# Communication Networks (0368-3030) / Spring 2011

The Blavatnik School of Computer Science,
Tel-Aviv University

Allon Wagner

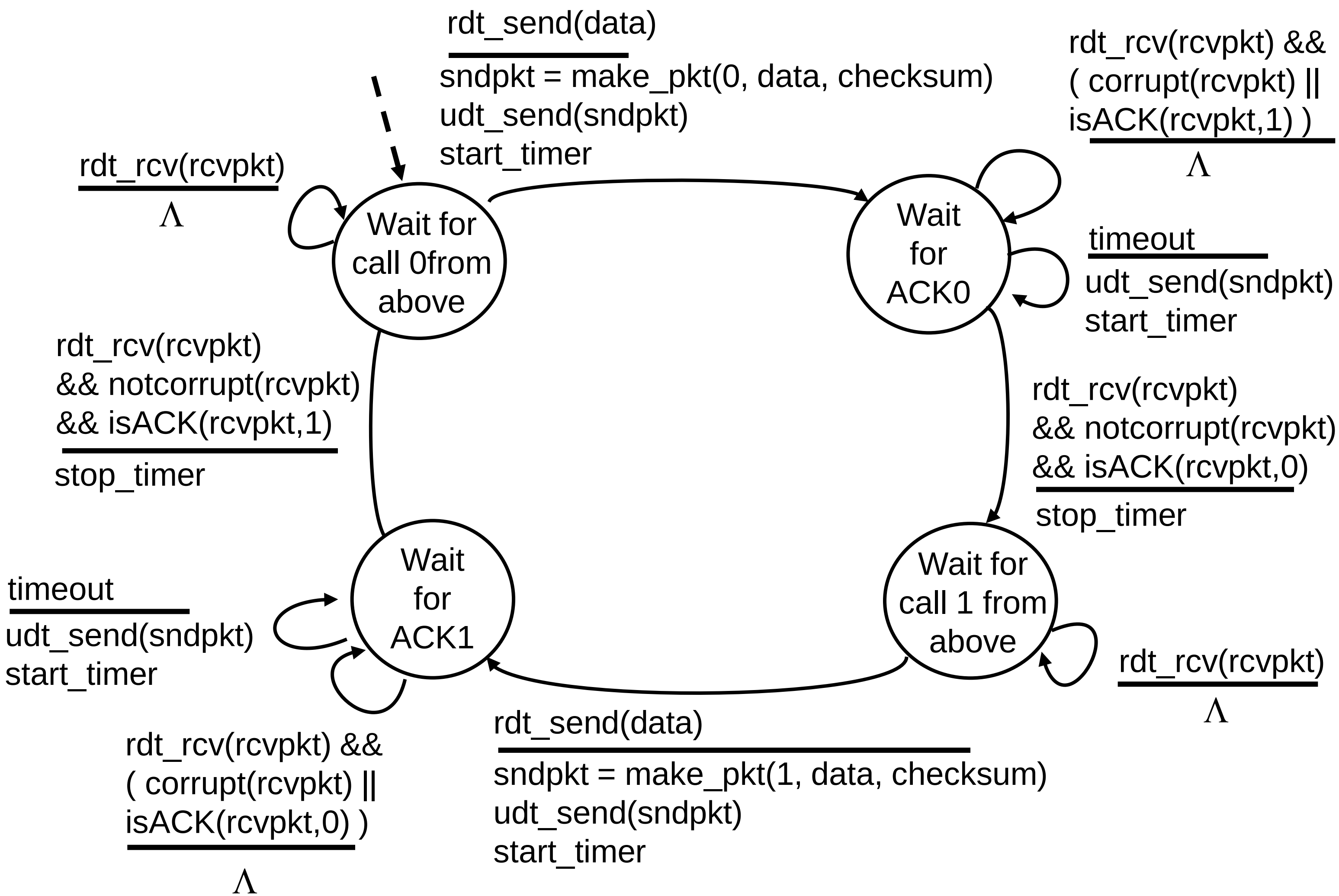
# Reliable Data Transfer

Kurose & Ross, Chapter 3.4 (5th ed.)

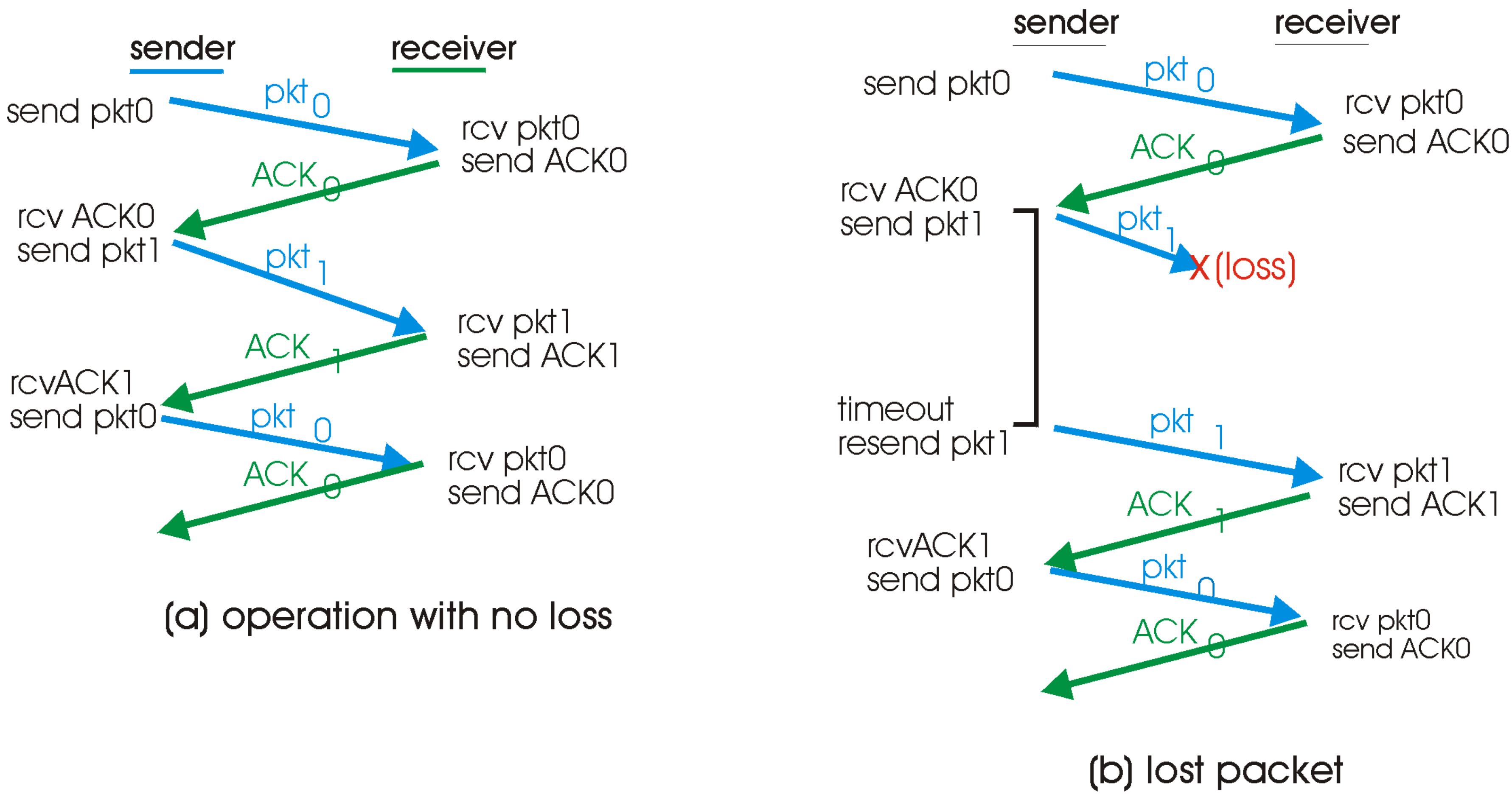
Many slides adapted from:

J. Kurose & K. Ross \
Computer Networking: A Top Down Approach (5th ed.)
Addison-Wesley, April 2009.

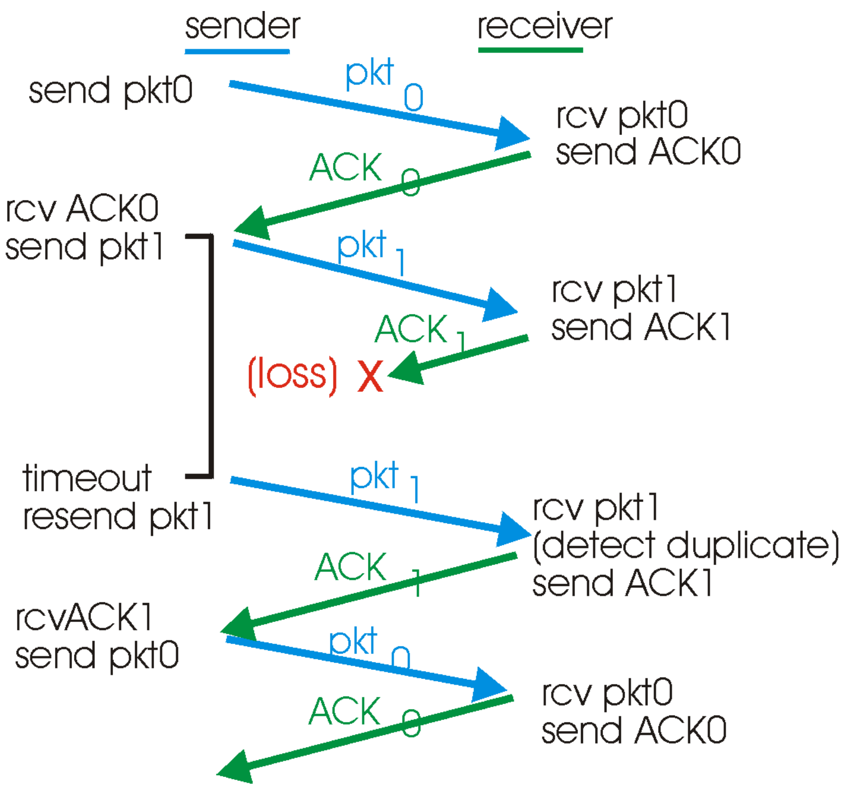
# rdt3.0 sender



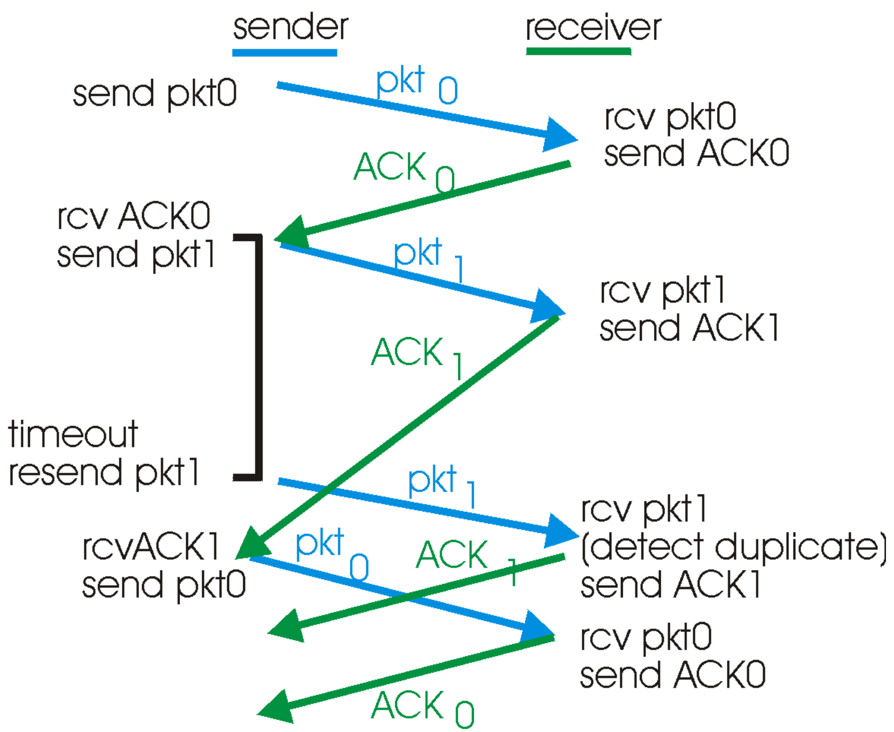
# rdt3.0 in action



# rdt3.0 in action



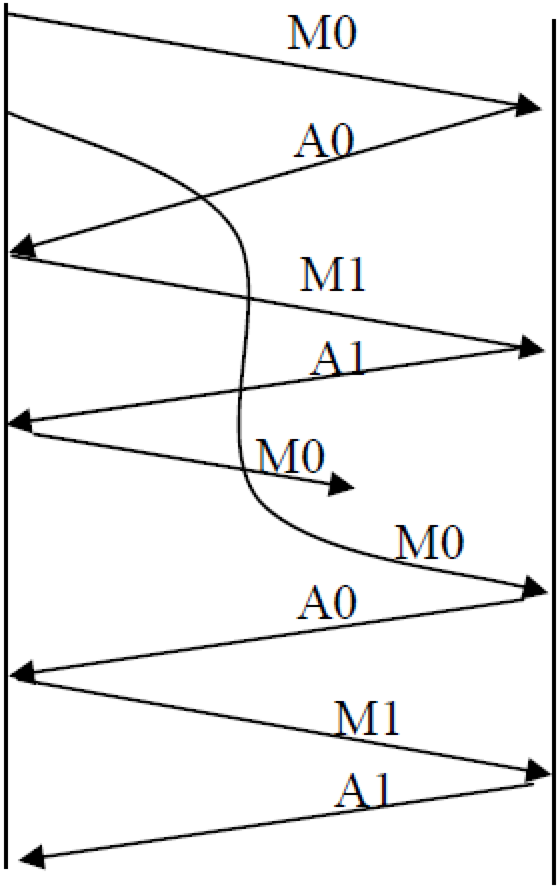

(c) lost ACK




(d) premature timeout

# Exercise (Kurose & Ross, 5th ed.)

- rdt 3.0 is correct only under a FIFO channel assumption.
  - Correct = guarantees reliable transmission. Data sent by sender is exactly the data reconstructed in the receiver side.
- Show a case where a non-FIFO channel (i.e., one that can cause packet reordering) causes rdt 3.0 to deliver incorrect data.



# Exercise (Kurose & Ross, 5th ed.)

- The sender of rdt 3.0 simply ignores all received packets that are either in error or have the wrong value in the acknum field of an ack packet.
- Suppose that in such circumstances, rdt 3.0 were simply to transmit the current data packet.
- Would the protocol still work?
- Would it be more or less efficient than before?

# Exercise (Kurose & Ross, 5th ed.)

- Would the protocol still work?
  - Yes. A retransmission is exactly what would happen if the ack was completely lost instead of garbled.
  - The receiver can't even distinguish between the two events.

# Exercise (Kurose & Ross, 5th ed.)

- Would it be more or less efficient than before?
  - Depends on the length of the sender timeout, compared to the expected channel delay.
  - If the timeout is very long, then the immediate retransmit can save us the long wait until the timeout expires.
  - However, premature timeouts can cause a pathologies.

# Exercise (Kurose & Ross, 5th ed.)

- Would it be more or less efficient than before?
  - In the original rdt 3.0, once an ack for a data packet is received, it can no longer cause retransmissions.
  - Assume the following scenario:
    - Packet 1 is sent.
    - Sender has a premature timeout. One extra copy of packet 1 is sent.
    - Receiver gets 2 copies and acks each of them. The 2nd ack is garbled.
    - This causes a retransmission of the current sender data packet (packet 2). Packet 2 was thus also sent twice.
    - The 2nd ack for packet 2 was garbled. Thus, packet 3 is also sent twice.
    - And so on…
  - Every data packet was sent twice even though no data packet was garbled and only one premature timeout occurred!
  - Original rdt 3.0 would have sent only packet 1 twice (due to the premature timeout).

# Performance of rdt3.0

- rdt3.0 works, but performance stinks
- ex: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

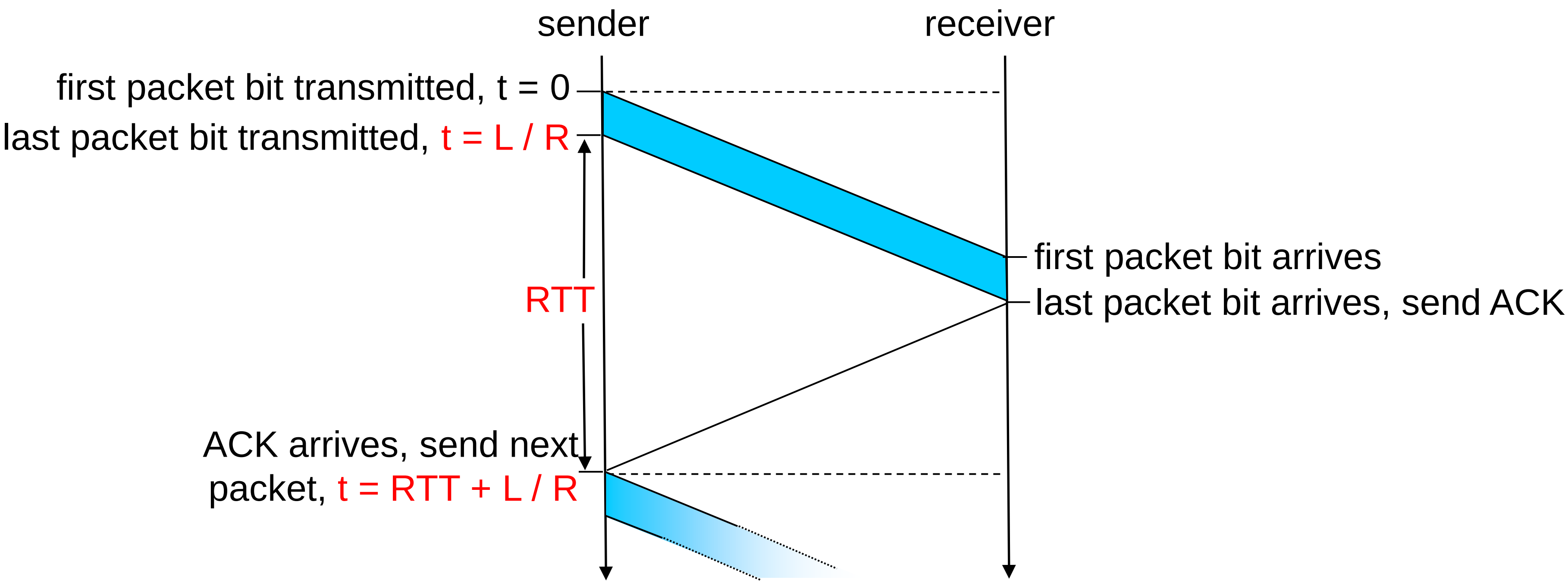
$$d_{trans} = \frac{L}{R} = \frac{8000\text{bits}}{10^9\text{bps}} = 8\text{microseconds}$$

  - U $_{sender}$: utilization – 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 thruput over 1 Gbps link
  - network protocol limits use of physical resources!

# rdt3.0: stop-and-wait operation

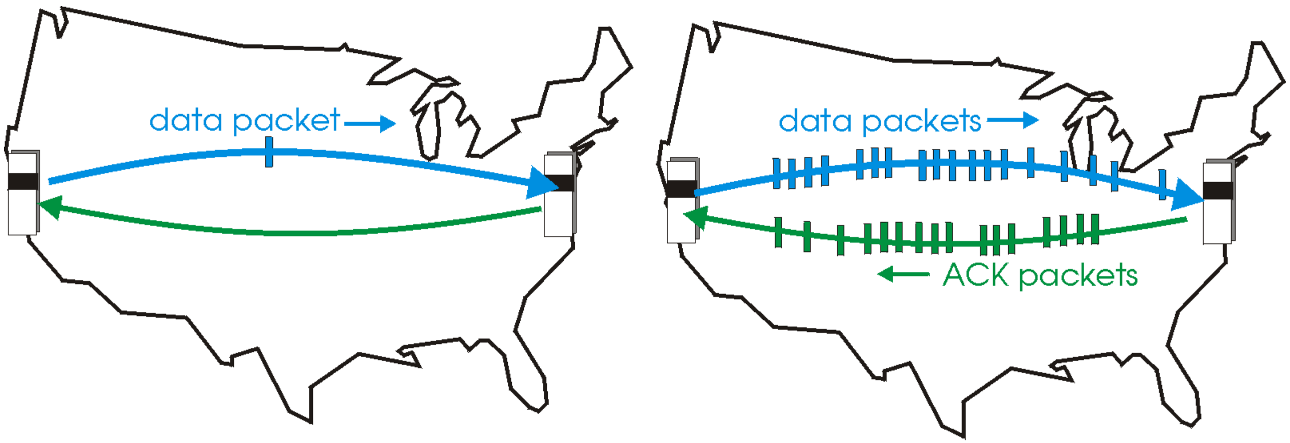



$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

# Pipelined protocols

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

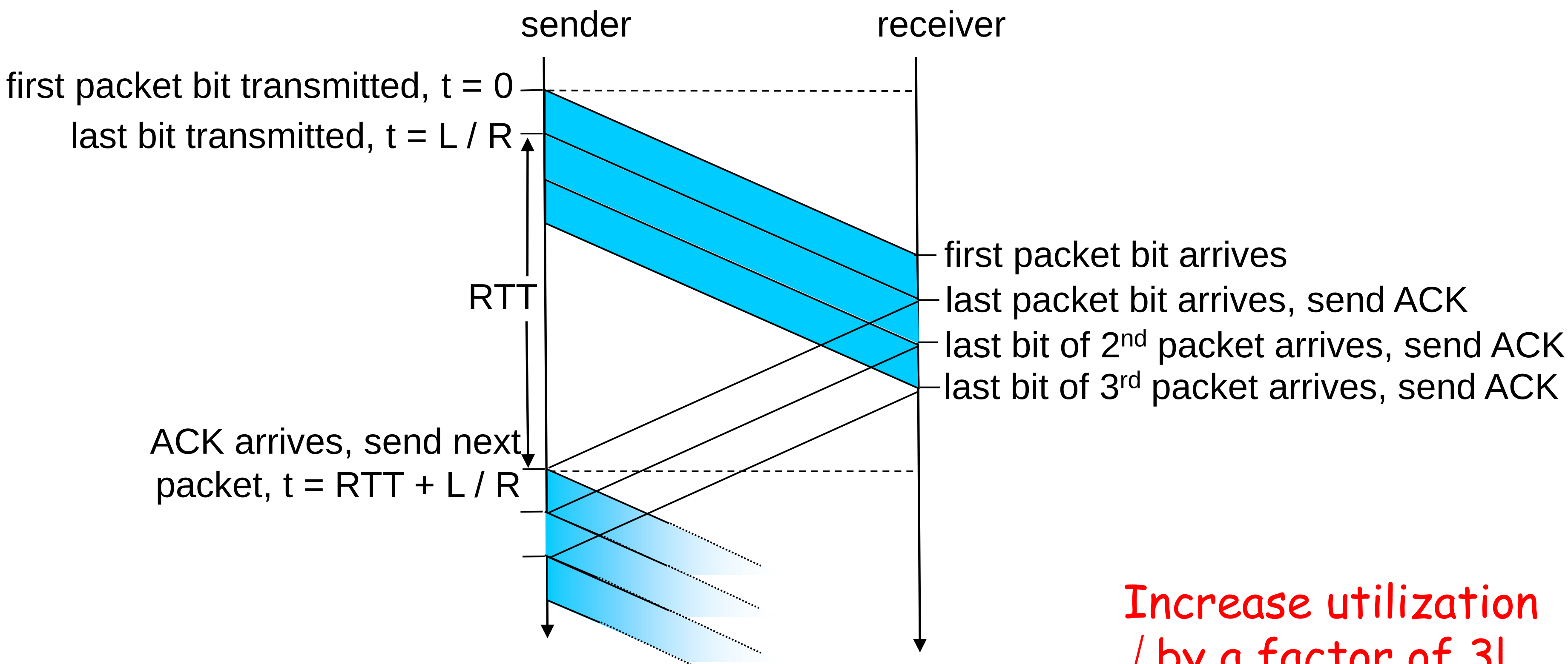
- range of sequence numbers must be increased
- buffering at sender and/or receiver




(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




Increase utilization by a factor of 3!

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

# Pipelined Protocols

**Go-back-N: big picture:**

- sender can have up to N unacked packets in pipeline
- rcvr only sends *cumulative* acks
  - doesn't ack packet if there's a gap
- sender has timer for oldest unacked packet
  - if timer expires, retransmit all unack'ed packets

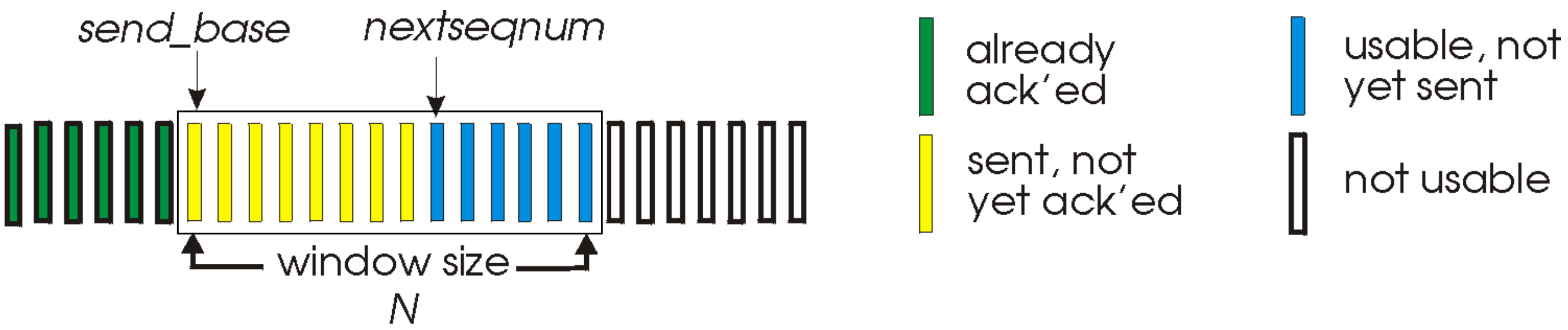
**Selective Repeat: big pic**

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

# Go-Back-N

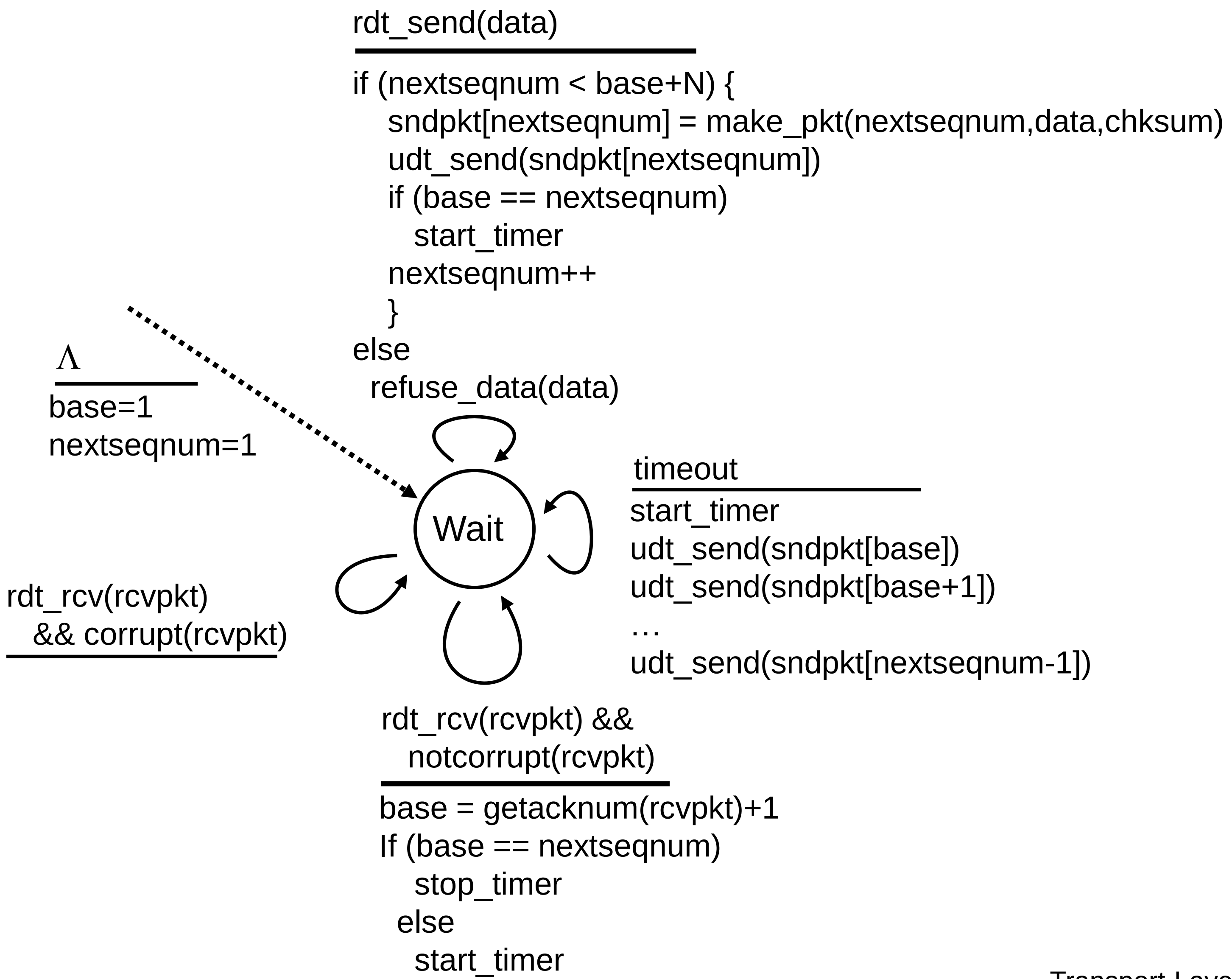
Sender:

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

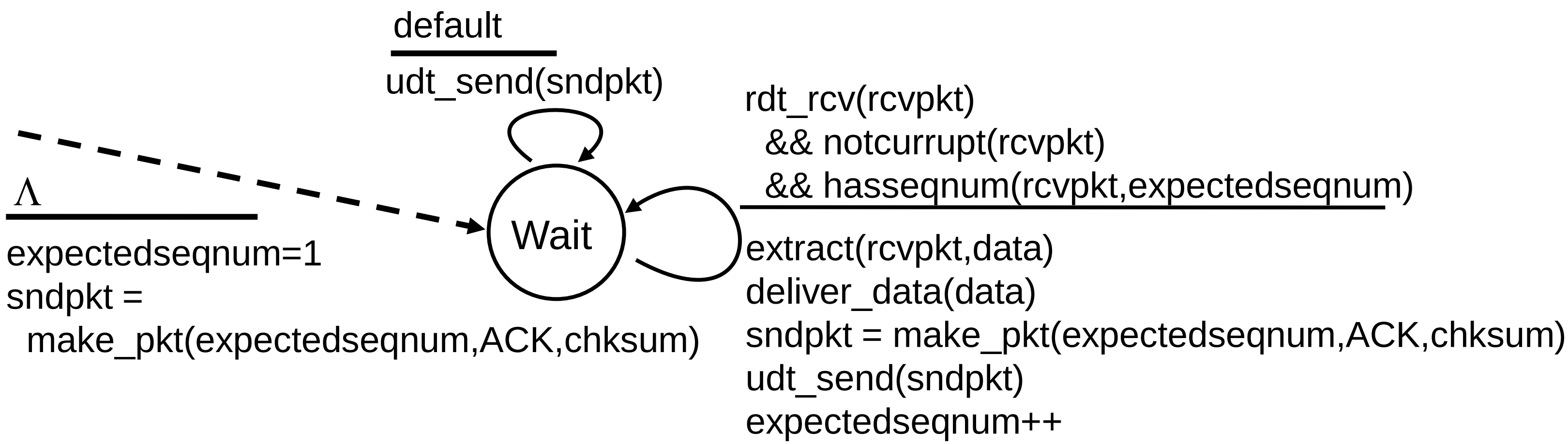



- ACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"
  - may receive duplicate ACKs (see receiver)
- timer for each in-flight pkt
- *timeout(n):* retransmit pkt n and all higher seq # pkts in window

# GBN: sender extended FSM



# GBN: receiver extended FSM

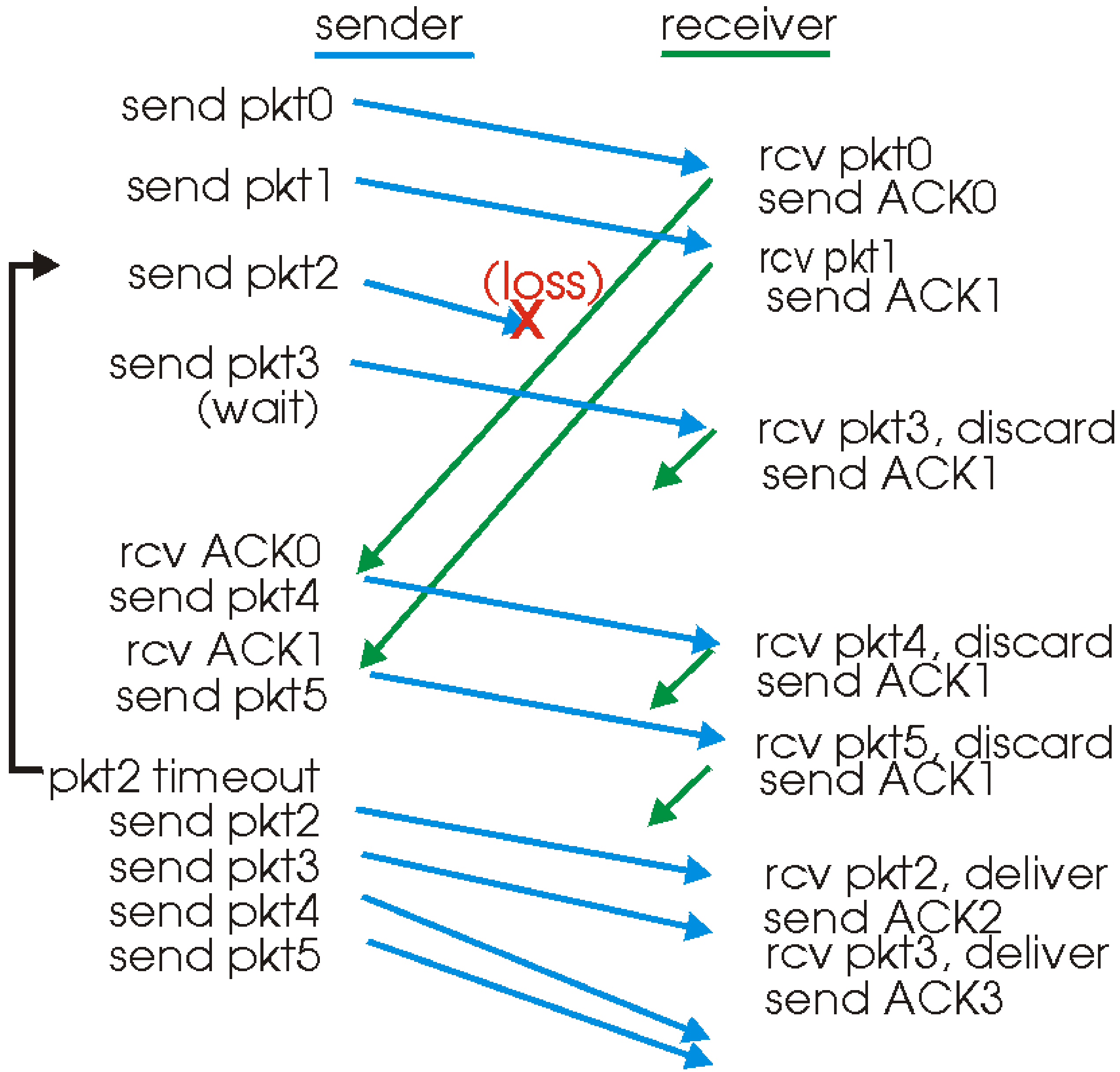



ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #

- may generate duplicate ACKs
- need only remember **expectedseqnum**

- out-of-order pkt:
  - discard (don't buffer) -> no receiver buffering!
  - Re-ACK pkt with highest in-order seq #

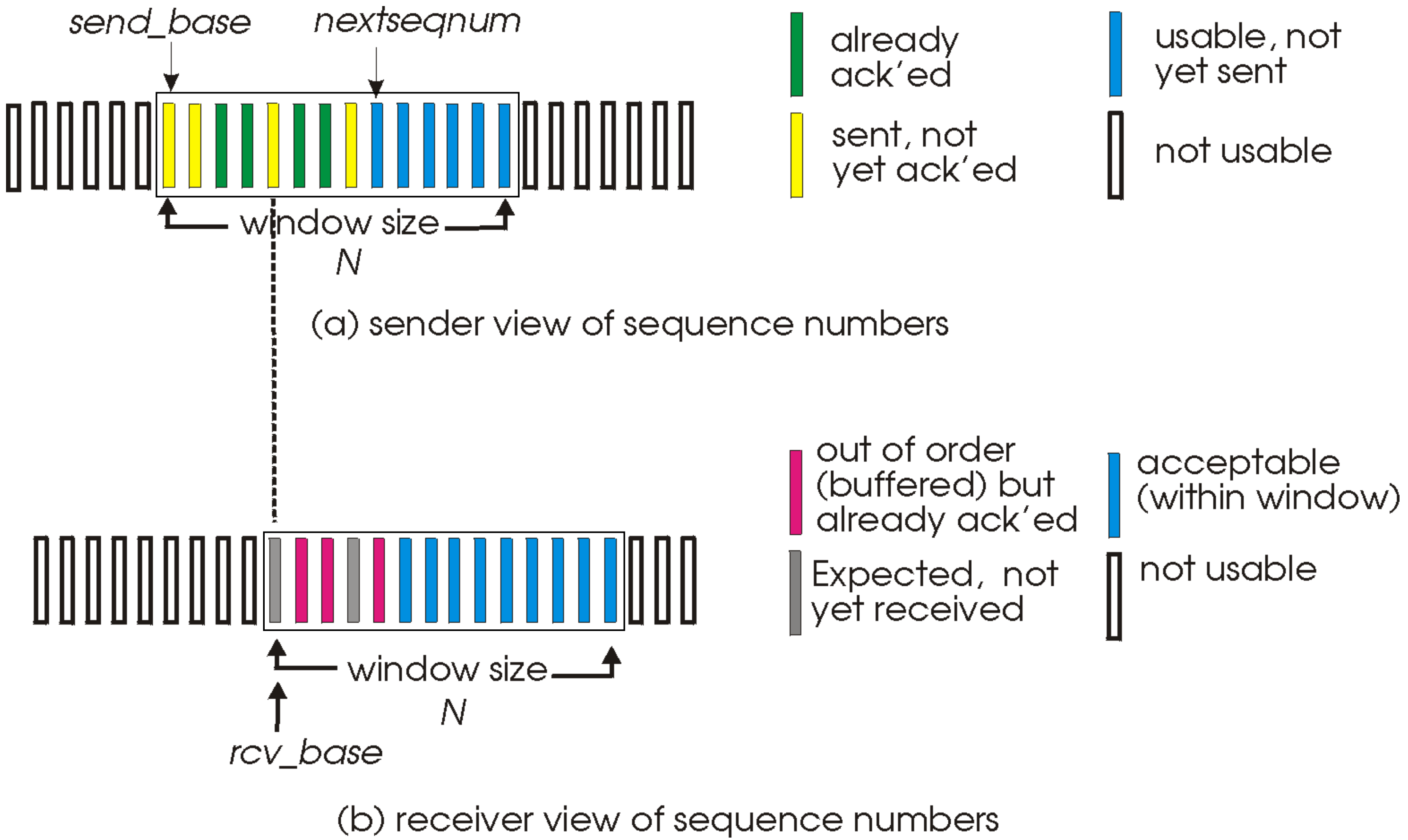
# GBN in action



# Selective Repeat

- 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, unACK'ed pkts

# Selective repeat: sender, receiver windows



# Selective repeat

**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]

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

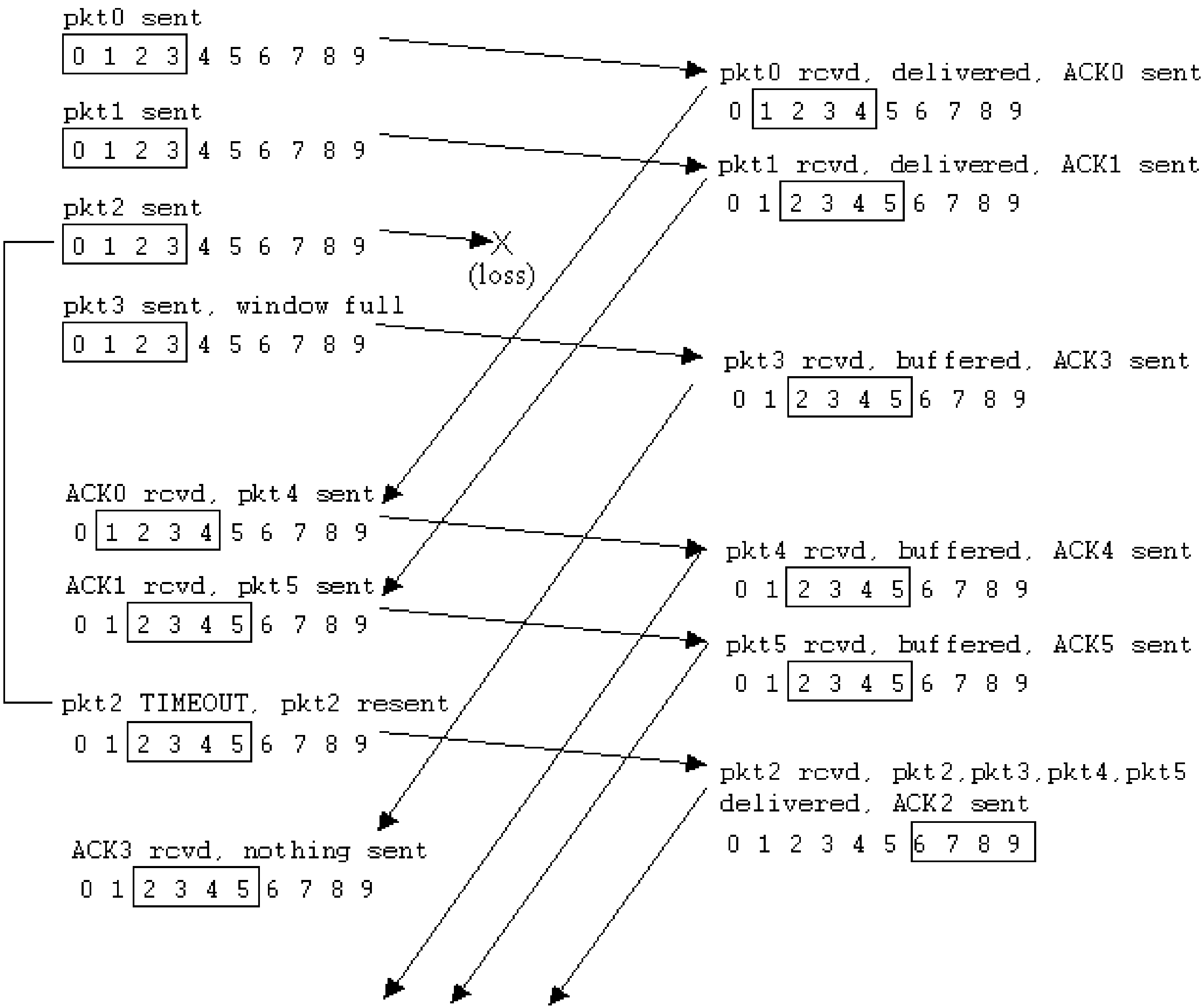
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignore

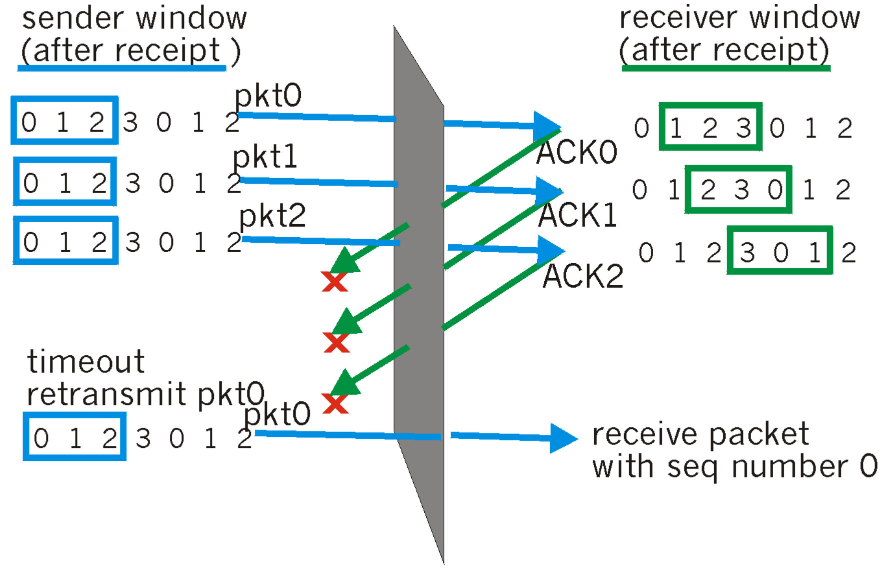
# Selective repeat in action



# Selective repeat: dilemma

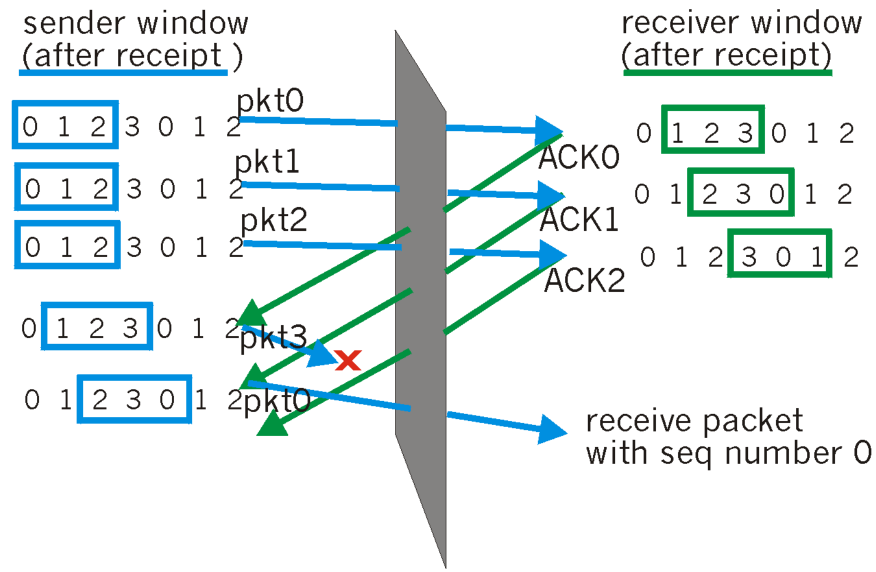
Example:

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




(a)

- receiver sees no difference in two scenarios!
- incorrectly passes duplicate data as new in (a)




(b)

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

# Minimal sequence range

- Assume we want to use a sender window of size $N$.
- What is the minimal number of unique sequence numbers we should allow to prevent such errors?
- The cyclic sequence number should never cause the sender and receiver's window to ambiguously overlap
- In FIFO channels:
  - GBN: $N + 1$
  - SR: $2N$
  - Proof: on-board

# Minimal sequence range (cont.)

- In non-FIFO channel, this cannot be guaranteed!
  - We assume that in realistic channels, old packets are cleared from the network after a reasonable time, so accidental overlap does not occur of the range of sequence numbers is "big enough".

# Exercise (Kurose & Ross, 5th ed.)

- Are the following statements true or false?
- With SR, it is possible for the sender to receive an ACK for a packet that falls outside of its current window.
- True. Suppose sender has a window size of 3.
  - Time $t_0$: it sends packets 1, 2, 3.
  - Time $t_1 > t_0$: receiver acks 1, 2, 3.
  - Time $t_2 > t_1$: sender times out and retransmits 1, 2, 3.
  - Time $t_3 > t_2$: receiver gets the duplicates and reacks 1, 2, 3.
  - Time $t_4 > t_3$: sender gets the ack sent at $t_1$, advances its window to 4, 5, 6.
  - Time $t_5 > t_4$: sender receives the acks sent at $t_2$, that fall outside of its current window.
- With GBN, it is possible for the sender to receive an ACK for a packet that falls outside of its current window.
- True, with the same scenario as described above. Only need to replace the selective acks with cumulative acks.

# Exercise (Kurose & Ross, 5th ed.)

- Are the following statements true or false?
- rdt 3.0 is the same as SR with a sender and receiver window size of 1.
- rdt 3.0 is the same as GBN with a sender and receiver window size of 1.
- Both are true. With a window size of 1, SR, GBN, and the rtd 3.0 are functionally equivalent.
  - The window size of 1 precludes the possibility of out-of-order packets (within the window).
  - A cumulative ACK is just an ordinary ACK in this situation, since it can only refer to the single packet within the window.

# Exercise

- Recall the GBN receiver: assume it is waiting for packet $m$ (i.e., it received correctly all the packets up to $m-1$ inclusive).
    - When a data packet with sequence $n=m$ is received, the receiver accepts it and advances its window.
    - Whenever a data packet with sequence $n \neq m$ is received, the receiver discards it and resends ack $m$ ("I am still waiting for $m$").
- Assume a FIFO channel and an infinite sequence number. Does the protocol remain correct if we perform the following changes?
- If $n < m$ the receiver discards the packet and does not send an ack. Otherwise, operate as before.
- Incorrect. Let the sender send packets $1, \ldots, m-1$. All received correctly, but all acks are lost.
    - The receiver waits for packet $m$.
    - But whenever the sender times-out expires, it resends packets $1, \ldots, m-1$.
    - Receiver discards them and does not ack.
    - Deadlock.

# Exercise

- if $n > m$, the receiver discards the packet and does not send an ack. Otherwise, operate as before.
- Correct. If $n > m$ was received, but the receiver is waiting for $m$, it means we have a gap. The sender will eventually timeout for $m$, and resend packet $n$ then.