SC250
Computer Networking I

Transport Layer Principles

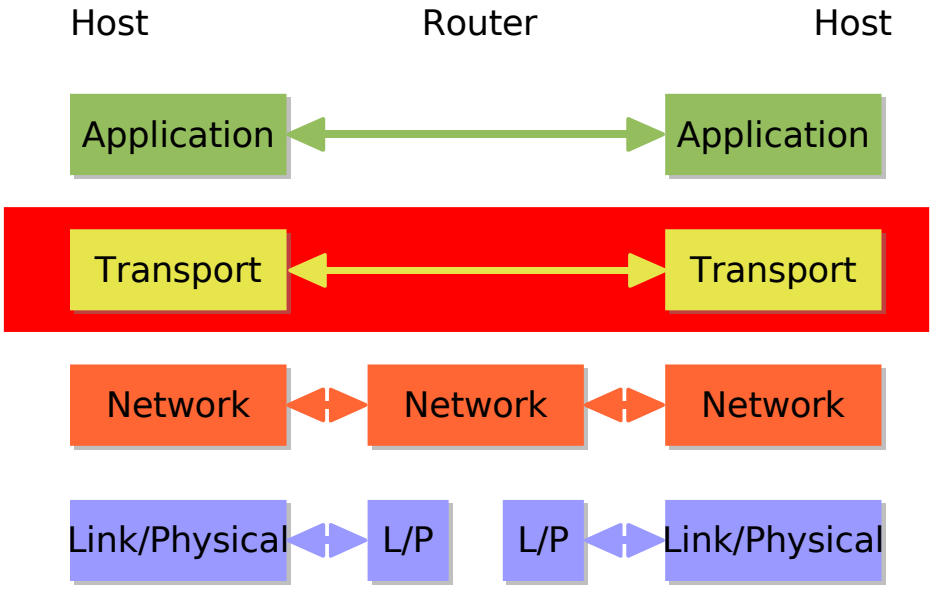
Prof. Matthias Grossglauser

School of Computer and Communication Sciences
EPFL

http://lcawww.epfl.ch

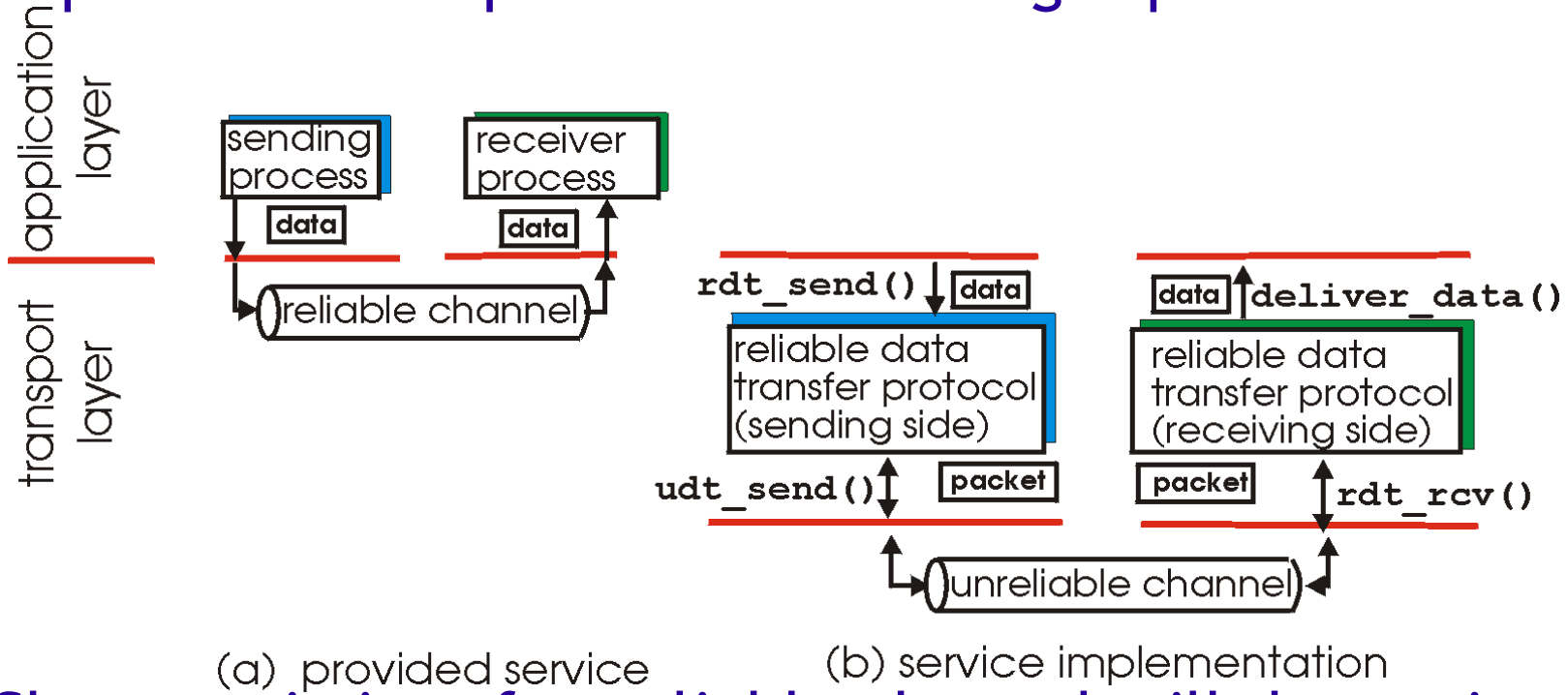
Principles of Reliable Data Transfer

- Reliable transfer
  - Getting data without any alteration (bit errors, loss, duplication,...) from a sender to a receiver
  - Achieving this over an unreliable channel, i.e., a channel that **does** alter information
  - We will need two-way unreliable channel, even to achieve one-way reliable transfer
- Internet:
  - IP: unreliable channel
  - TCP: reliable service, available through socket API



Principles of Reliable Data Transfer

- Applies to transport layer (TCP), but also to some link-layer protocols
- Top-10 list of important networking topics!

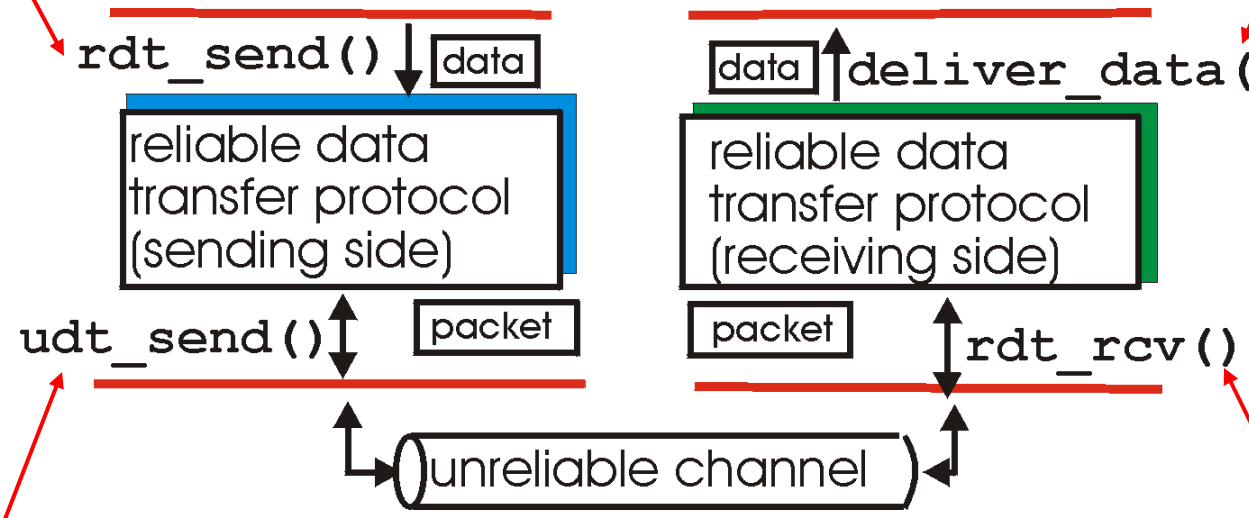


- Characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Reliable Data Transfer: Getting Started

`rdt_send()`: called from above, (e.g., by app.). Passed data to deliver to receiver upper layer

`deliver_data()`: called by **rdt** to deliver data to upper layer



`udt_send()`: called by rdt, to transfer packet over unreliable channel to receiver

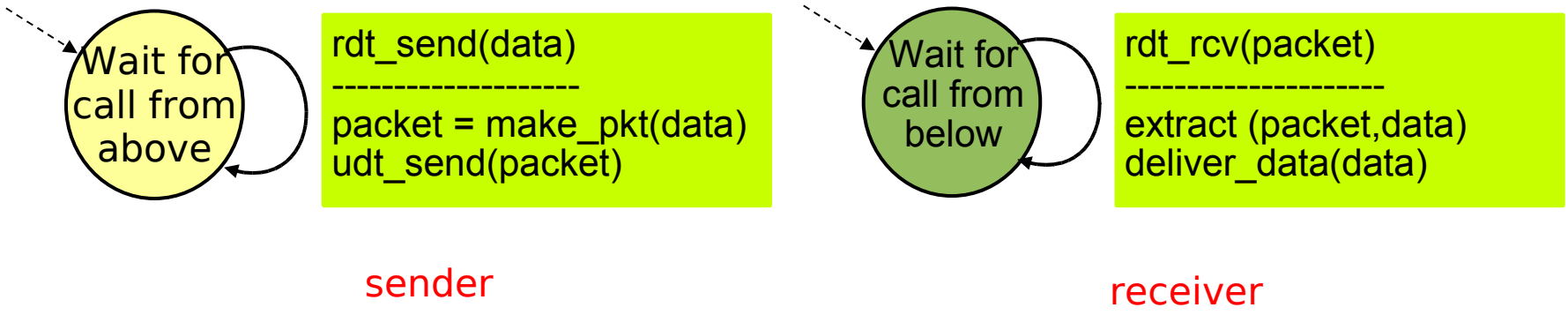
`rdt_rcv()`: called when packet arrives on rcv-side of channel

Reliable Data Transfer: Getting Started

- Incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- Consider only unidirectional data transfer
  - but control info will flow on both directions!
- Use finite state machines (FSM) to specify sender, receiver

RDT1.0: Reliable Transfer over Reliable Channel

- Underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- Separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver read data from underlying channel



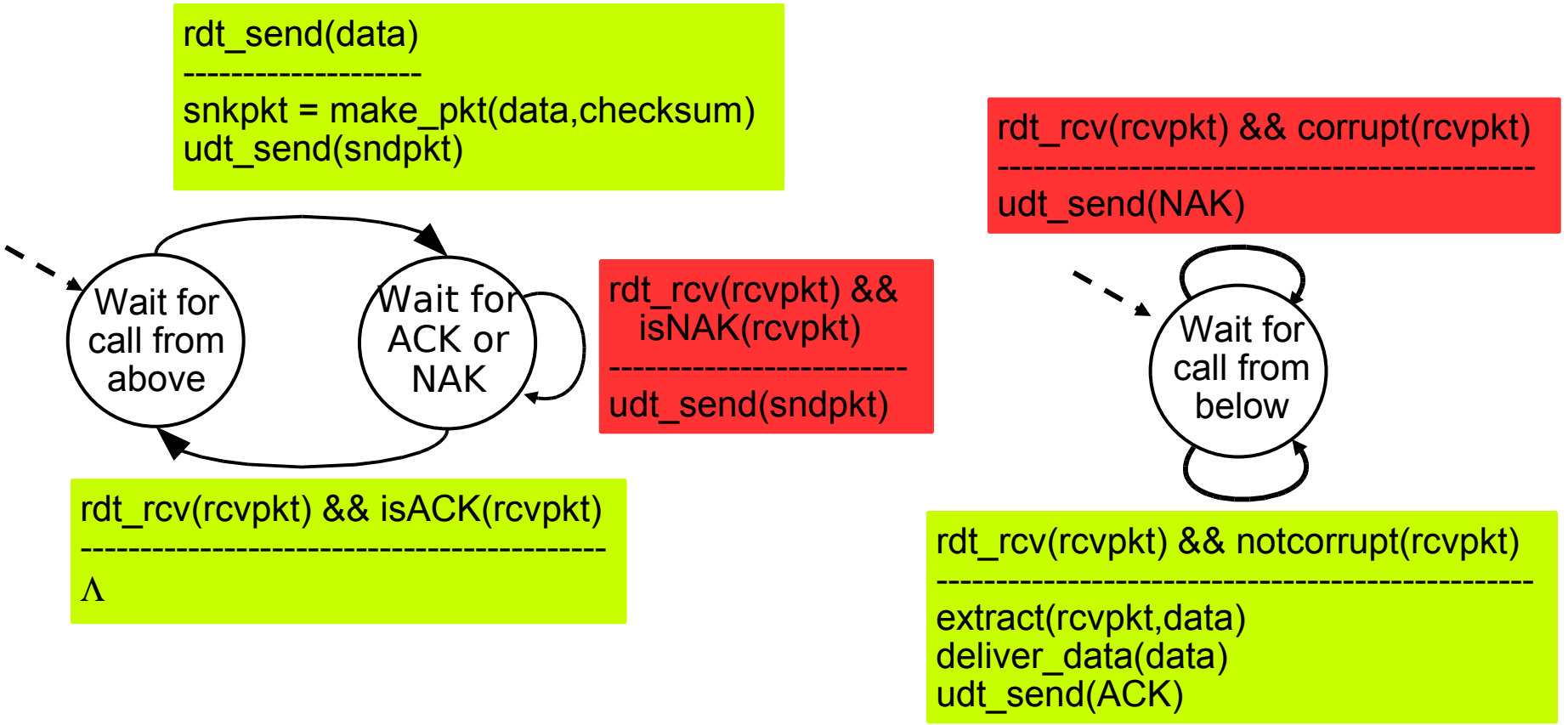
RDT2.0: Channel with Bit Errors

- Underlying channel may flip bits in packet
  - checksum to detect bit errors: send (m, c), where c=h(m) is the checksum; receiver checks whether c=h(m)
- The question: how to recover from errors:
  - Automatic repeat request (ARQ): control information from receiver back to sender
  - acknowledgements (ACKs): receiver explicitly tells sender that pkt received OK
  - negative acknowledgements (NAKs): receiver explicitly tells sender that pkt had errors
  - sender retransmits pkt on receipt of NAK
- New mechanisms in rdt2.0 (beyond rdt1.0):
  - error detection, ACK/NAK control messages

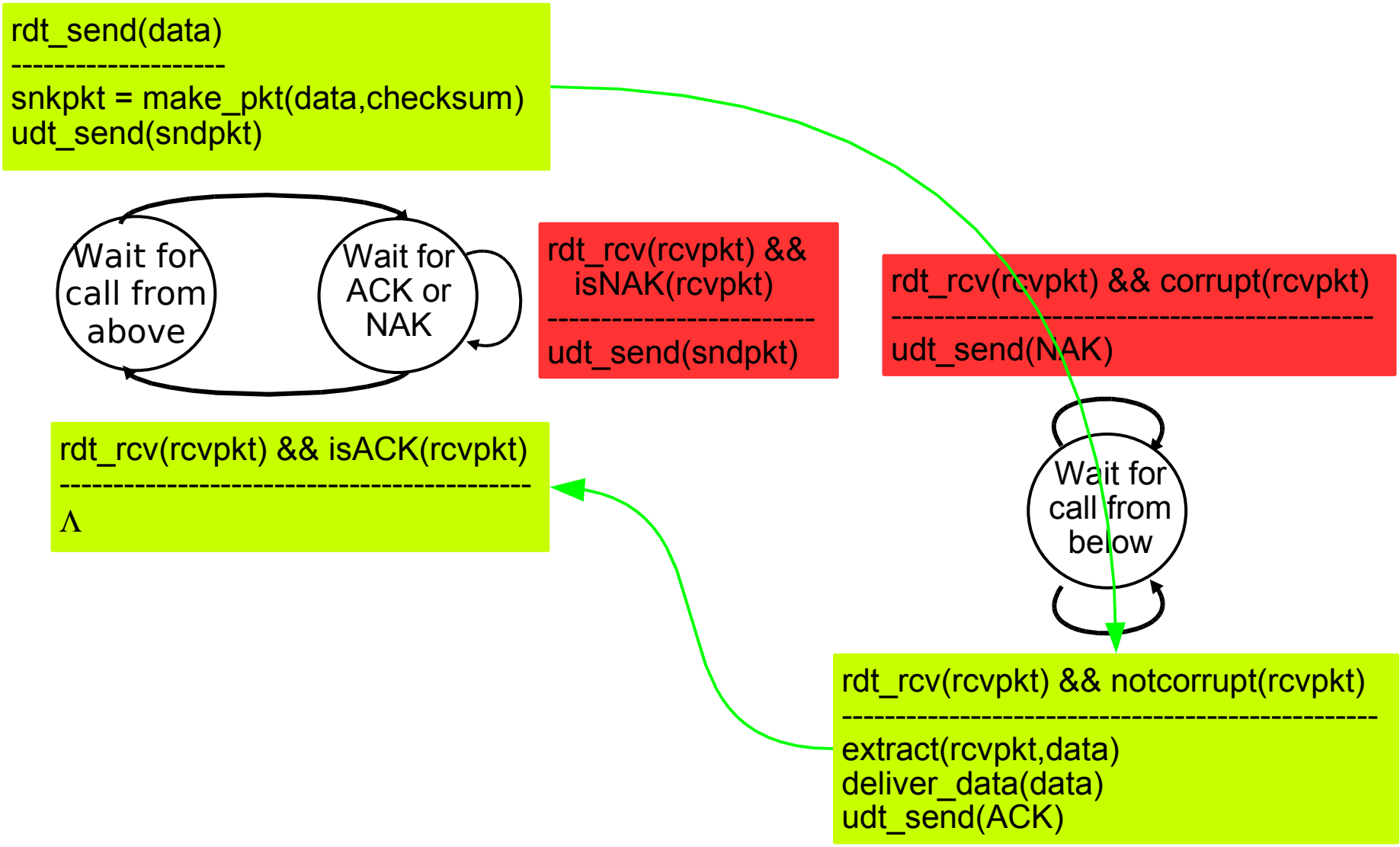
RDT2.0: FSM Specification

sender

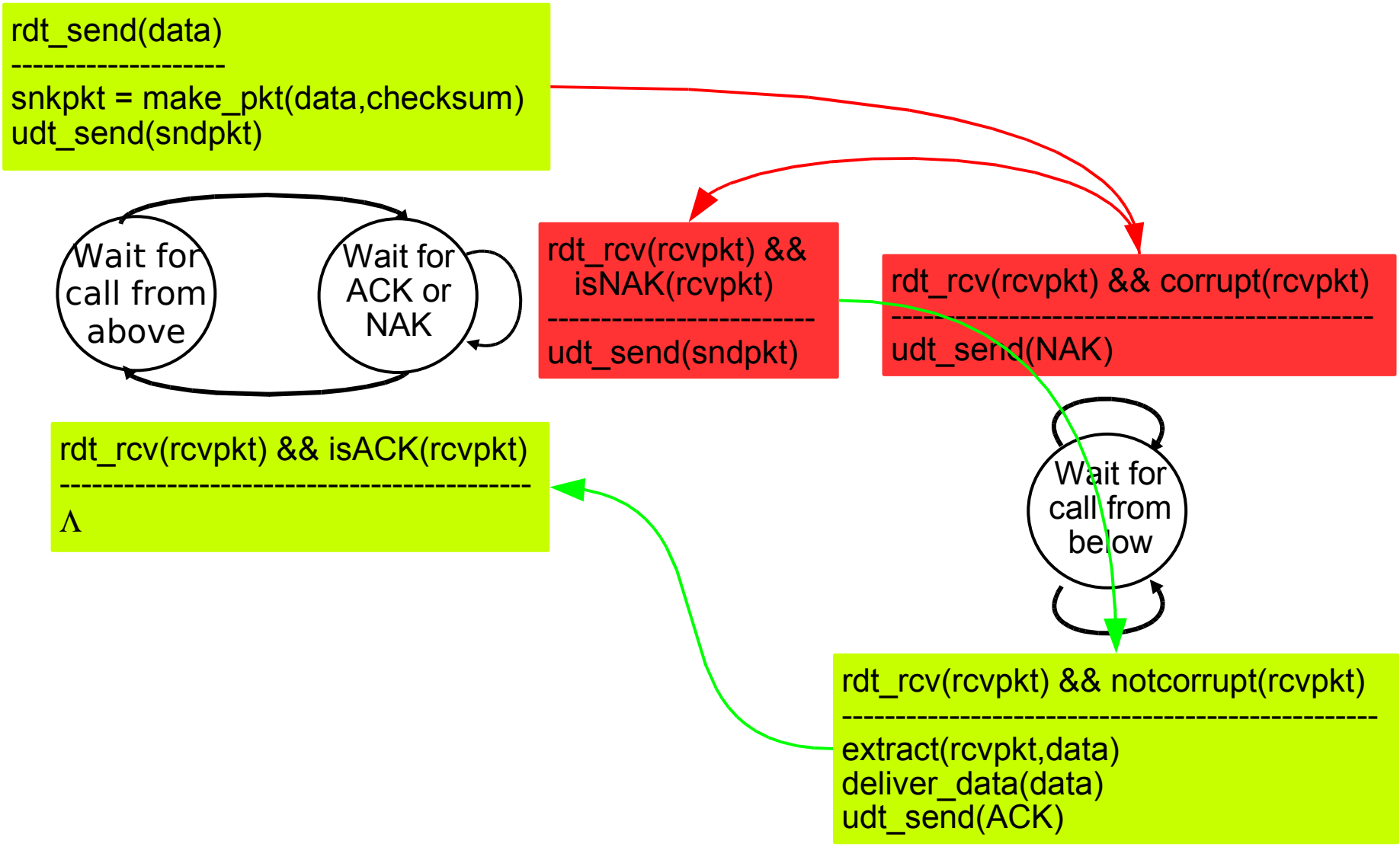
receiver



RDT2.0: Operation with no Errors



RDT2.0: Error Scenario



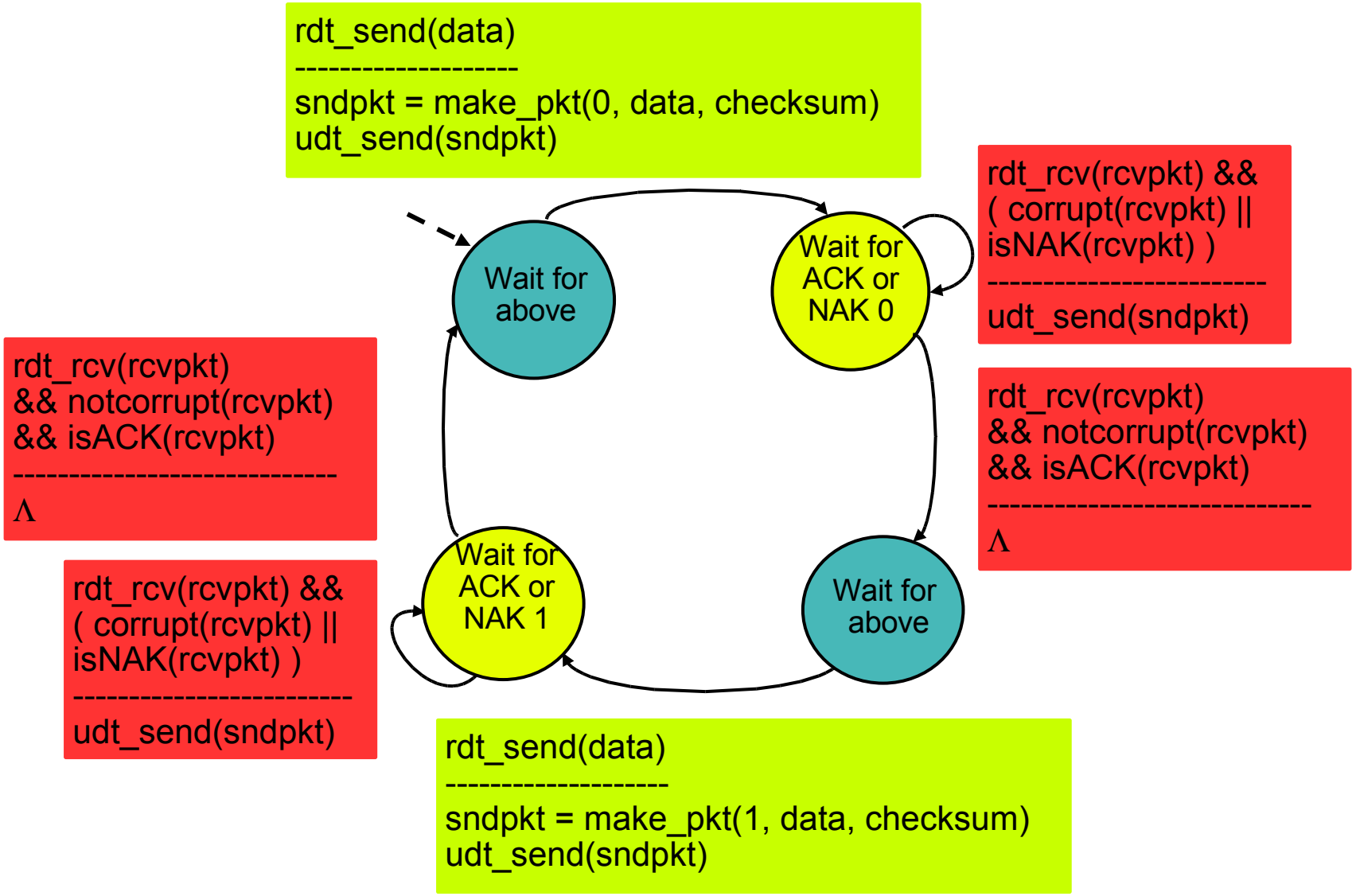
RDT2.0 has a Fatal Flaw!

- What happens if ACK/NAK corrupted?
  - sender doesn't know what happened at receiver!
  - can't just retransmit: possible duplicate
- What to do?
  - sender ACKs/NAKs receiver's ACK/NAK? What if sender ACK/NAK lost?
  - retransmit, but this might cause retransmission of correctly received pkt!
- Handling duplicates:
  - sender adds sequence number to each pkt
  - sender retransmits current pkt if ACK/NAK garbled
  - receiver discards (doesn't deliver up) duplicate pkt

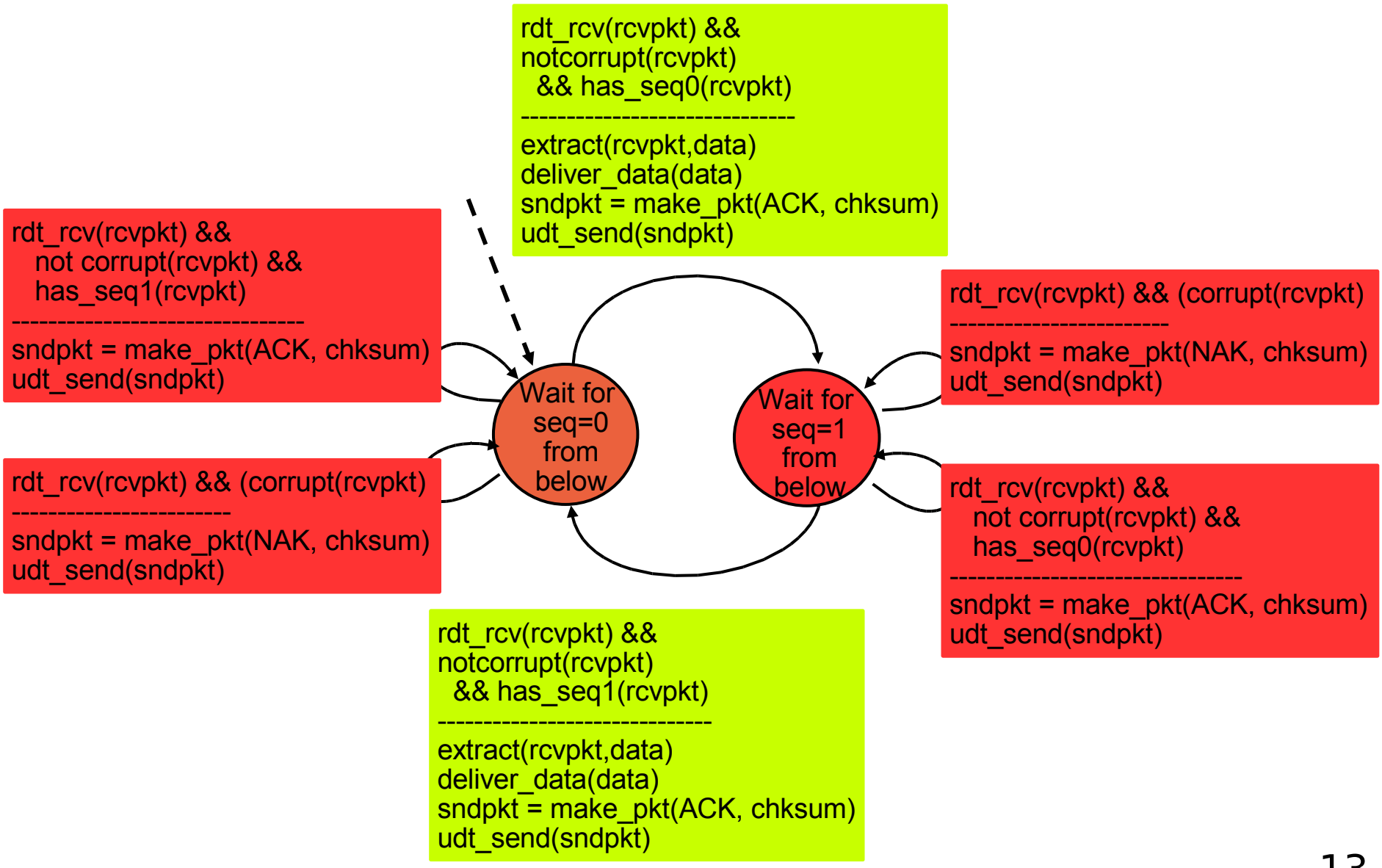
stop and wait

Sender sends one packet, then waits for receiver response

RDT2.1 Sender: Handles Garbled ACK/NAKs



RDT2.1, Receiver: Handles Garbled ACK/NAKs



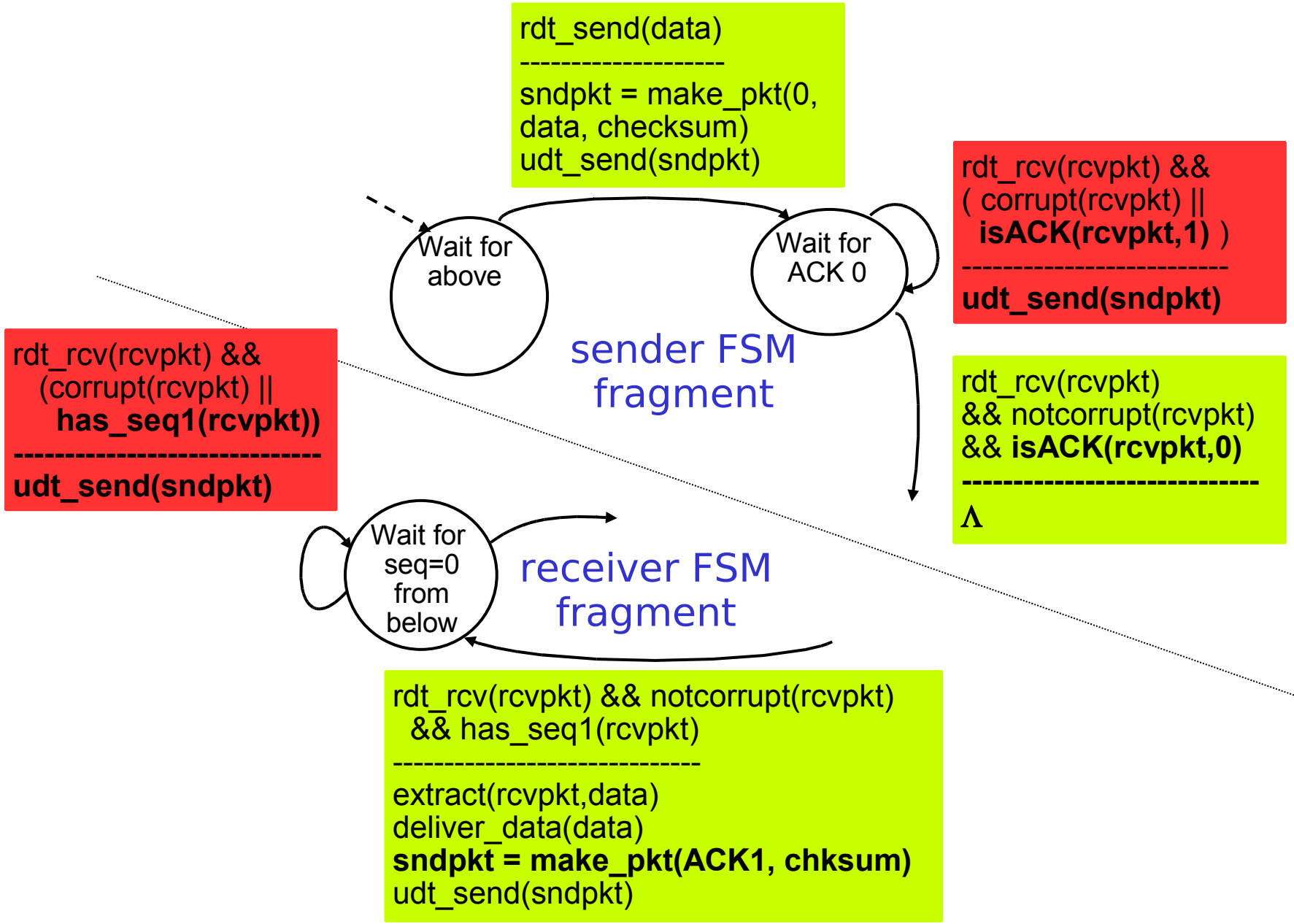
RDT2.1: Discussion

- Sender:
  - Seq # added to pkt
  - Two seq. #'s (0,1) will suffice. Why?
  - Must check if received ACK/NAK corrupted
  - Twice as many states
    - state must "remember" whether "current" pkt has 0 or 1 seq. #
- Receiver:
  - Must check if received packet is duplicate
    - state indicates whether 0 or 1 is expected pkt seq #
  - Note: receiver can not know if its last ACK/NAK received OK at sender

RDT2.2: A NAK-Free Protocol

- Same functionality as rdt2.1, using ACKs (NAKs) only
- Instead of NAK, receiver sends ACK for last pkt received OK
  - receiver must *explicitly* include seq # of pkt being ACKed
- Duplicate ACK at sender results in same action as NAK: *retransmit current pkt*

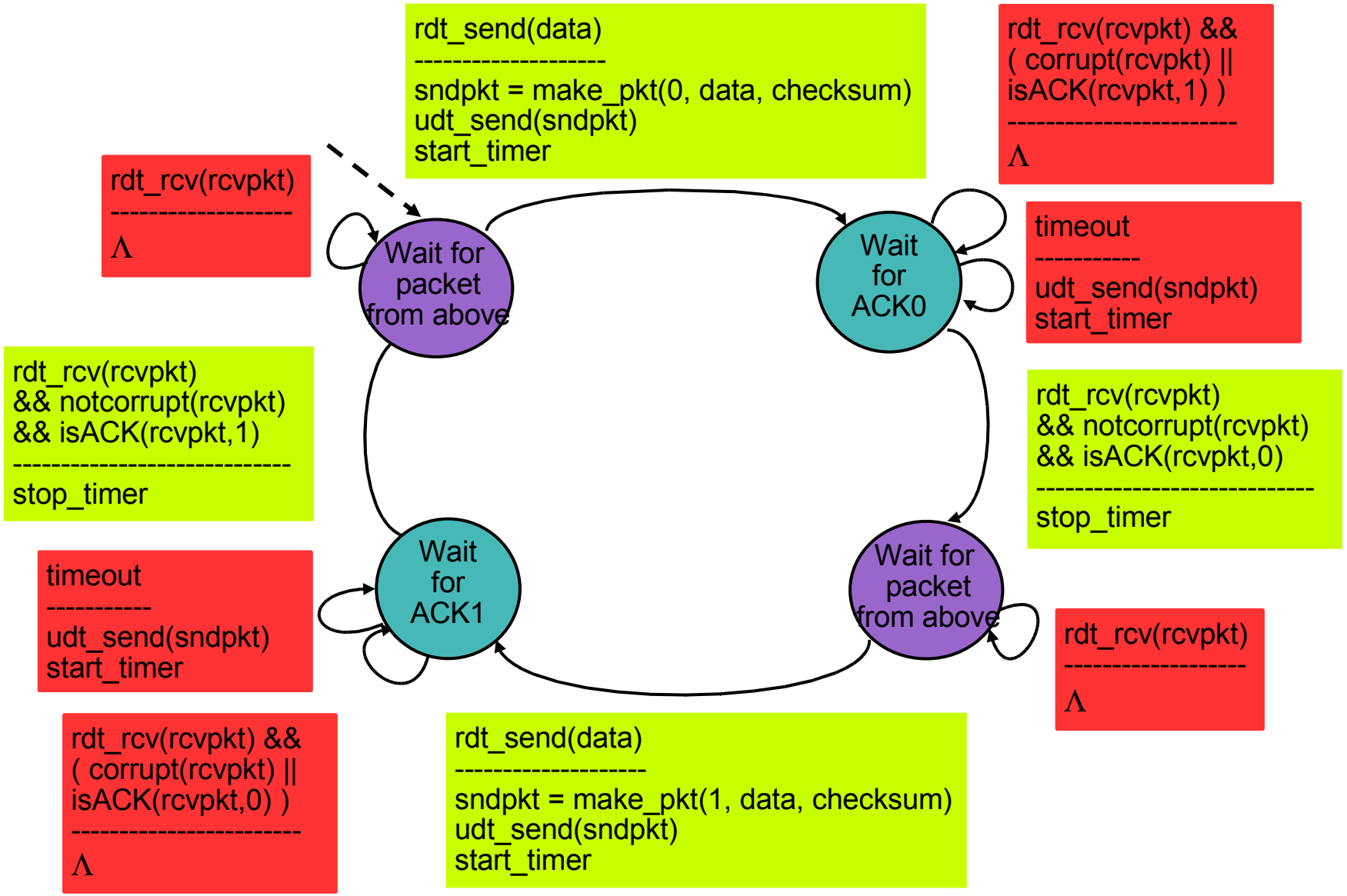
RDT2.2 Sender and Receiver Fragments



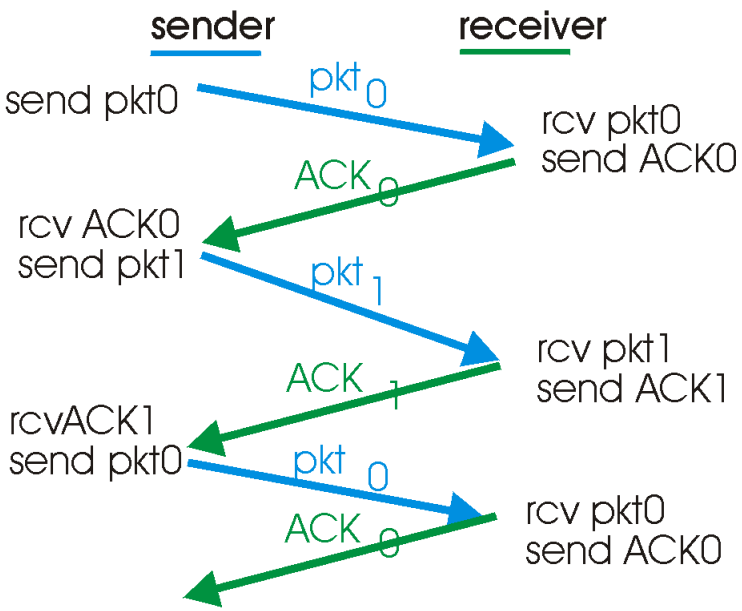
RDT3.0: Channels with Errors *and* Loss

- New assumption: underlying channel can also lose packets (data or ACKs)
  - checksum, seq. #, ACKs, retransmissions will be of help, but not enough
- Q: how to deal with loss?
  - sender waits until certain that data or ACK lost, then retransmits
  - drawbacks?
- Approach: sender waits "reasonable" amount of time for ACK
  - retransmits if no ACK received in this time
  - if pkt (or ACK) just delayed (not lost):
    - retransmission will be duplicate, but use of seq. #'s already handles this
    - receiver must specify seq # of pkt being ACKed
  - requires countdown timer

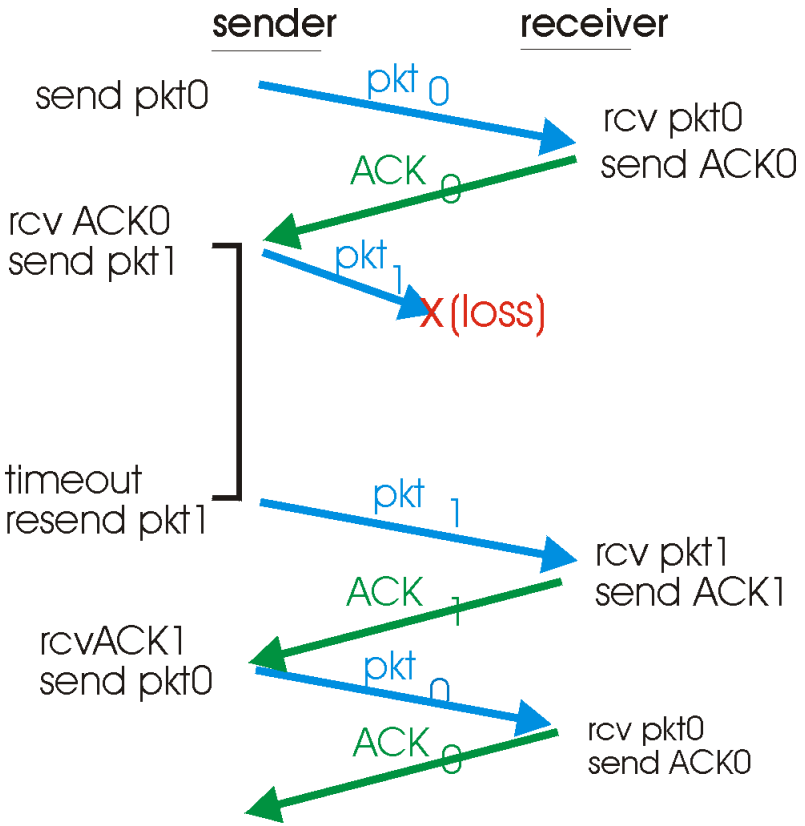
RDT3.0 Sender



RDT3.0 in Action

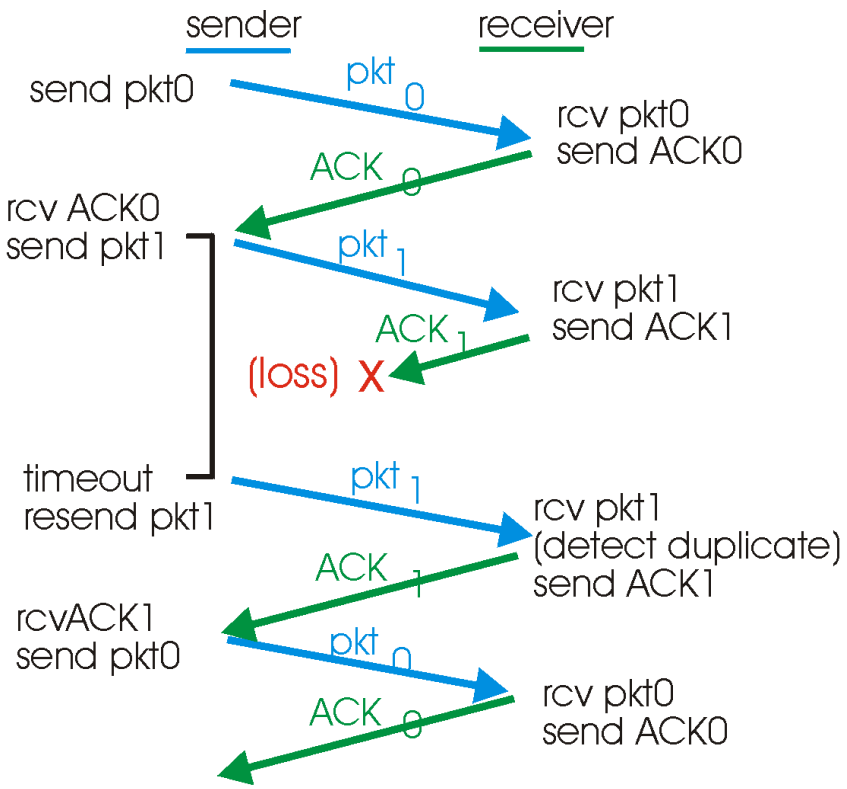


(a) operation with no loss

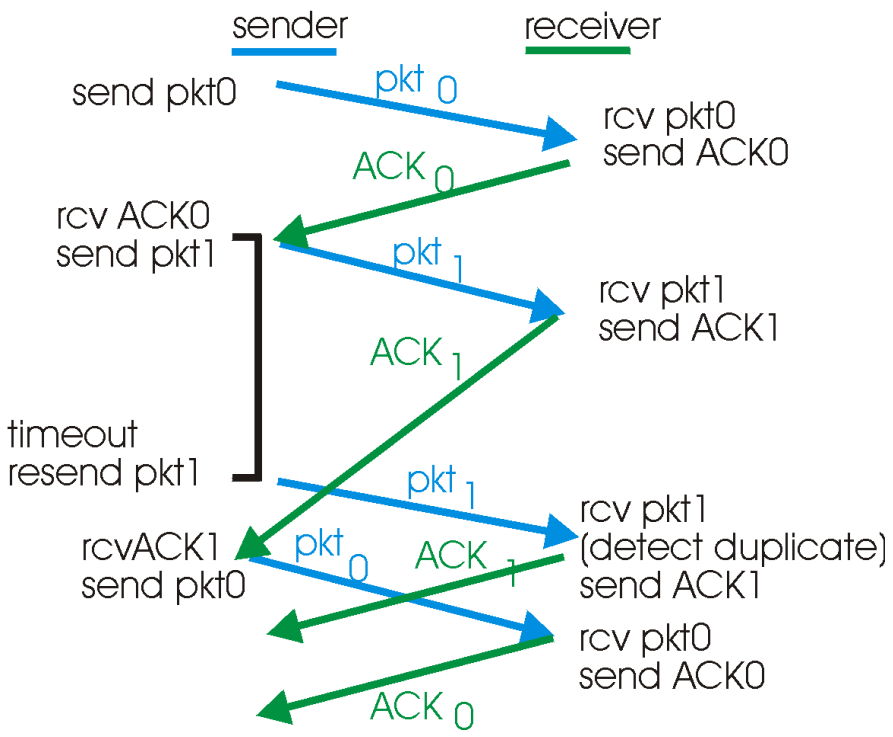


(b) lost packet

RDT3.0 in Action

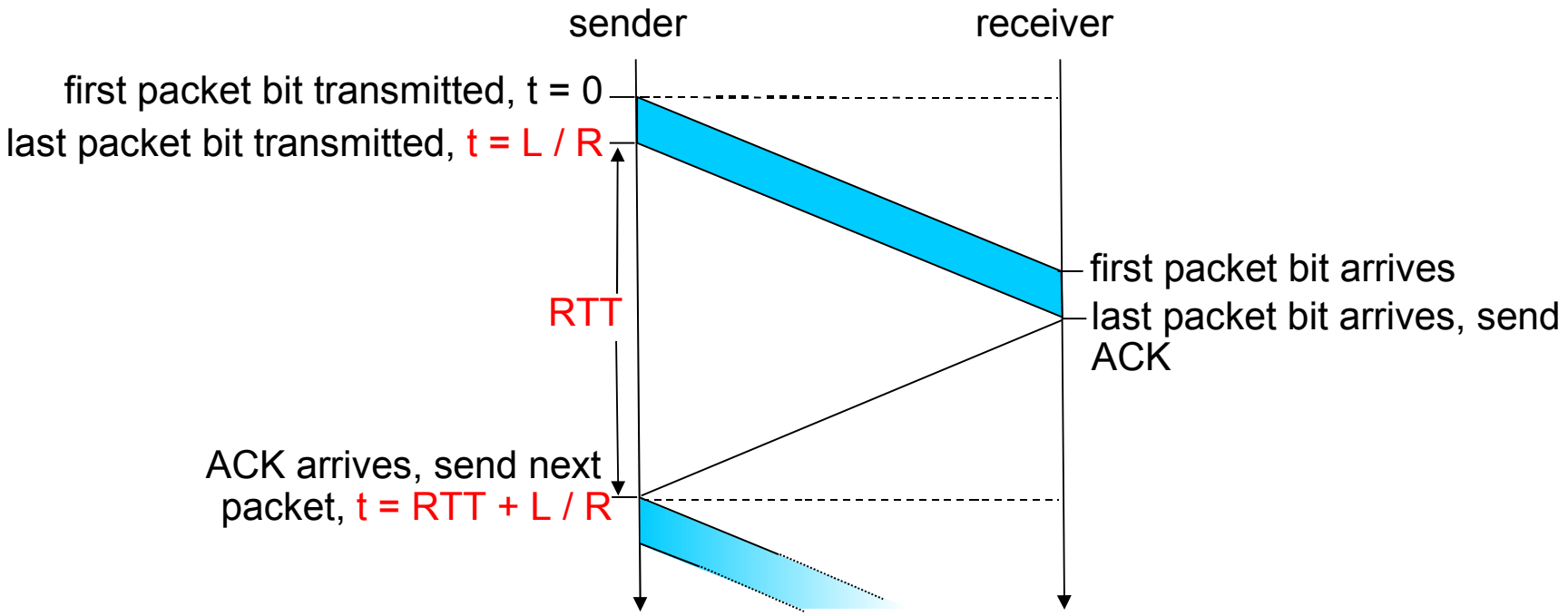


(c) lost ACK



(d) premature timeout

RDT3.0: Stop-and-Wait Operation



$$U_{sender} = \frac{L/R}{L/R + RTT}$$

Performance

- rdt3.0 works, but lousy performance
- example: 1 Gbps link, 15 ms e2e prop. delay, 1KB packet:

$$T_{transmit} = \frac{L \text{ (packet length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{8kb/pkt}{1e9\ b/sec} = 8 \text{ microsec}$$

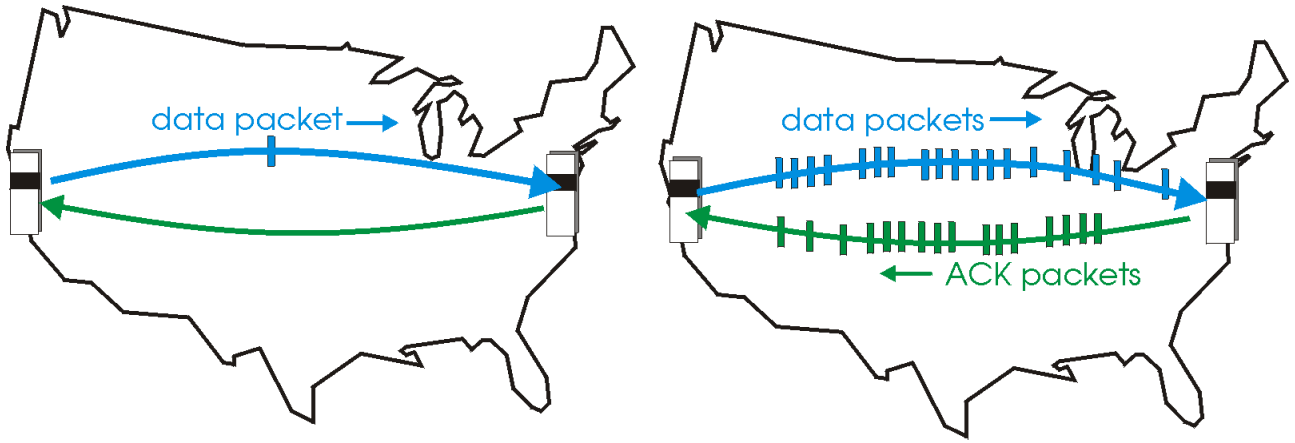
$$U_{sender} = \frac{L/R}{RTT+L/R} = \frac{0.008}{30.008} = 0.00027$$

- U_{sender}: utilization – fraction of time sender busy sending
- 1KB pkt every 30 msec -> 33kB/sec thruput over 1 Gbps link
- protocol limits use of physical resources!

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

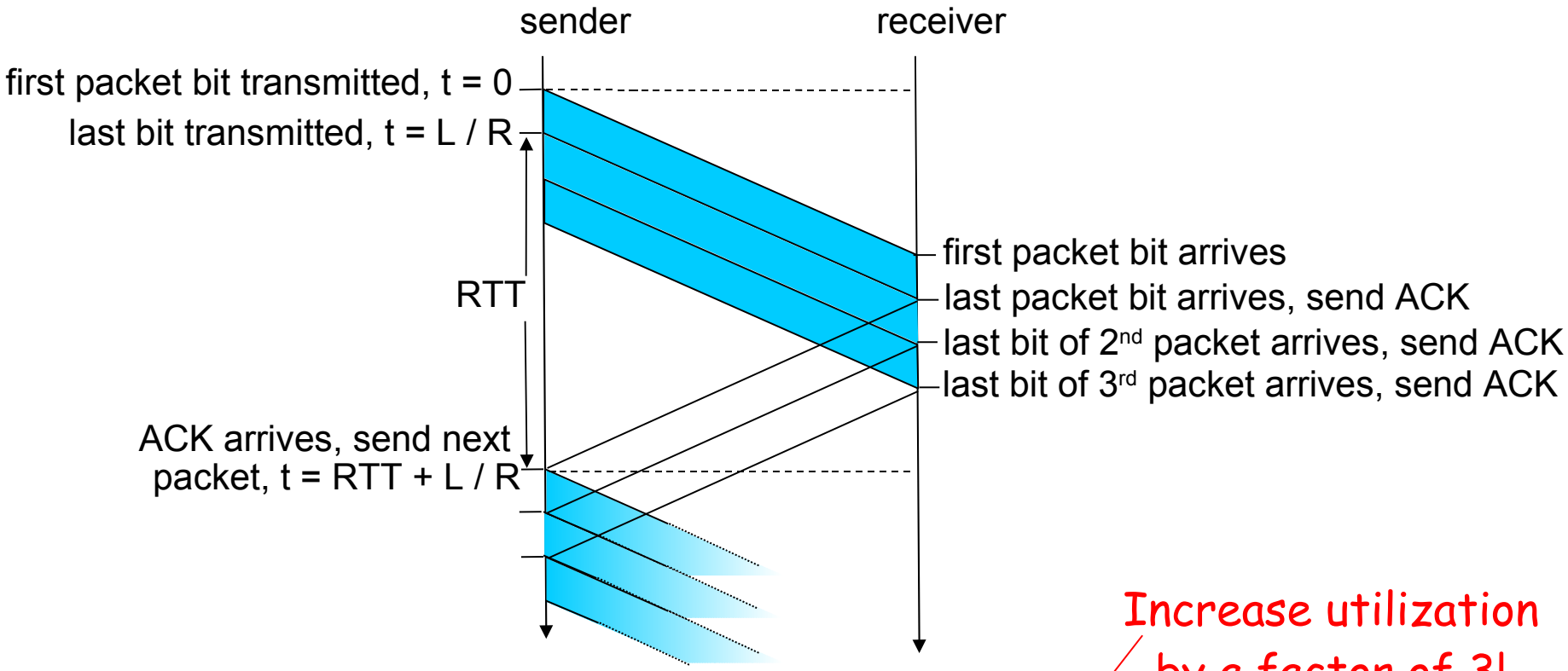
(b) a pipelined protocol in operation

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

Concepts for Pipelined Protocols

- Acknowledgements:
  - Selective ACK(n), NAK(n):
    - Says “packet n received/not received”
  - Cumulative cACK(n) = {...ACK(n-2),ACK(n-1),ACK(n)}
    - Says “every packet up to and including n received”
- Timer:
  - Should be larger than RTT, otherwise risk of malfunction
- Sequence number space:
  - Set of identifiers for packets (stop-and-go: {0,1})
- Sliding window:
  - Interval of sequence number space, advances upon receipt
  - Budget of in-flight data for pipelined operation
  - Note: sequence # space imposes constraint on window size

Pipelining: Increased Utilization



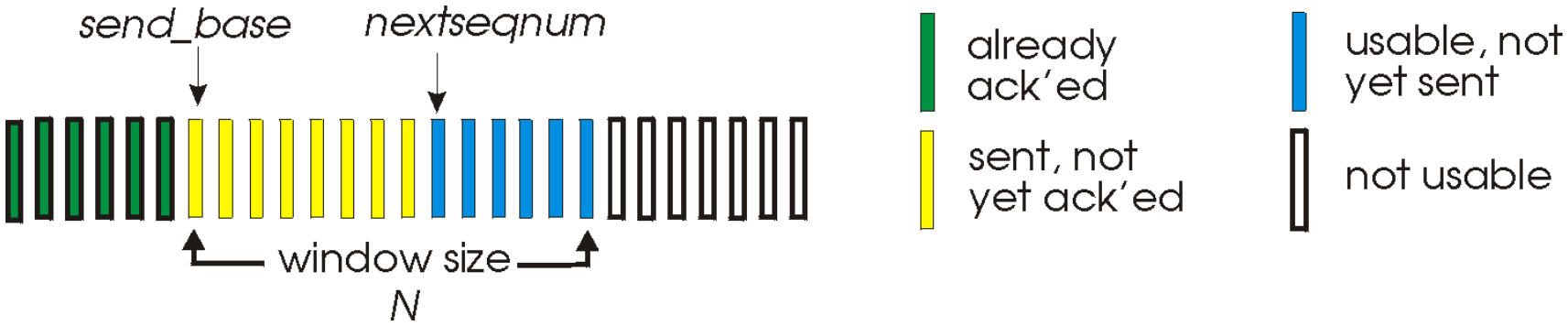
Increase utilization by a factor of 3!

$$U_{sender} = \frac{3\,L/R}{L/R + RTT}$$

Go-Back-N (GBN)

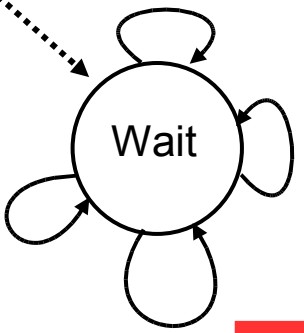
Sender:

- Sequence number (seq #) in packet header
- Window (budget) of up to N consecutive unacknowledged pkts allowed



- cACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"
  - Says: "we are done with everything before (n+1)"
  - may receive duplicate ACKs (see receiver)
- timer for each in-flight pkt (i.e., sent, but not yet received)
- *timeout(n):* retransmit pkt n and all higher seq # pkts in window

GBN, Sender: Extended FSM



Λ
--
base=1
nextseqnum=1

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

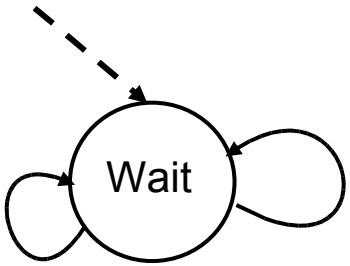
rdt_rcv(rcvpkt)
  && corrupt(rcvpkt)

Λ

rdt_rcv(rcvpkt) &&
  notcorrupt(rcvpkt)
---
base = getacknum(rcvpkt)+1
If (base == nextseqnum)
  stop_timer
 else
  start_timer

Timeout
---
start_timer
udt_send(sndpkt[base])
udt_send(sndpkt[base+1])
…
udt_send(sndpkt[nextseqnum-1])

GBN, Receiver: Extended FSM



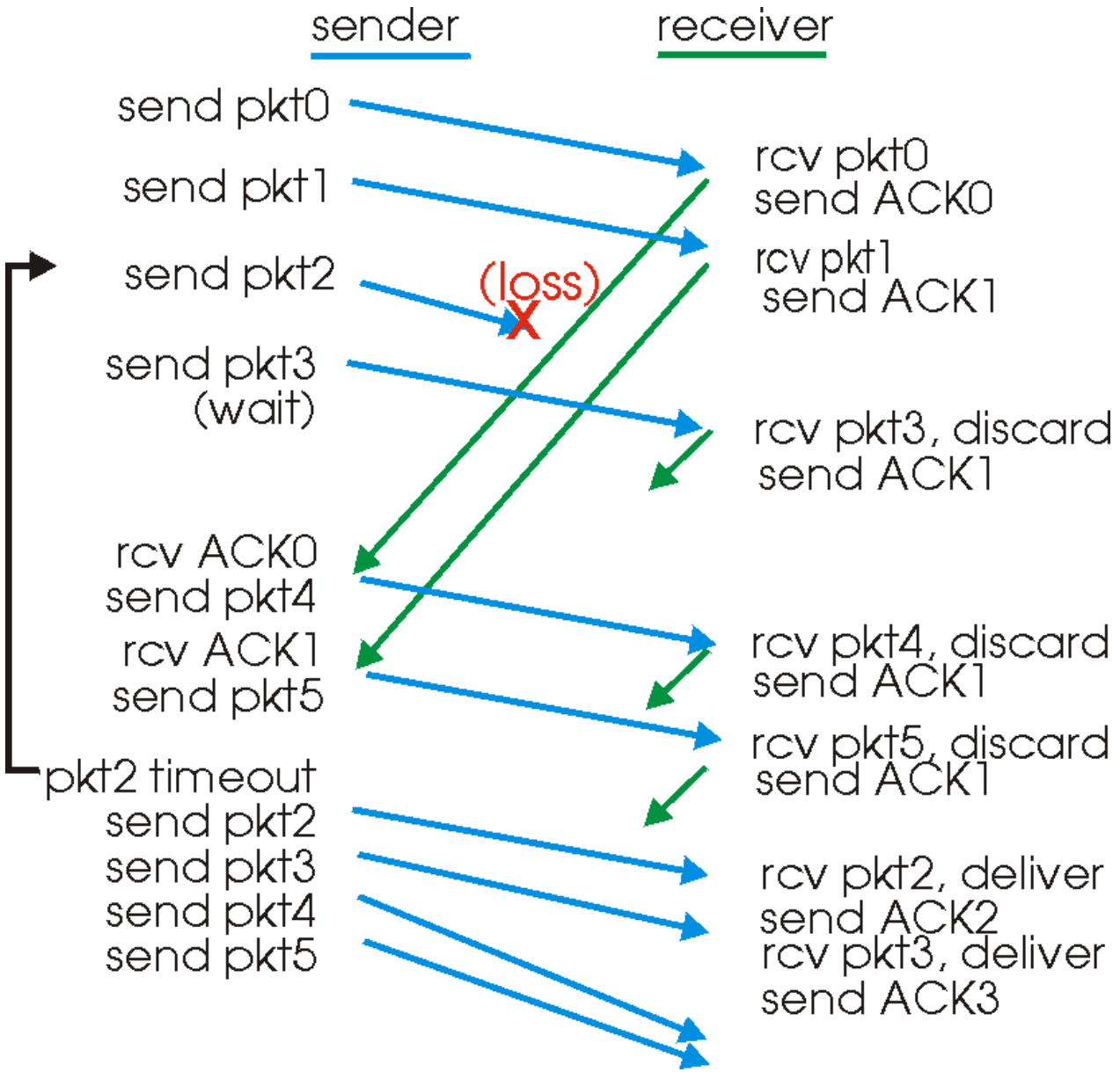
default
---
udt_send(sndpkt)

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

Λ
--
expectedseqnum=1
sndpkt =
make_pkt(expectedseqnum,cACK,chksum)

- ACK-only: always send cACK for correctly-received pkt with highest **in-order** seq #
  - may generate duplicate cACKs
  - need only remember expectedseqnum
- Out-of-order packets:
  - Discard (don't buffer) -> **no receiver buffering!**
  - Re-ACK pkt with highest in-order seq #

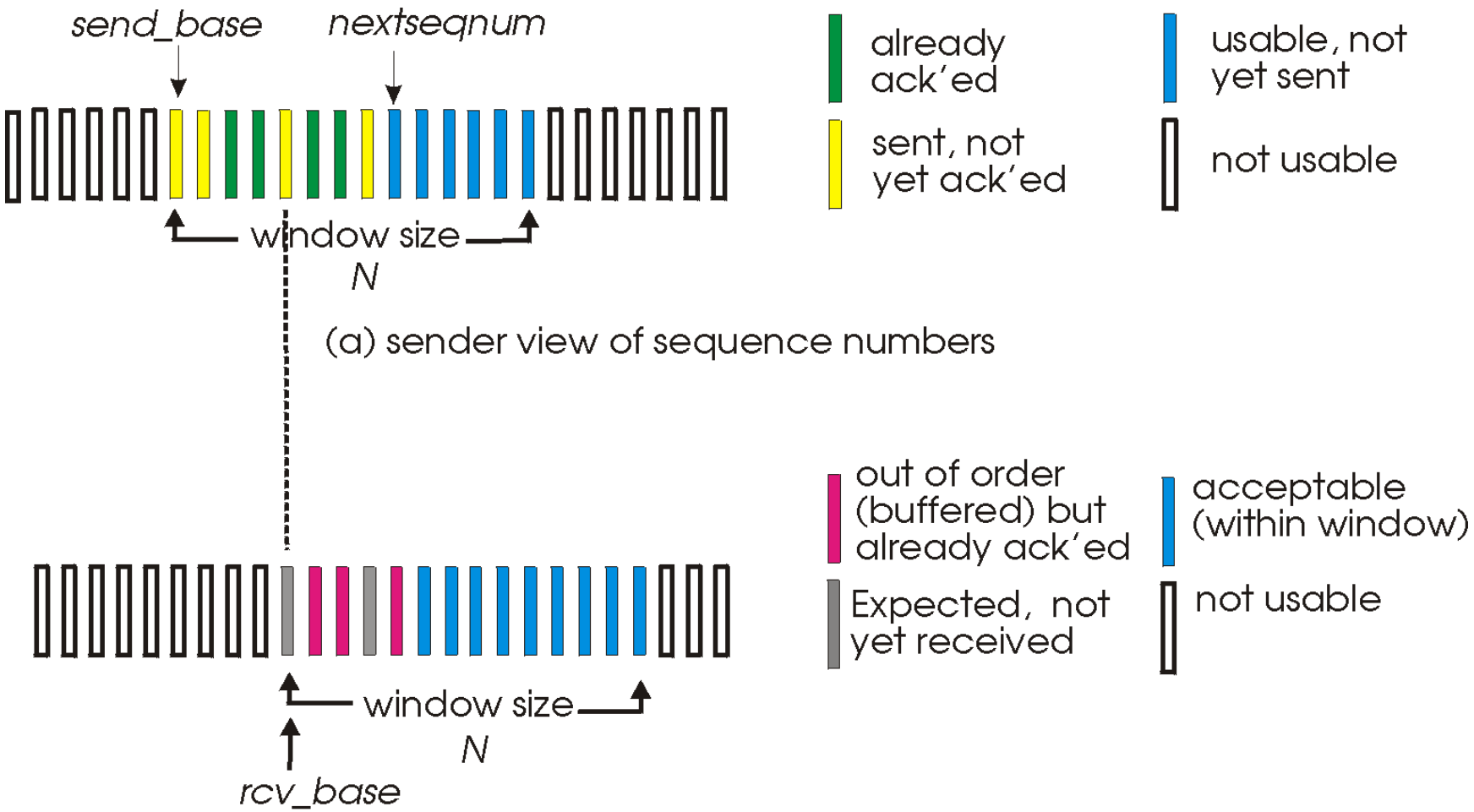
GBN in Action



Selective Repeat (SR)

- Drawback of GBN:
  - Entire window gets retransmitted due to a single error
  - Especially costly with large window
- 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, unacknowledged 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

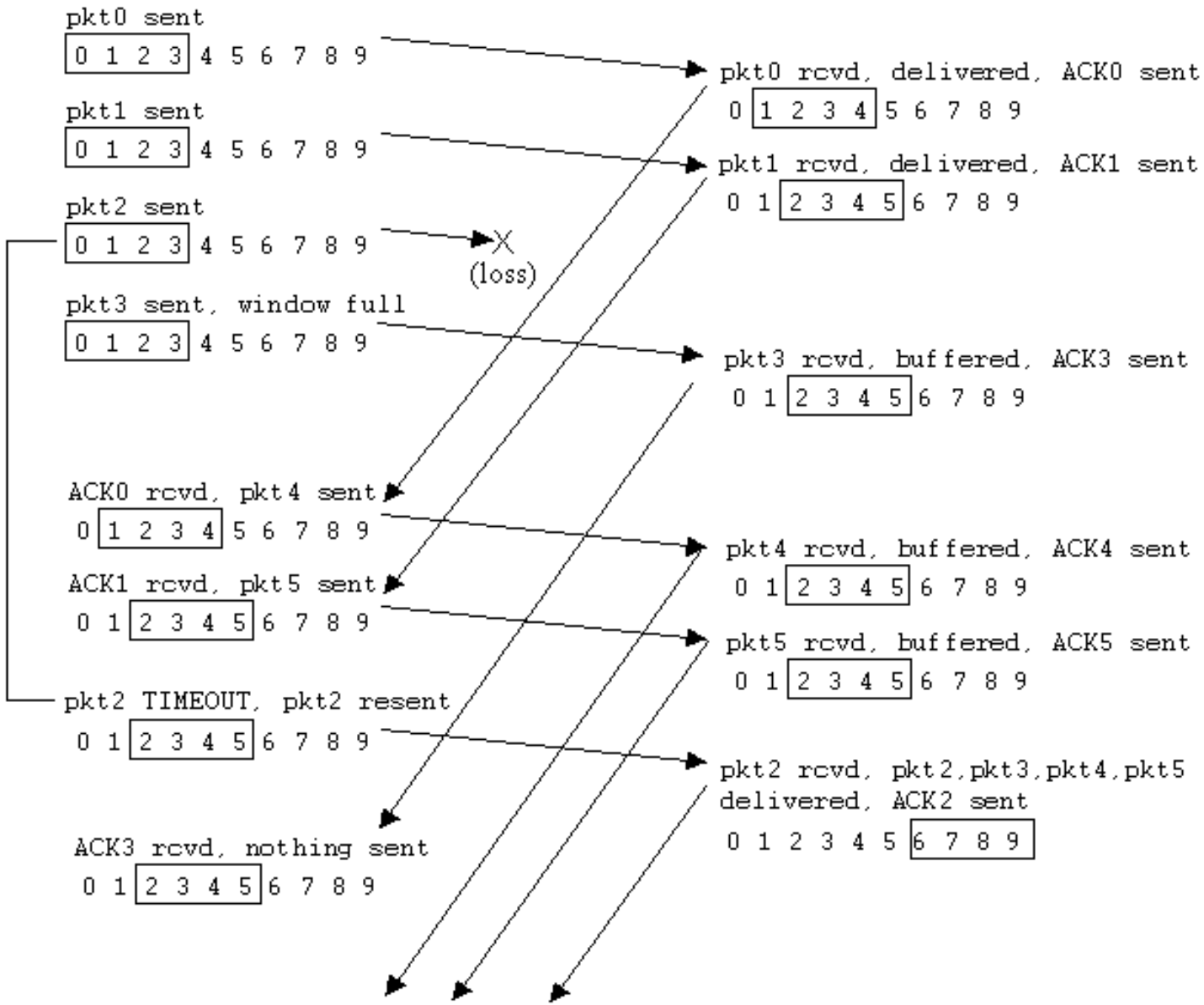
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignore

Selective Repeat in Action



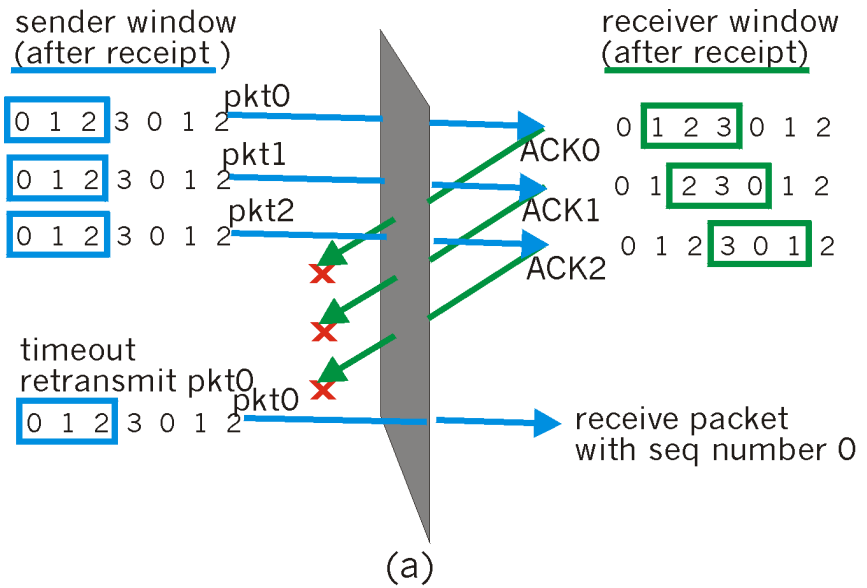
SR: Sizes of Window and Sequence # Space

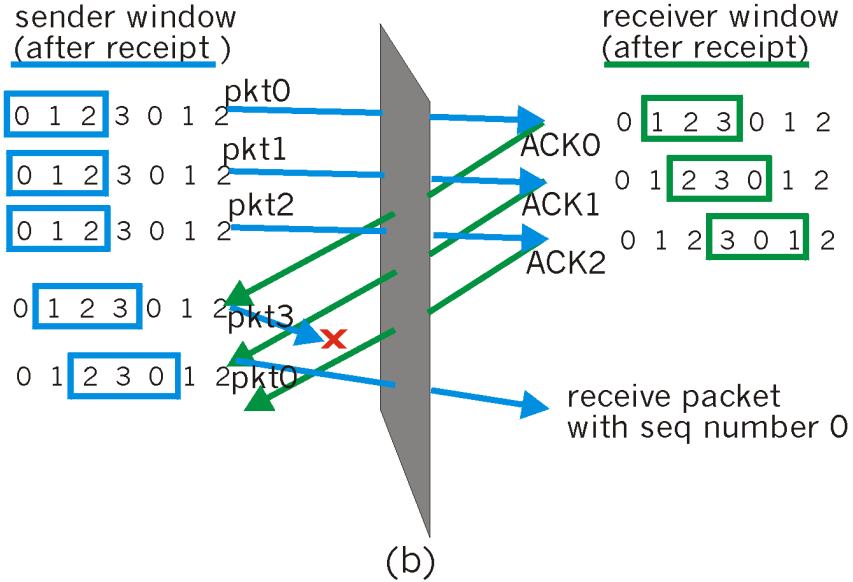
Example:

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

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

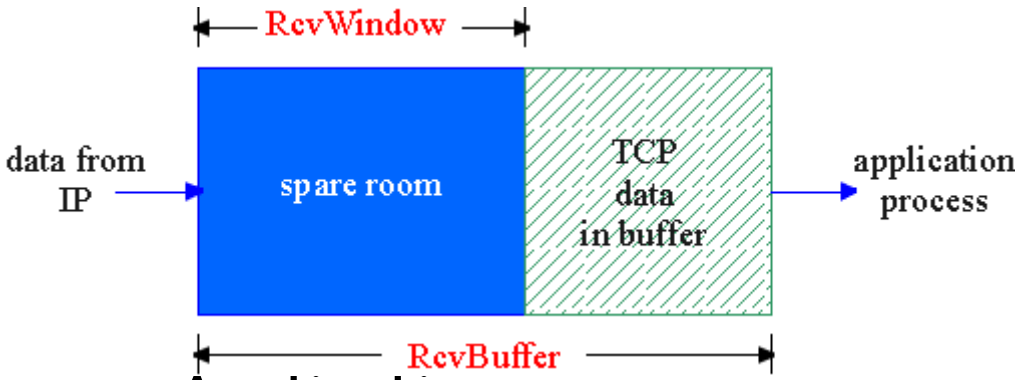
Q: what relationship between seq # size and window size?





Flow Control

- Receive side of TCP connection has a receive buffer:



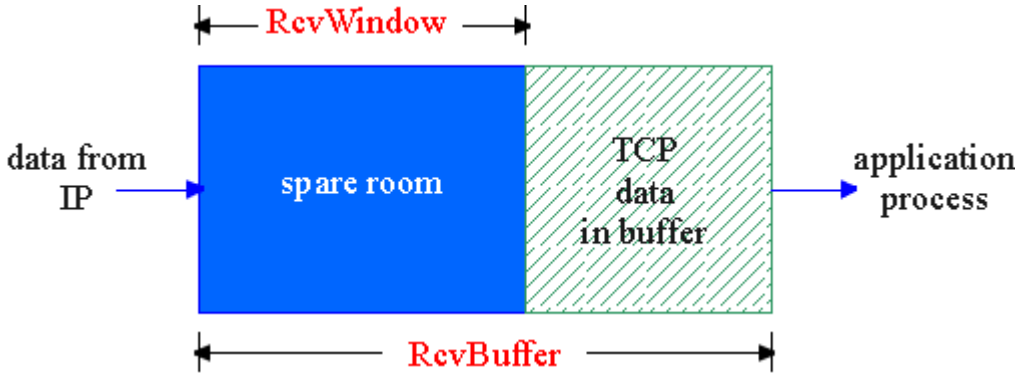
- Application process may be slow at reading from buffer

flow control

sender won't overflow receiver's buffer by transmitting too much, too fast

- Speed-matching service: matching the send rate to the receiving app's drain rate

Flow Control: How it Works



(Suppose TCP receiver discards out-of-order segments)

- spare room in buffer

= **RcvWindow**

= **RcvBuffer-[LastByteRcvd - LastByteRead]**

- Rcvr advertises spare room by including value of **RcvWindow** in segments
- Sender limits unACKed data to **RcvWindow**
  - guarantees receive buffer doesn't overflow

Summary: Transport Layer Principles

- Transport layer functionality
  - Types of errors
    - Corrupted packets (bit errors) -> checksums
    - Packet loss (queue overflow) -> sequence numbers
    - Not considered: reordering (both of data and ACK packets)
  - Reliable delivery
    - ARQ (Automatic Repeat Request): ACKs, NAKs, cACKs
    - Pipelining for performance
  - Flow control
    - Matching sender speed to receiver
  - Note: these are all end-to-end issues; another issue is congestion control, i.e., how to share the network
- TCP: main reliable transport protocol in TCP/IP