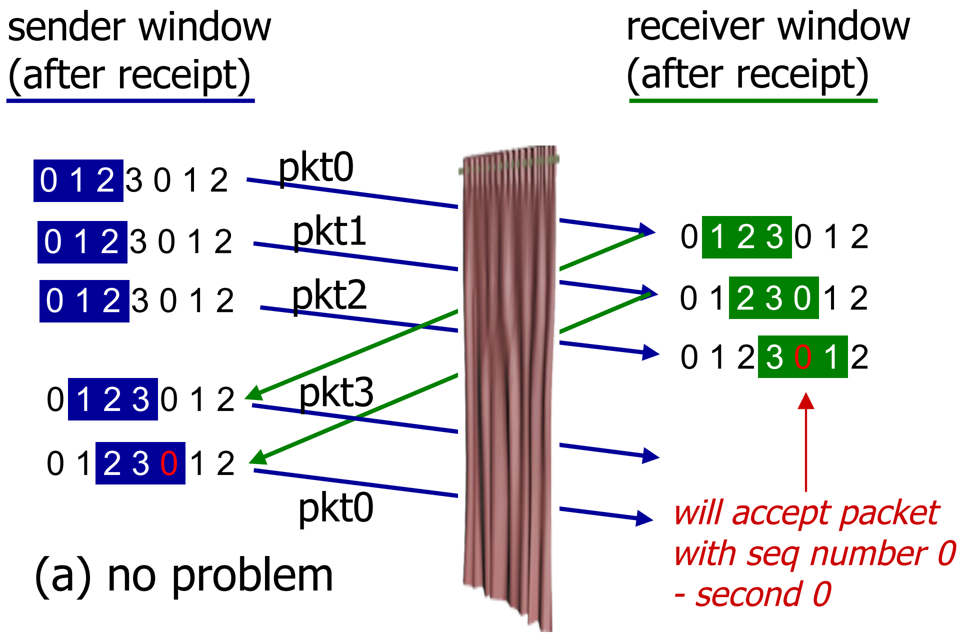
Problem: receiver sees no difference in 2 scenarios!

- new data arrive in (a)
- duplicate data accepted as new in (b)

Q1: would the problem occur if seq# up to 5?

Q2: what relationship between seq # size (SN) and window size (N) to avoid problem?

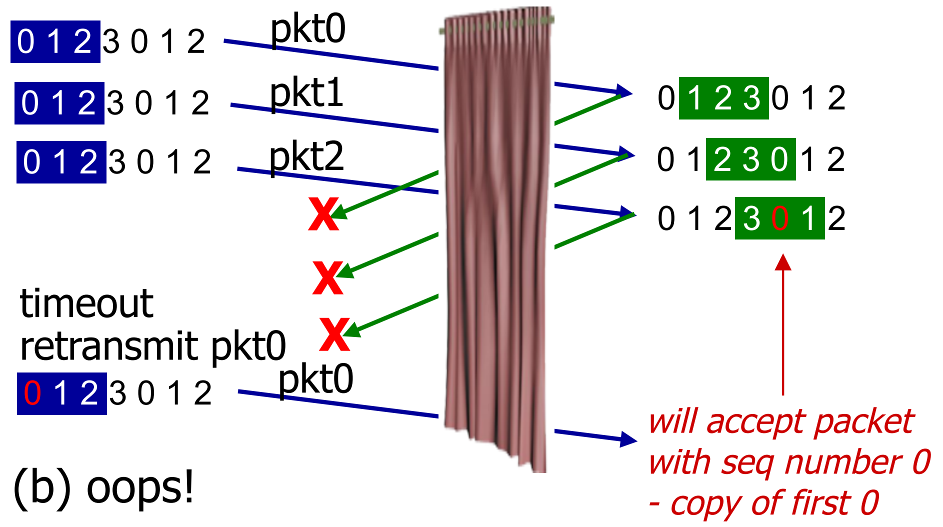
A2: SN>=2N because sender and receiver window must have overlap of at least 1



(a) no problem

will accept packet with seq number 0 - second 0

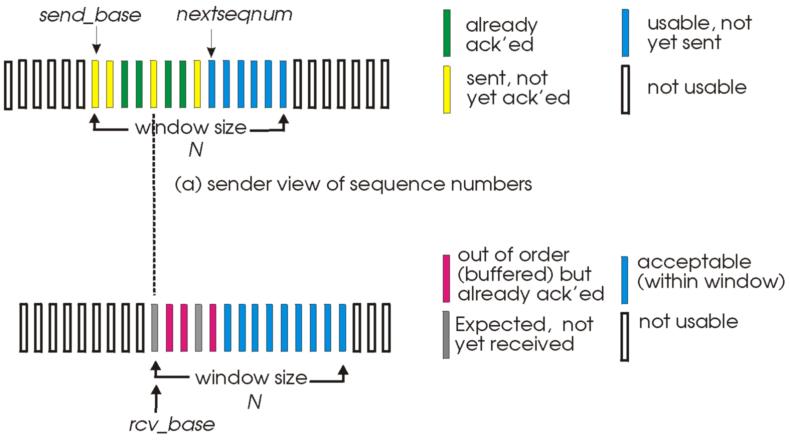
receiver can't see sender side.
receiver behavior identical in both cases!
something's (very) wrong!



(b) oops!

will accept packet with seq number 0 - copy of first 0

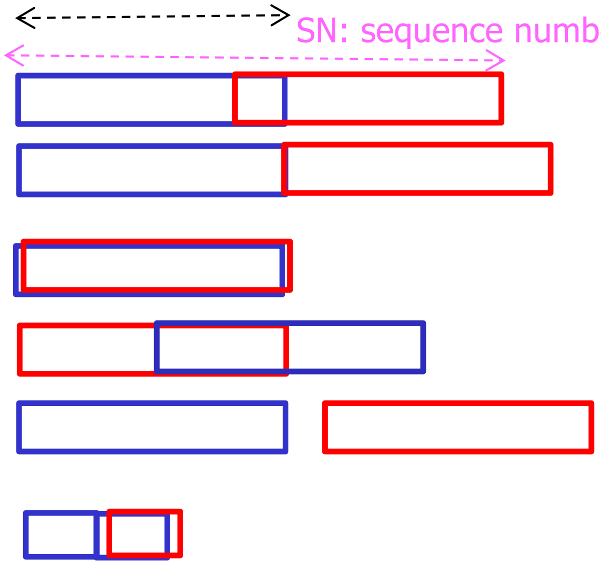
SR: sender, receiver windows & Seq No



(a) sender view of sequence numbers

(b) receiver view of sequence numbers

N= Sender window
= Receiver window



SN: sequence numbers for packets and acks

Sender & receiver windows overlap. At most consecutive.
SN>=2N →seqnums for packets and acks are unique

Sender & receiver windows in perfect sync

Impossible

Impossible

Special Case: Stop and Wait: Sender W=2, Receiver W=1

Go-back-N vs SR: Mechanisms

Go-back-N:

- Sender can have up to N unack'ed packets in pipeline
  - sliding window
- Rcvr sends *cumulative* ack for last in-order packet
  - maintains expectedseqnum
  - doesn't accept or ack out-of-order packet
- Sender maintains timer for oldest unacked packet
  - if timer expires, retransmit all unack'ed packets

Selective Repeat:

- Sender can have up to N unack'ed packets in pipeline
  - sliding window
- Rcvr sends *individual ack* for each packet
  - maintains Rcvr window
  - buffers and acks all packets within Rcvr window
- Sender maintains timer for each unacked packet
  - when timer expires, retransmit only that one unack'ed packet

GBN vs SR: Performance

- Compared to Stop-and Wait
  - They both fill the pipeline

- Loss rate
  - Light loss:
    - SR: selectively retransmits what is needed
    - GBN: a single packet lost causes unnecessary retransmission of all packets in the window
  - Heavy loss
    - GBN: ok

- Complexity:
  - GBN is simpler – less state