# Reliable Data Transfer Protocol

CS 3516
B-term 2018

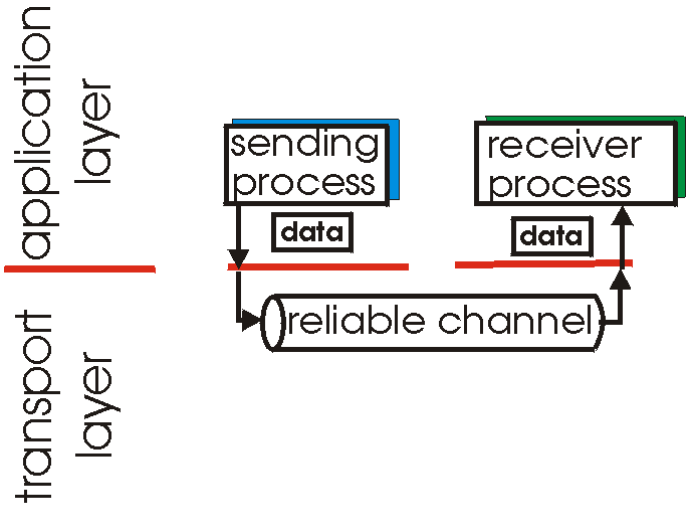
Instructor: Krishna Venkatasubramanian

# Announcements

- Project 1 due on Friday
- Project 2 and Lab2 will be out on Friday
- Quiz 2 on Friday
  - Will include questions from topics in chapter 1 (metrics), application layer, along with topics from today

# Principles of reliable data transfer

- important in application, transport, link layers
  - top-10 list of important networking topics!

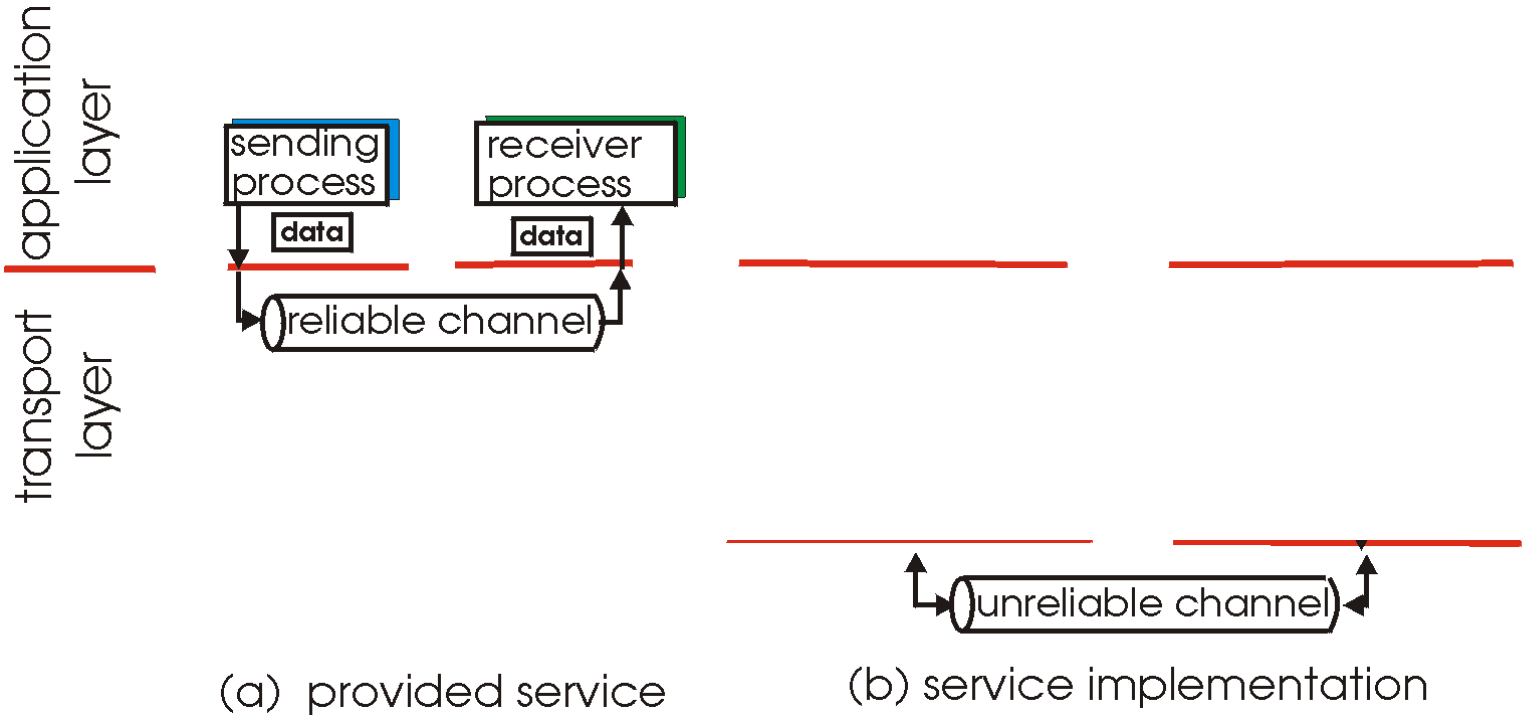



(a) provided service

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

# Principles of reliable data transfer

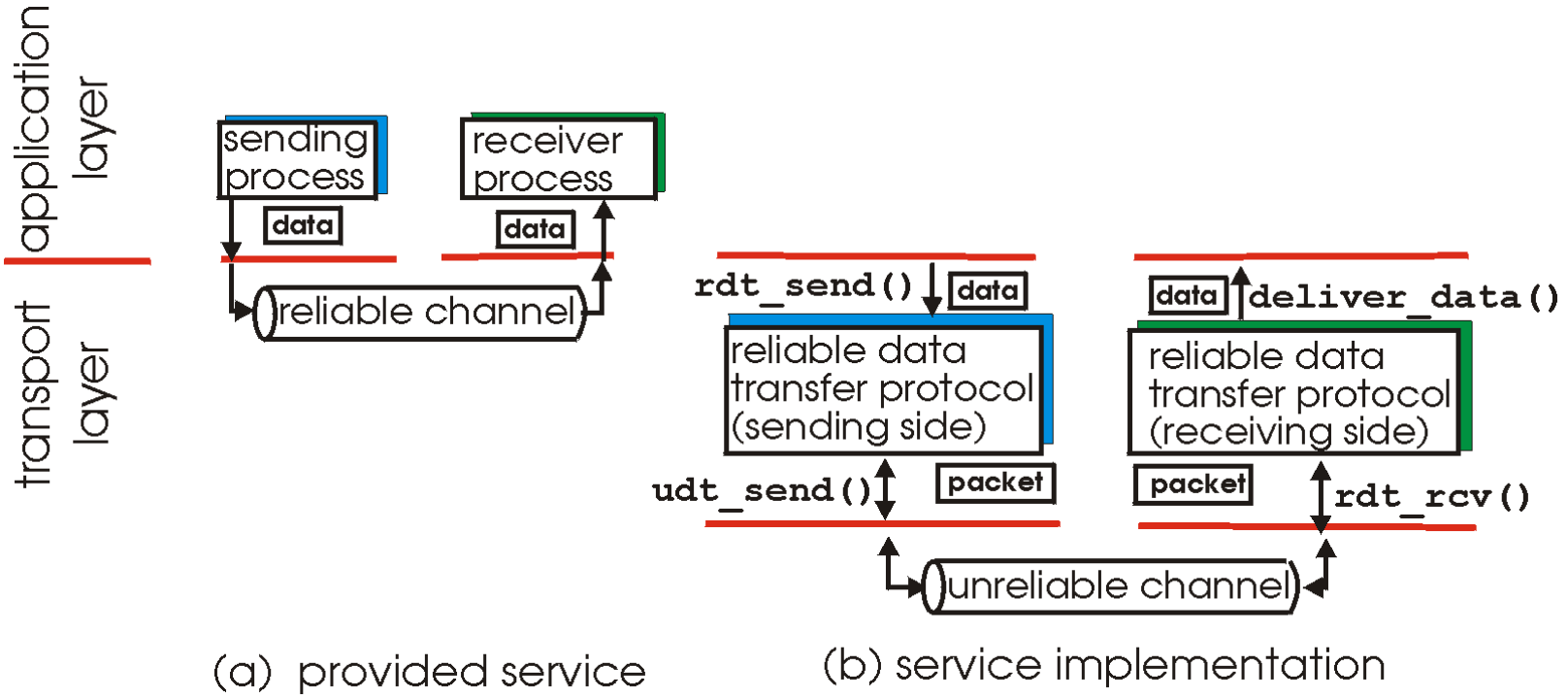
- important in application, transport, link layers
  - top-10 list of important networking topics!




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

# Principles of reliable data transfer

- important in application, transport, link layers
  - top-10 list of important networking topics!

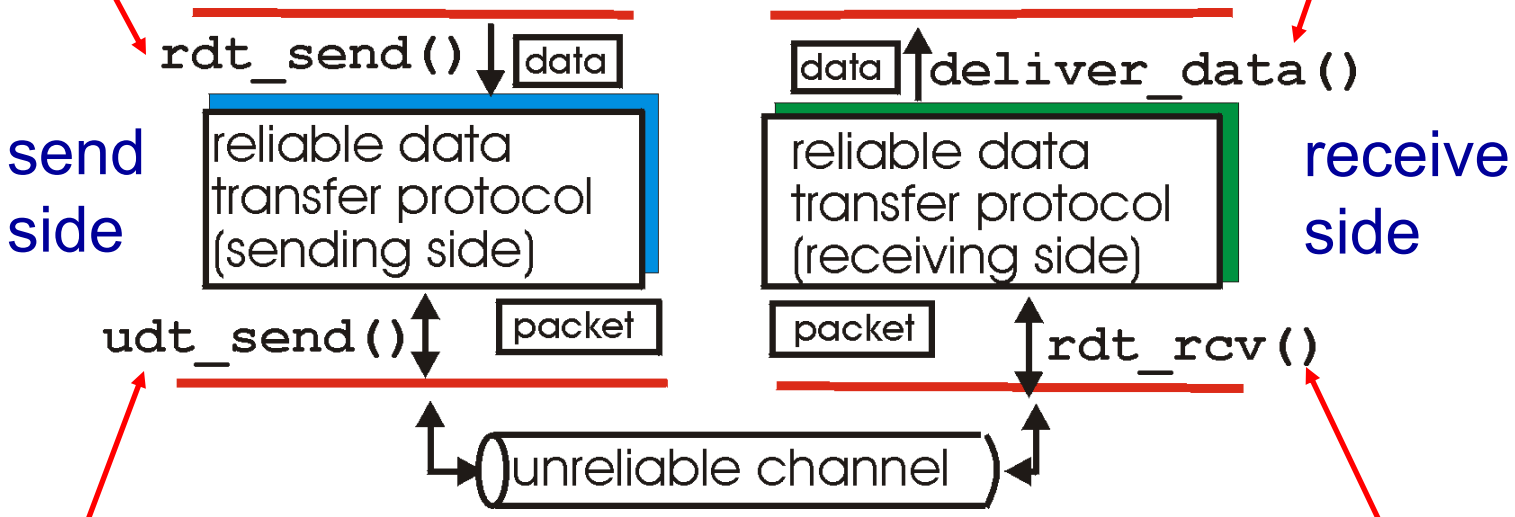



- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

# 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




**`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

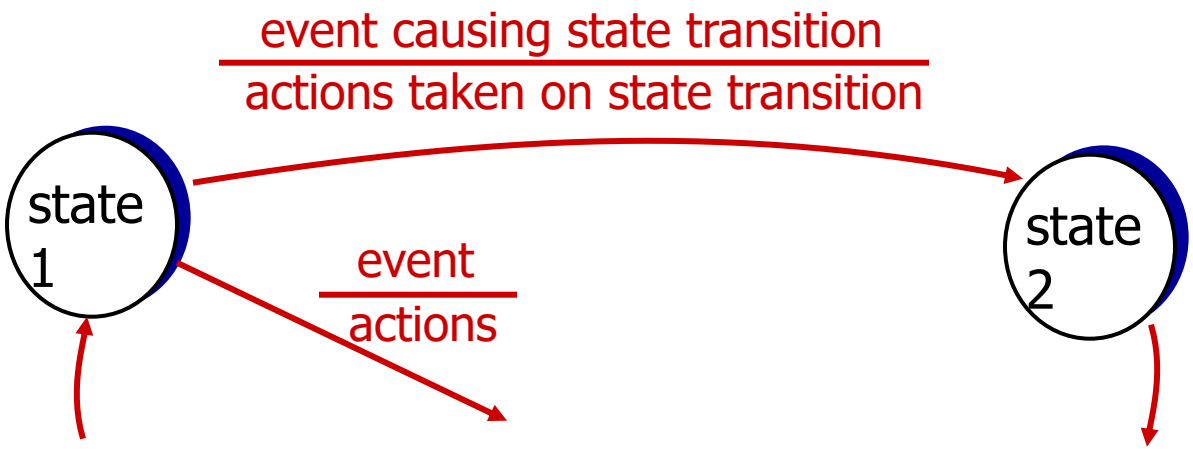
we'll:

- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow on both directions!

- THREE main issues in reliable data transfer:
  - Packet Corruption
  - Packet Loss
  - Packet reordering (NOT CONSIDERED)

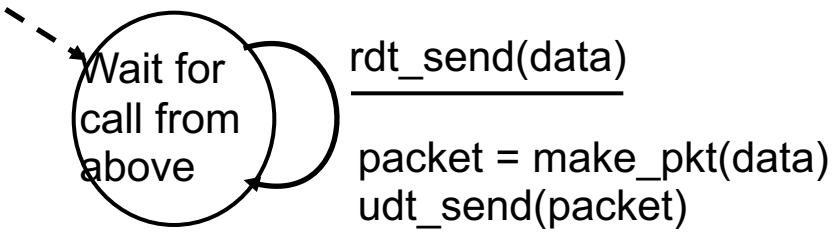
- use finite state machines (FSM) to specify sender, receiver

state: when in this "state" next state uniquely determined by next event

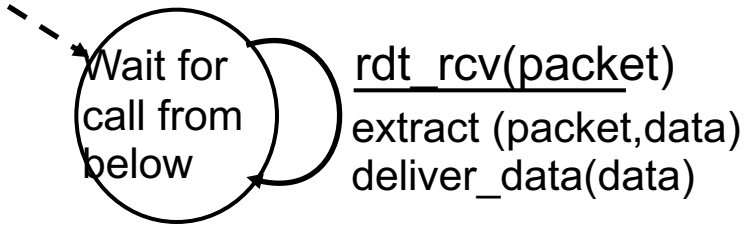


# rdt1.0: reliable transfer over a reliable channel

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




sender




receiver

# rdt2.0: channel with bit errors (still no packet loss)

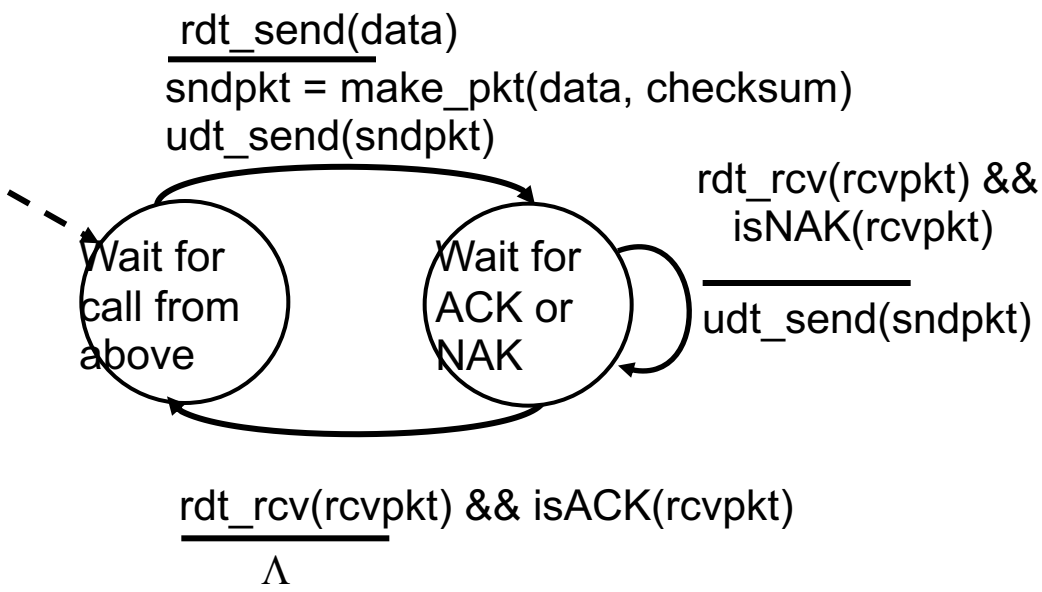
- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors:

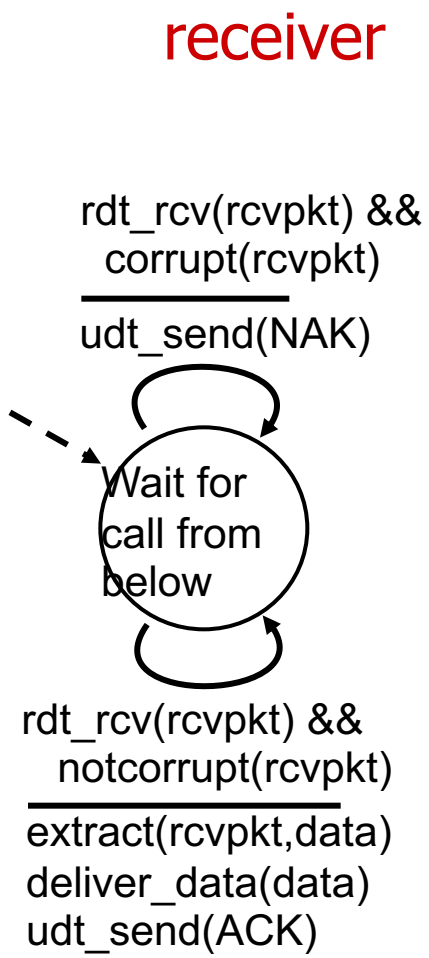
*How do humans recover from "errors" during conversation?*

# rdt2.0: channel with bit errors

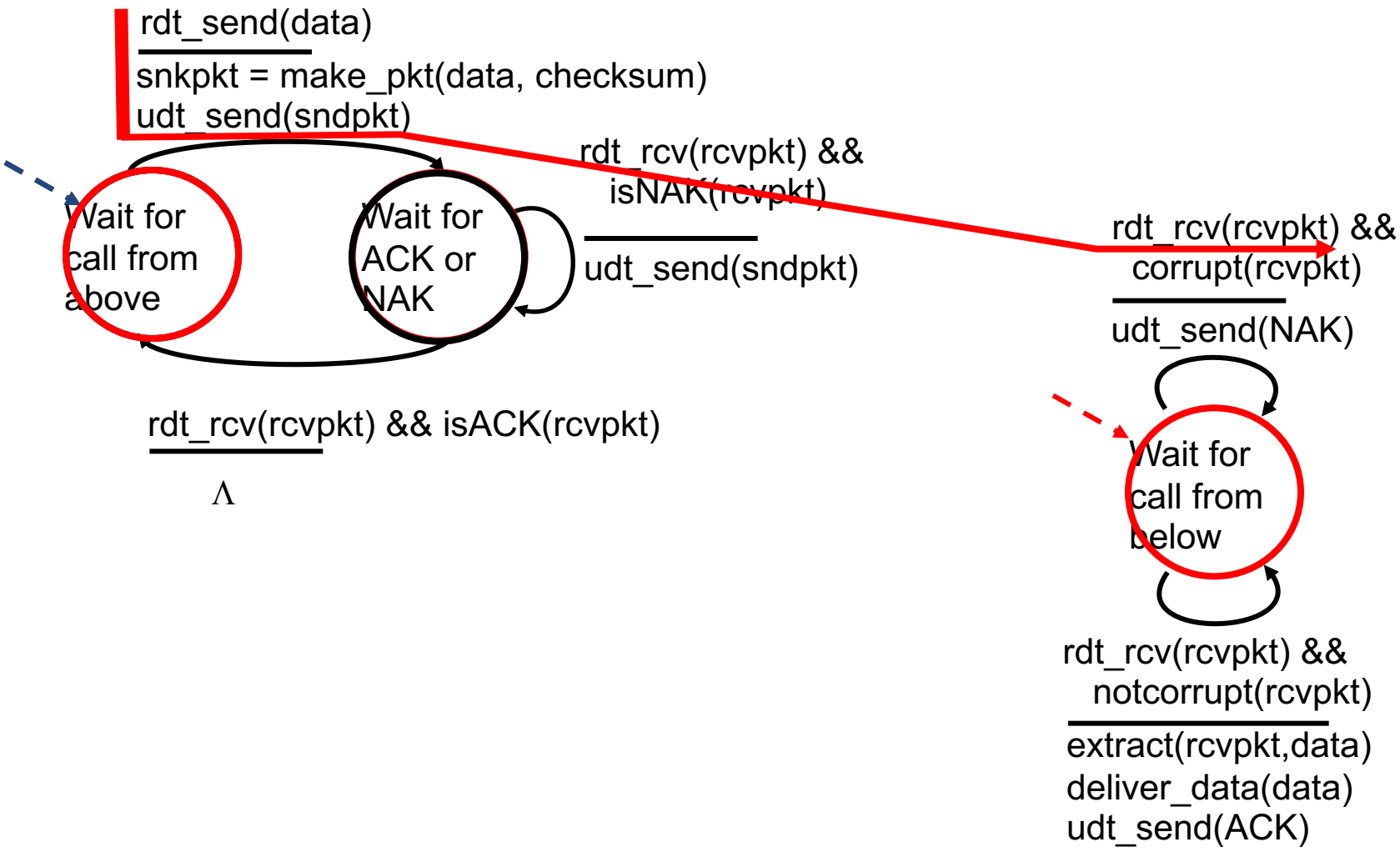
- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors:
  - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors
  - sender retransmits pkt on receipt of NAK
- new mechanisms in `rdt2.0` (beyond `rdt1.0`):
  - error detection
  - feedback: control msgs (ACK,NAK) from receiver to sender for every packet
  - retransmission of messages by sender for which NAK is received
- Called Automatic Repeat Request (ARQ) protocols

# rdt2.0: FSM specification

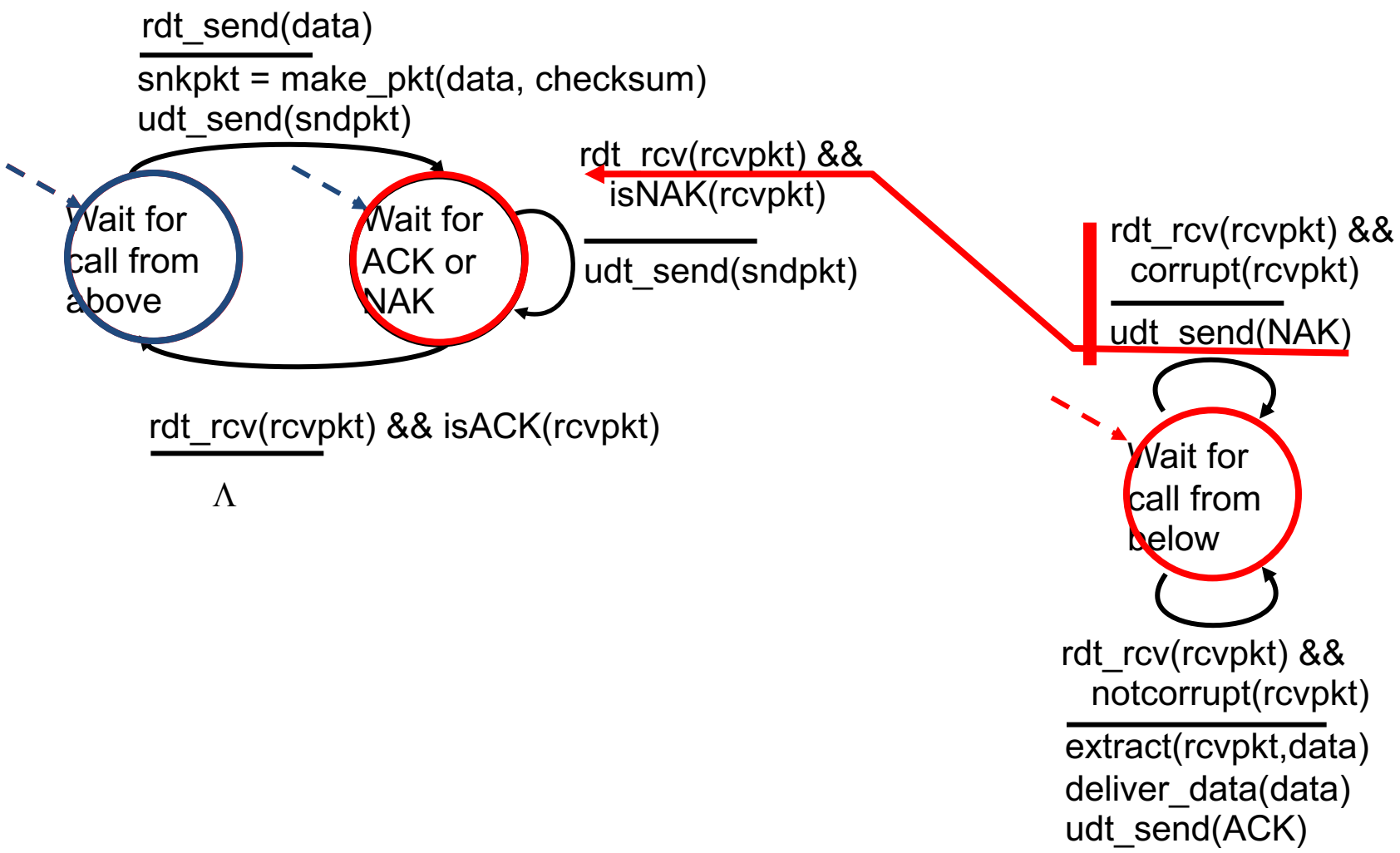


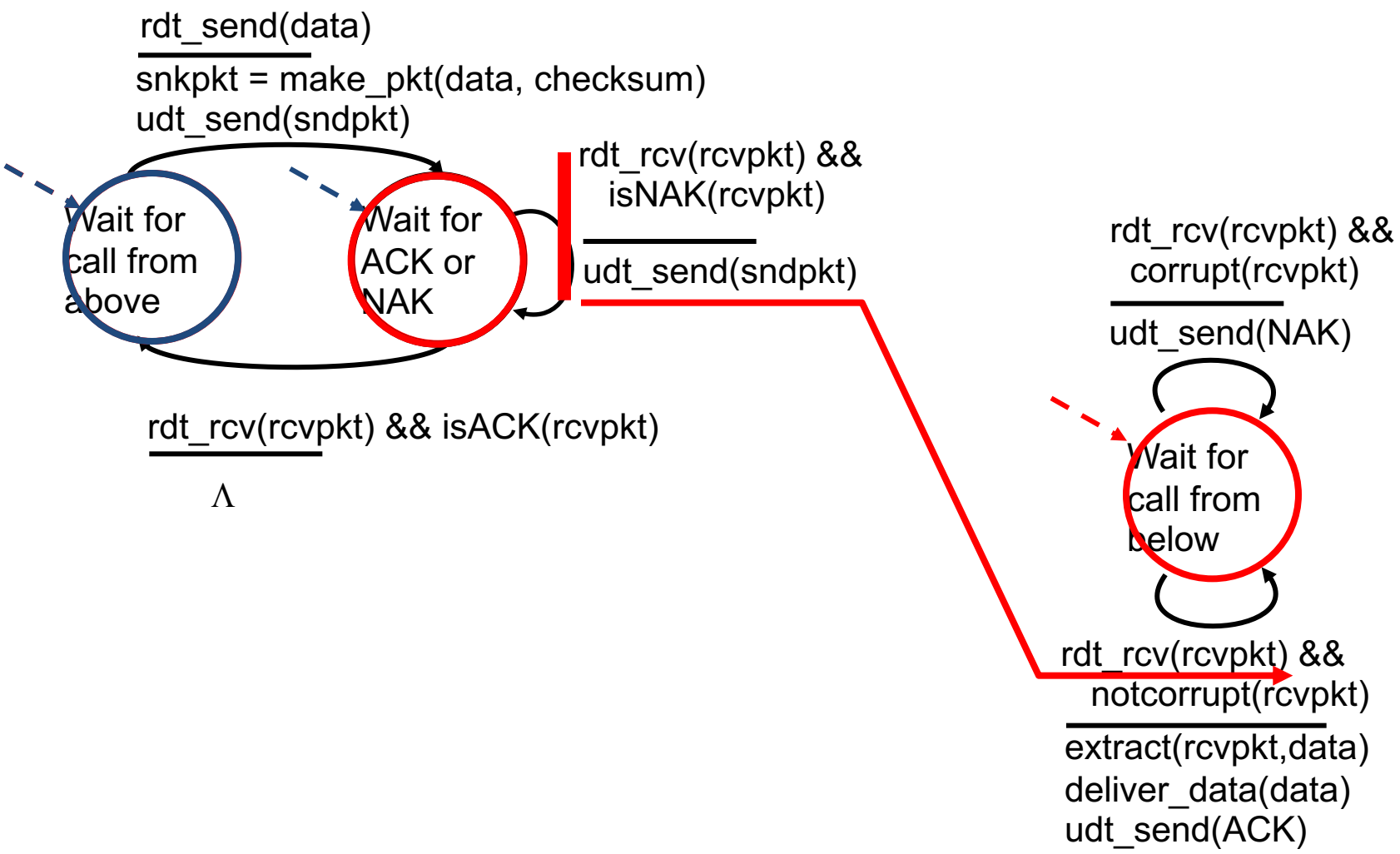
# rdt2.0: error scenario



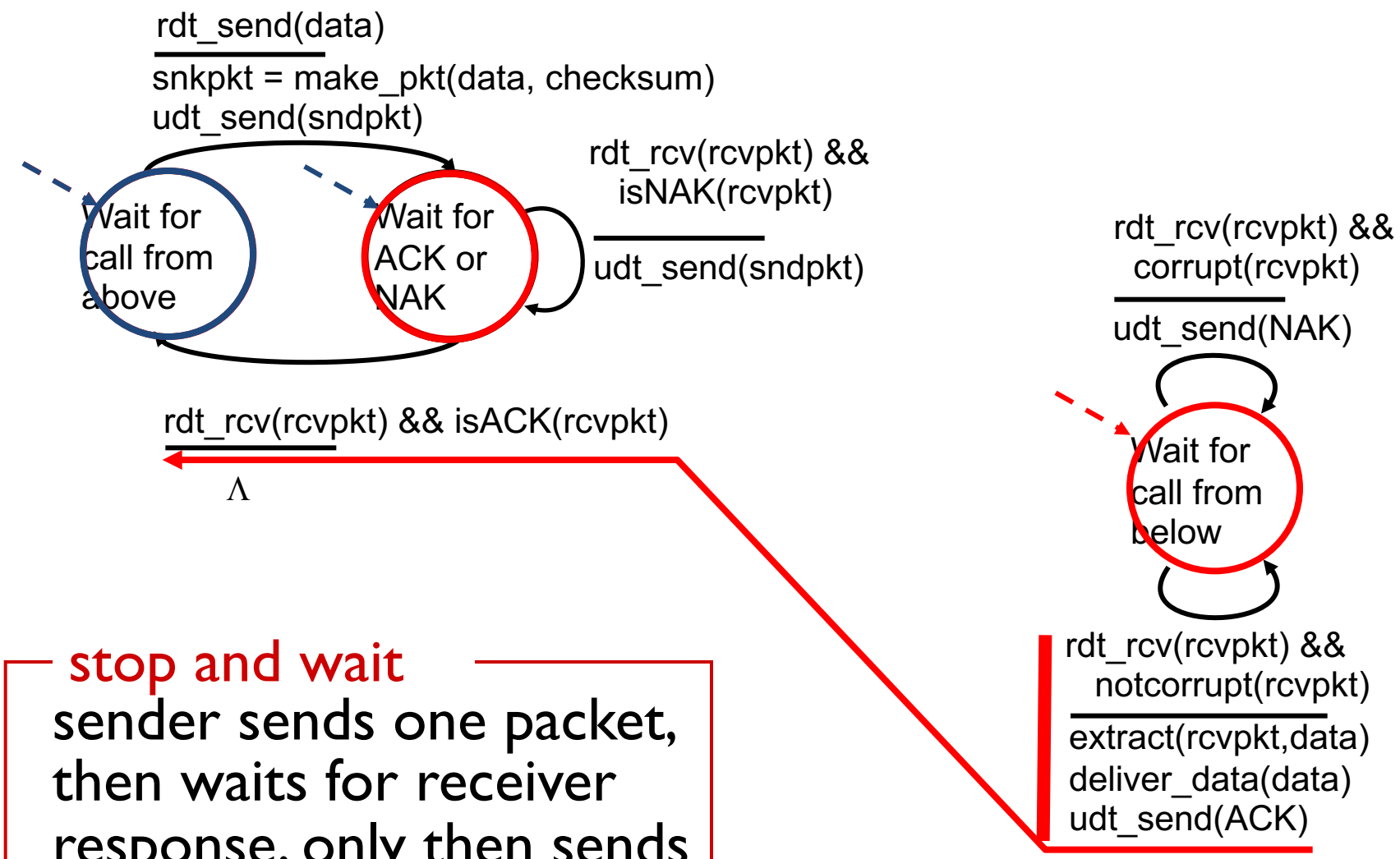
# rdt2.0: error scenario



# rdt2.0: error scenario



# rdt2.0: error scenario




**stop and wait**

sender sends one packet, then waits for receiver response, only then sends next packet

# 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

# 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

handling duplicates:

- sender retransmits current pkt if ACK/NAK corrupted
- sender adds *sequence number* to each pkt
- receiver discards (doesn't deliver up) duplicate pkt

# rdt2.1: dealing with ack/nak errors

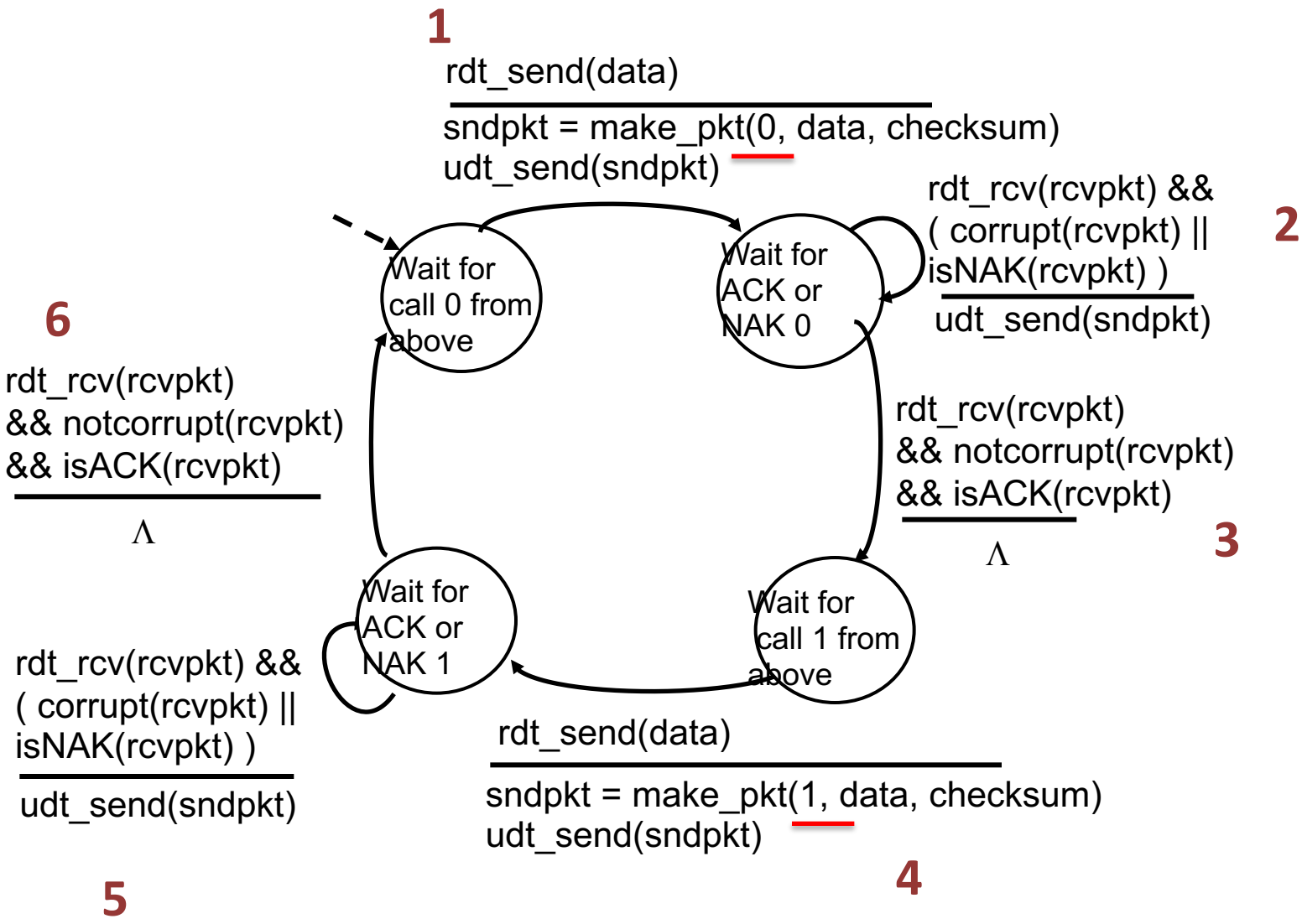
sender:

- 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 "expected" pkt should have seq # of 0 or 1

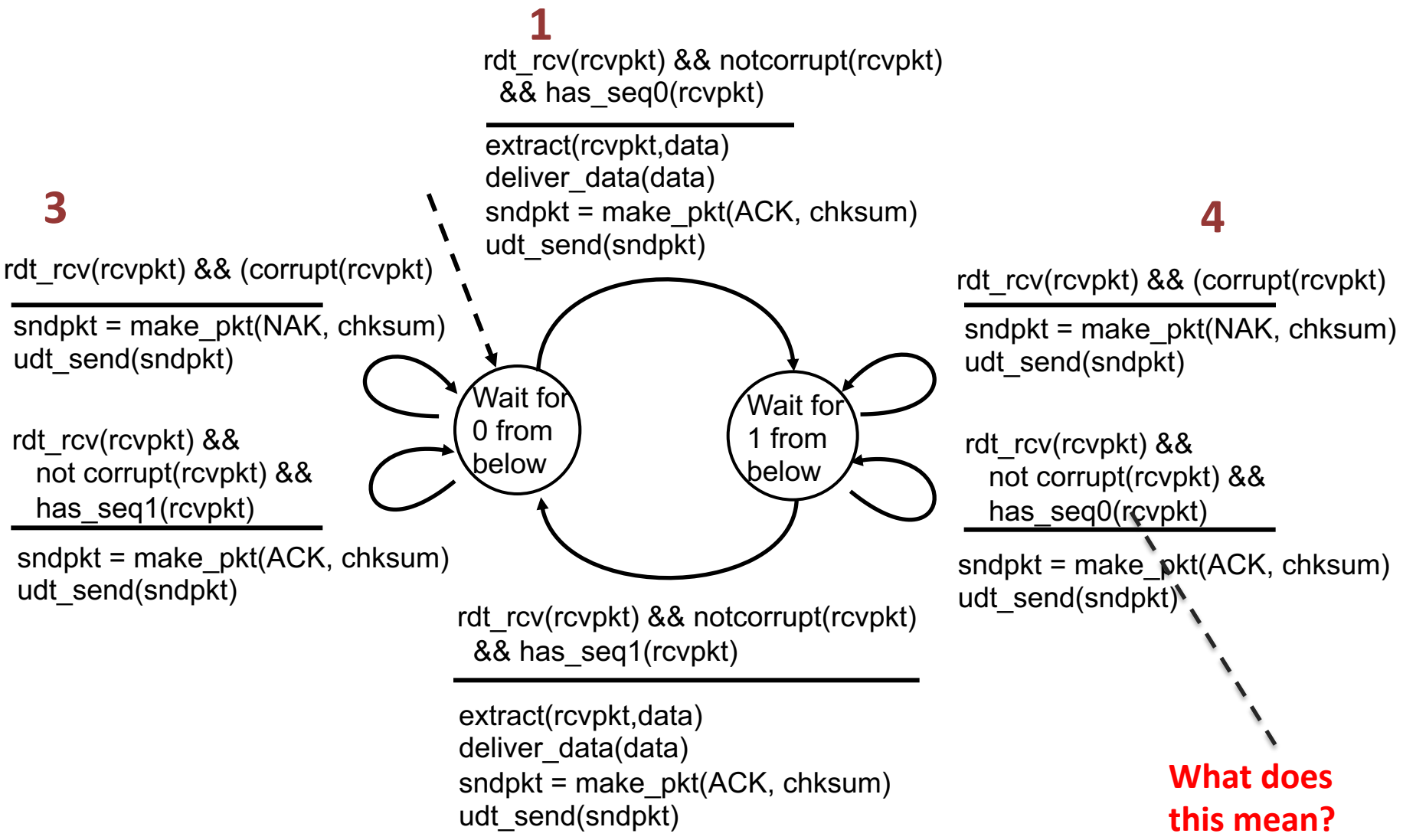
receiver:

- 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
  - Only way to know that is if the same packet # was received more than once

# rdt2.1: sender, handles garbled ACK/NAKs



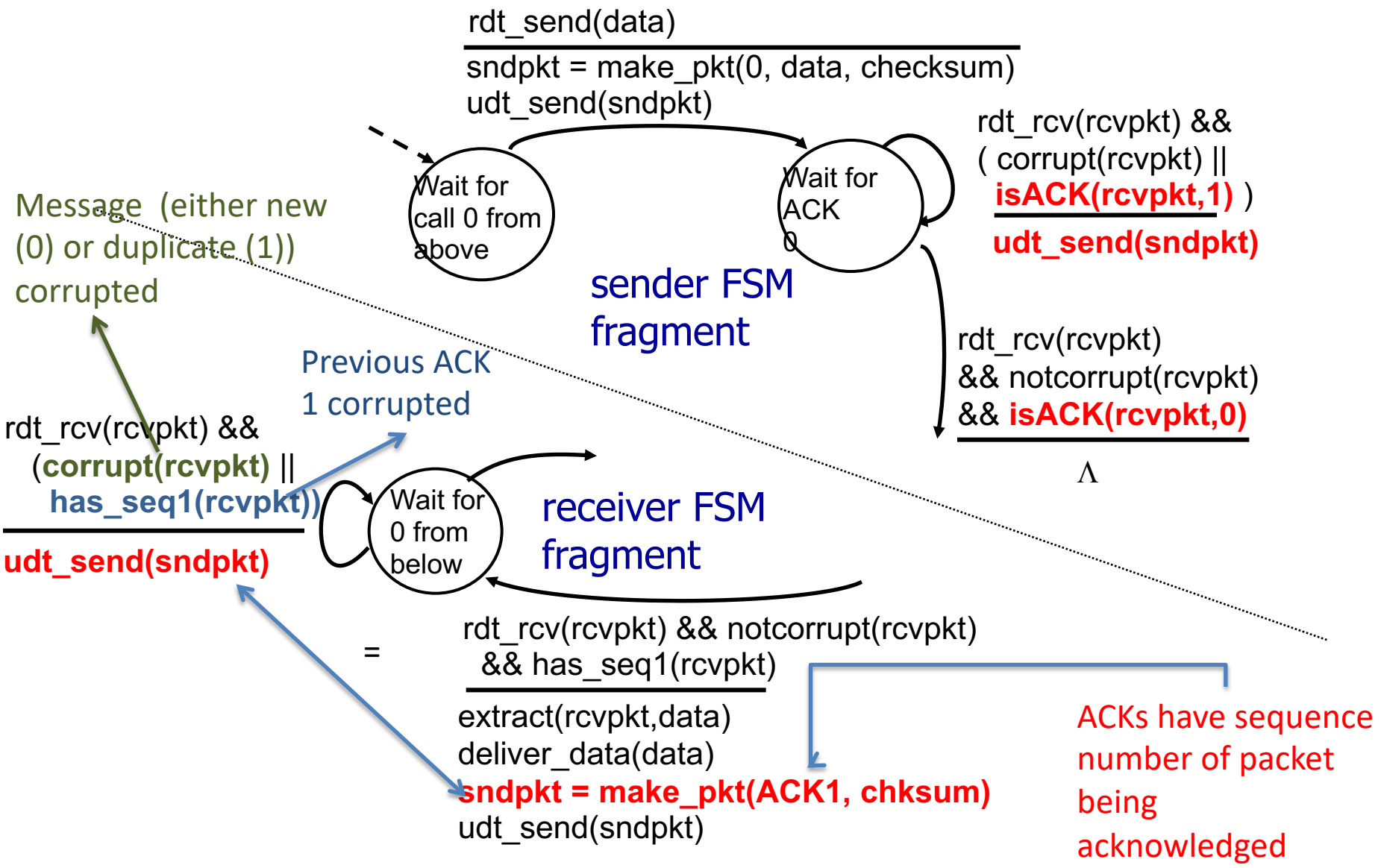
# rdt2.1: receiver, handles garbled ACK/NAKs



# rdt2.2: a NAK-free (ACK-only) protocol

- same functionality as rdt2.1, using ACKs 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, receiver fragments



# rdt3.0: channels with errors *and* loss/delay

new assumption:

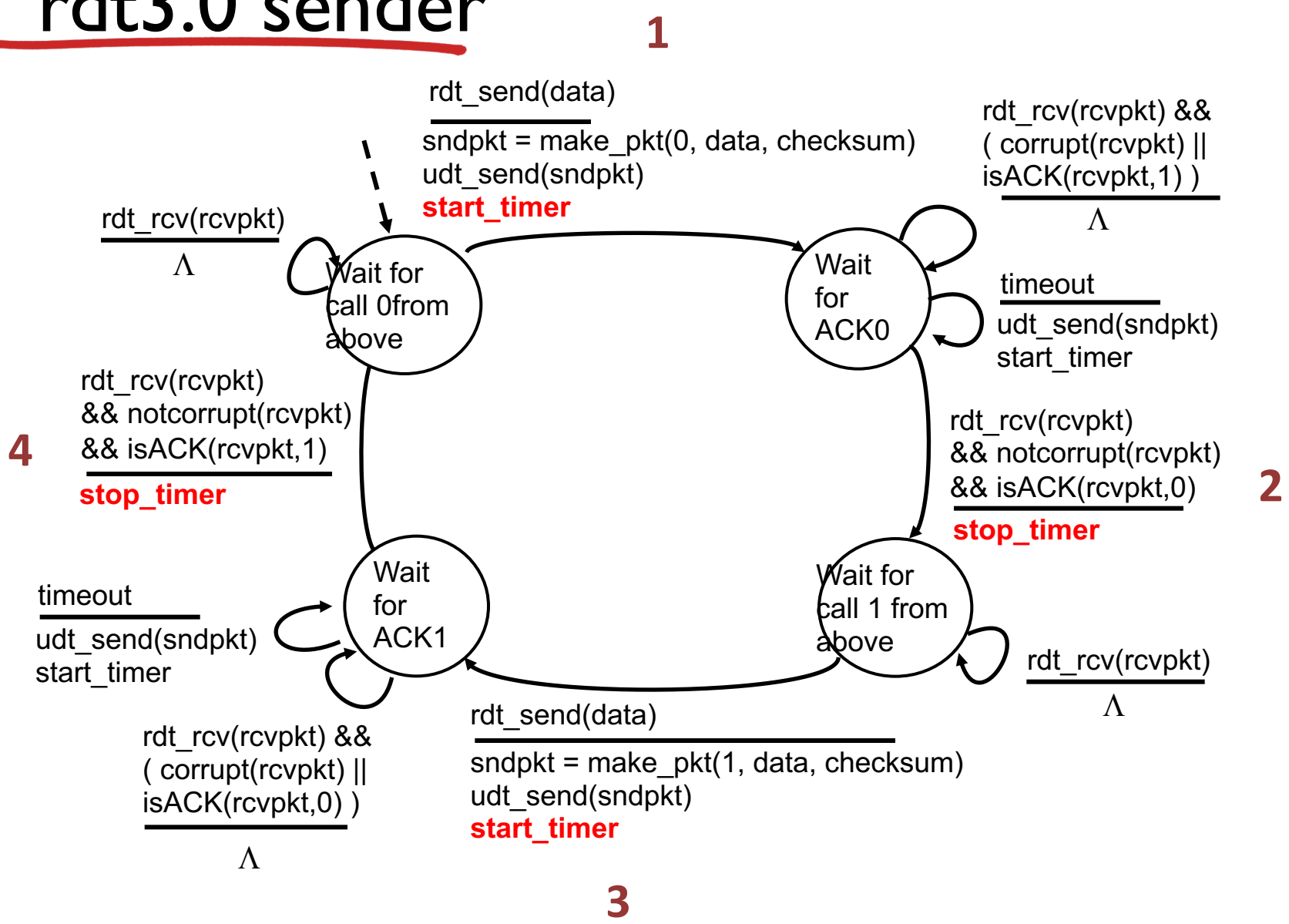
underlying channel can also lose packets (data, ACKs)

- checksum, seq. #, ACKs, retransmissions will be of help … but not enough

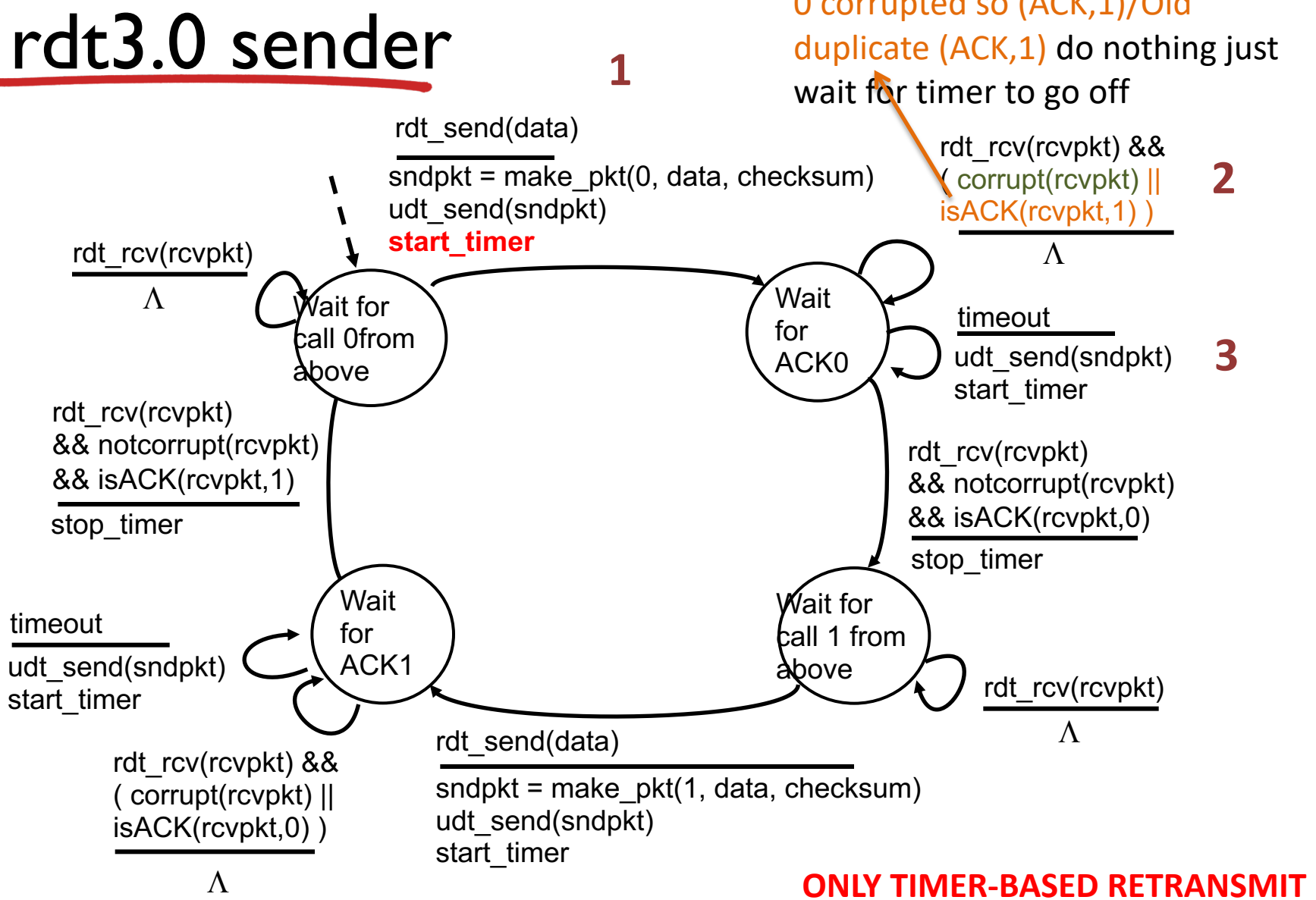
approach: sender waits "reasonable" amount of time for ACK

- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but seq. #'s already handles this
  - receiver must specify seq # of pkt being ACKed
- requires countdown timer
  - **For each PACKET**

# rdt3.0 sender



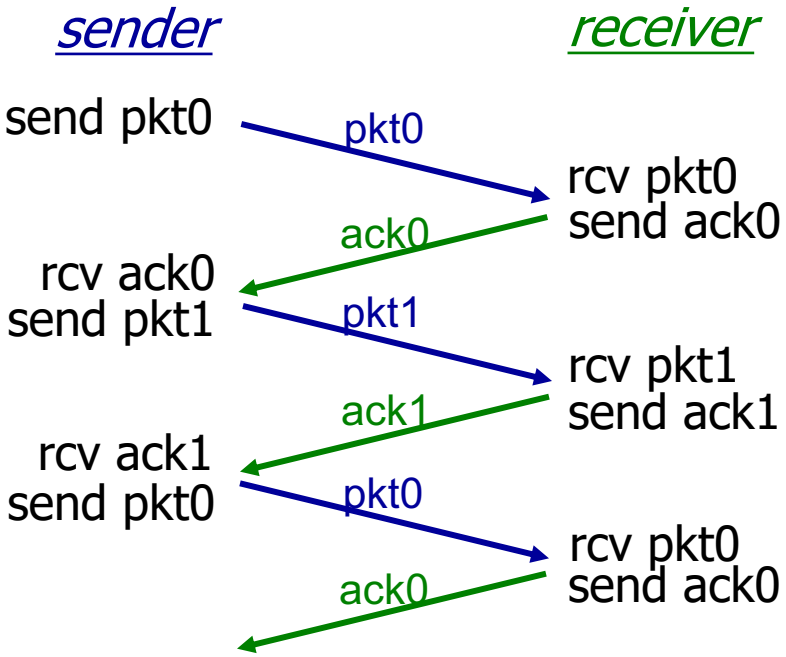
# rdt3.0 sender



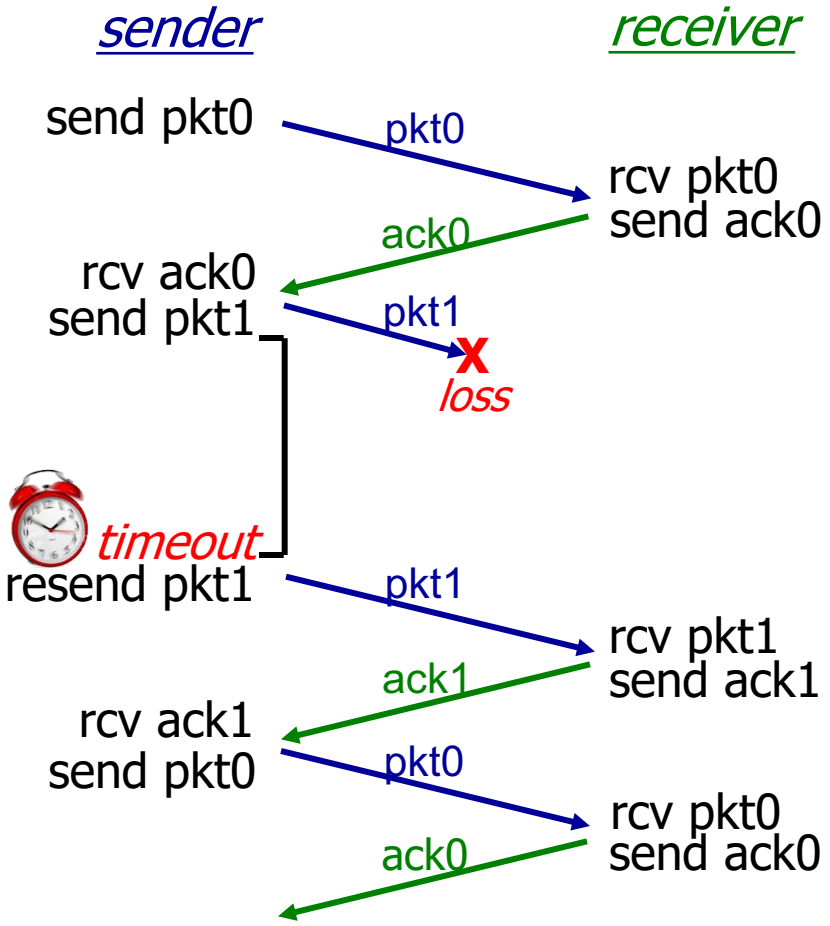

ACK 0, ACK 1 corrupted OR Packet 0 corrupted so (ACK,1)/Old duplicate (ACK,1) do nothing just wait for timer to go off

**ONLY TIMER-BASED RETRANSMIT**
**Unlike rdt 2.2**

# rdt3.0 in action

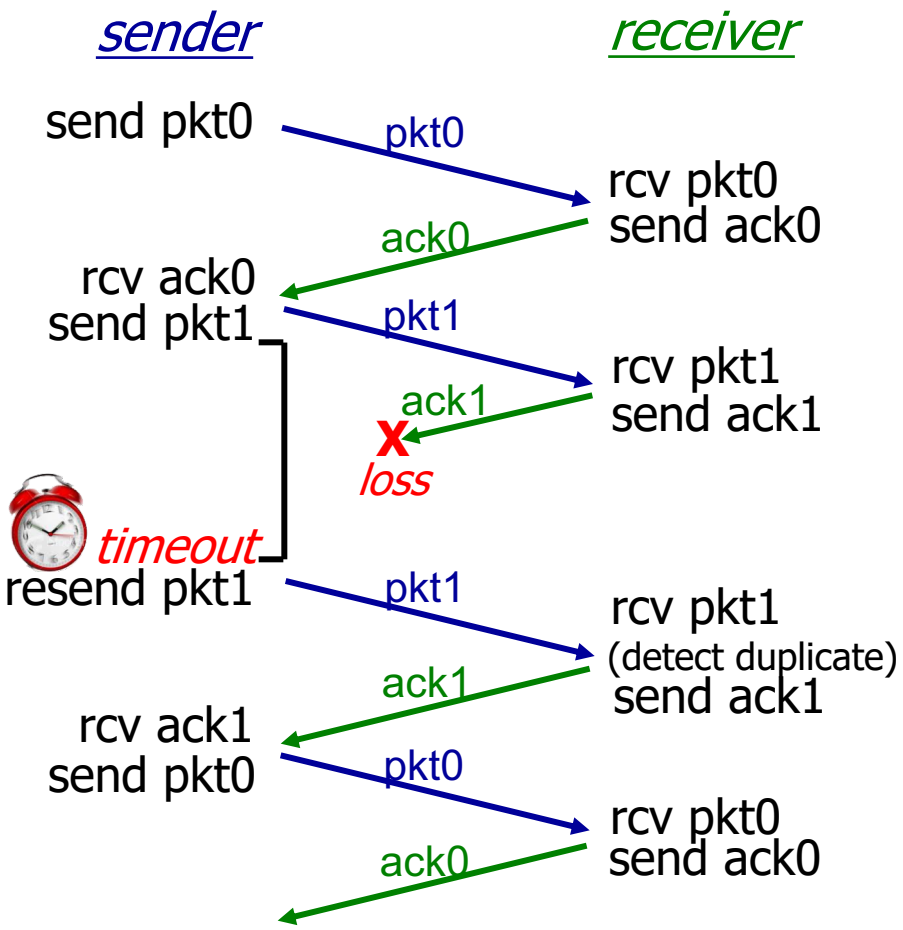



(a) no loss

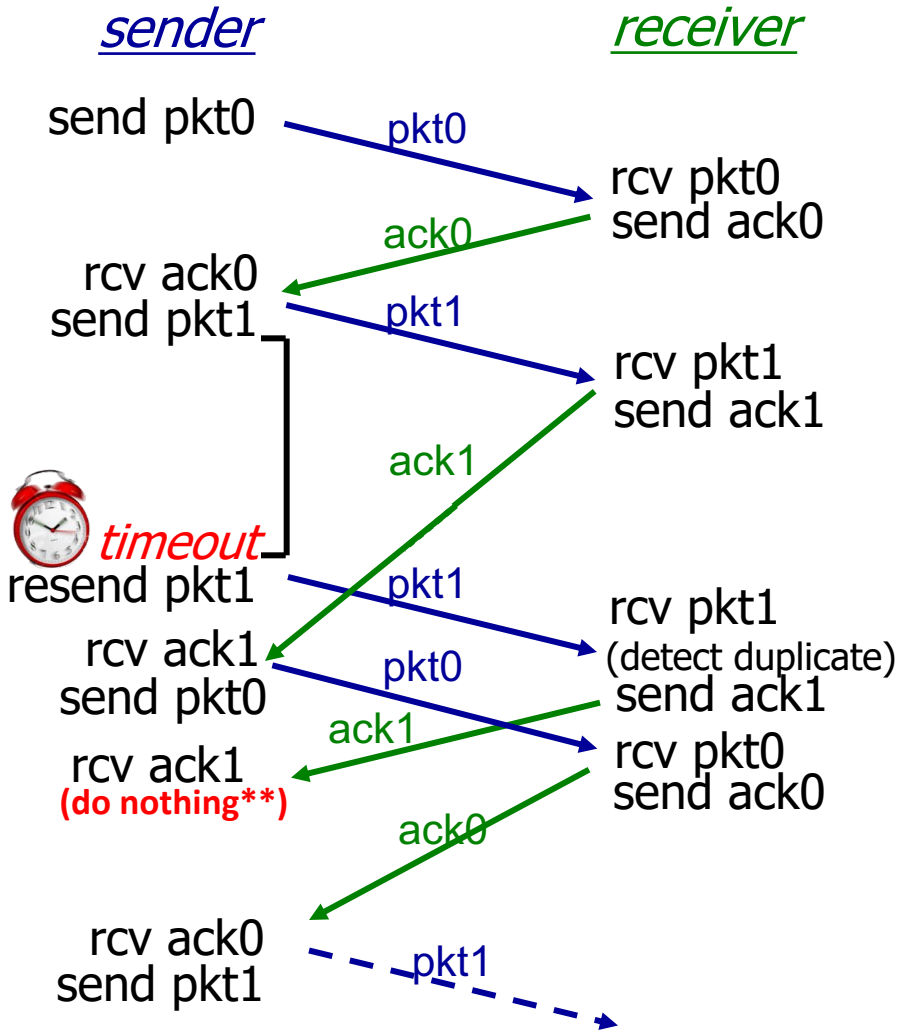



(b) packet loss

# rdt3.0 in action




(c) ACK loss




(d) premature timeout/ delayed ACK

**Alternating-Bit Protocol**

# Performance of rdt3.0

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

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

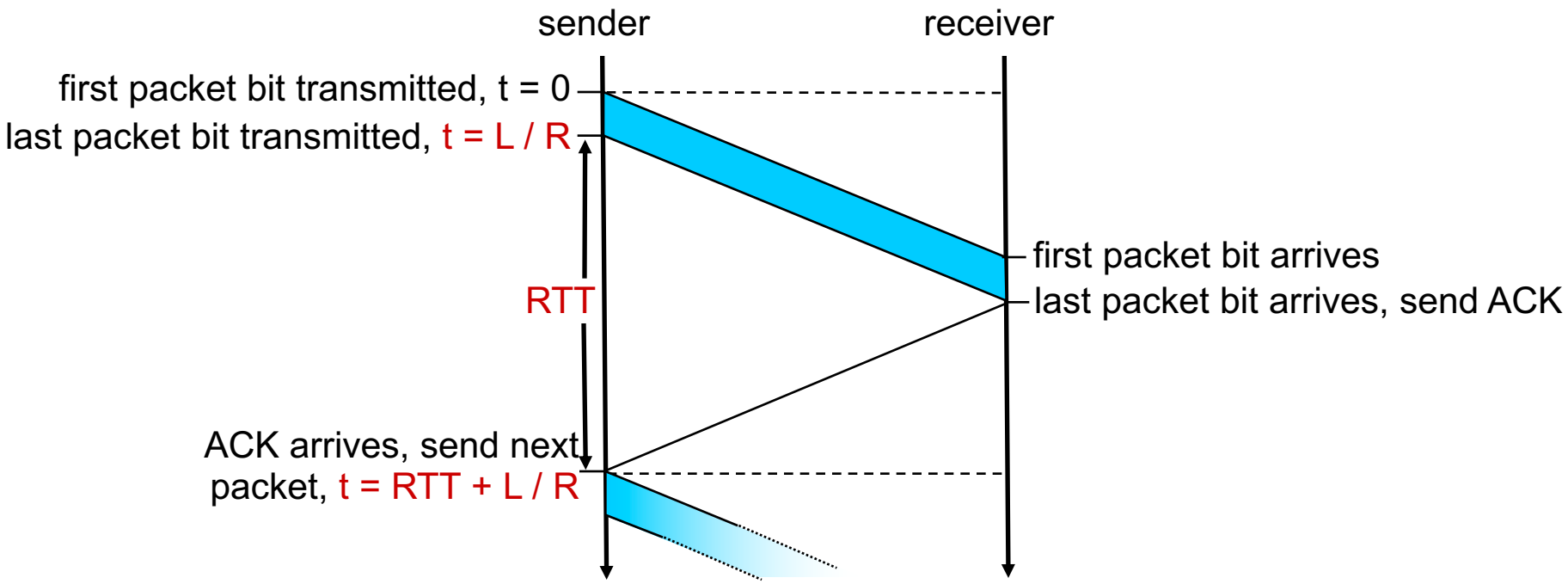
  - $U_{sender}$: *utilization* – fraction of time sender busy sending

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

  - 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



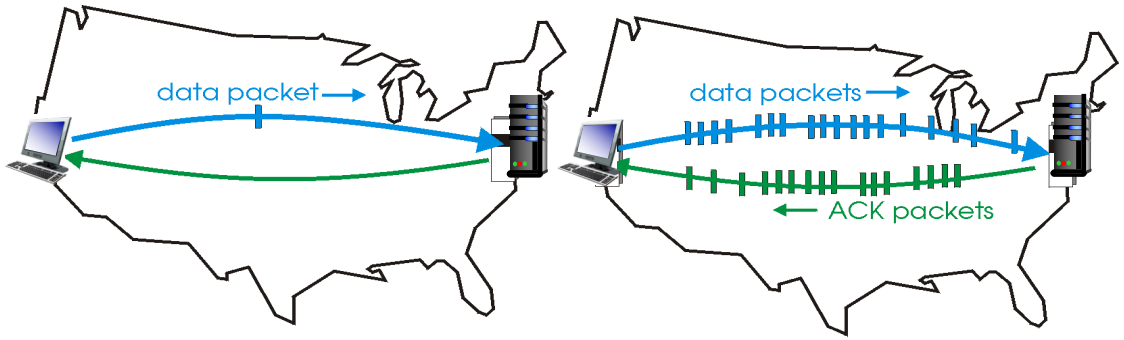

$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

How do we fix this ?

# Pipelined protocols

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

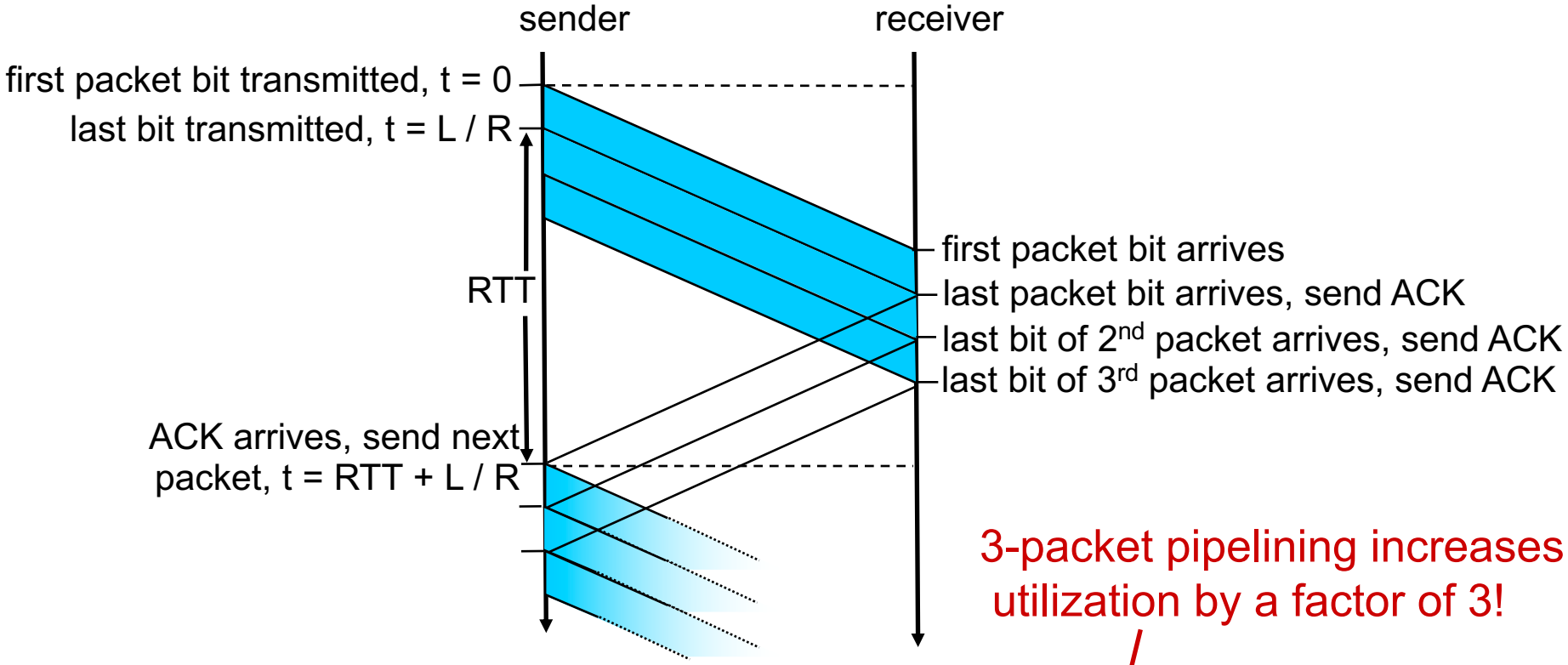
- range of sequence numbers must be increased
- buffering at sender and/or receiver




(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




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

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

# Pipelined protocols: overview

Go-back-N:

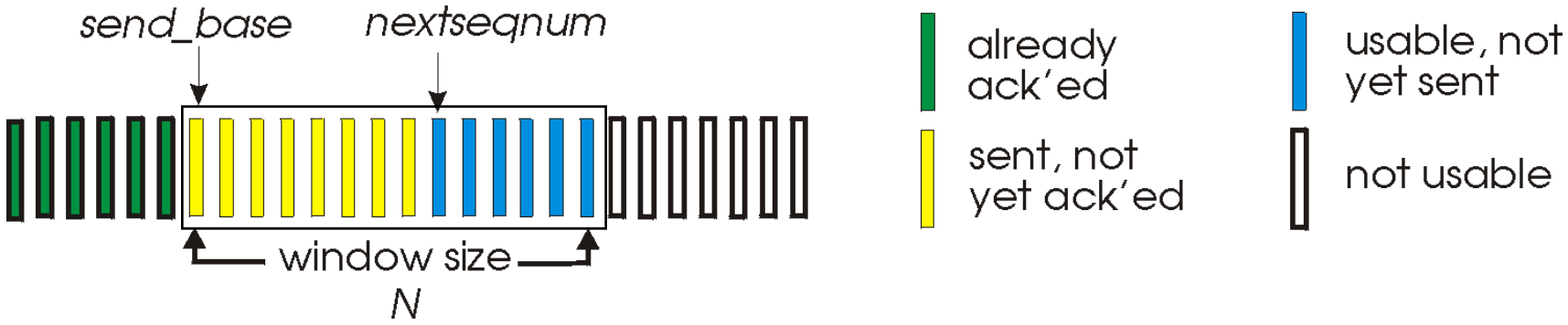
- sender can have up to N unacked packets in pipeline
- receiver only sends *cumulative ack*
  - doesn't ack packet if there's a gap
- sender has timer for oldest unacked packet
  - when timer expires, retransmit *all* unacked packets

Selective Repeat:

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

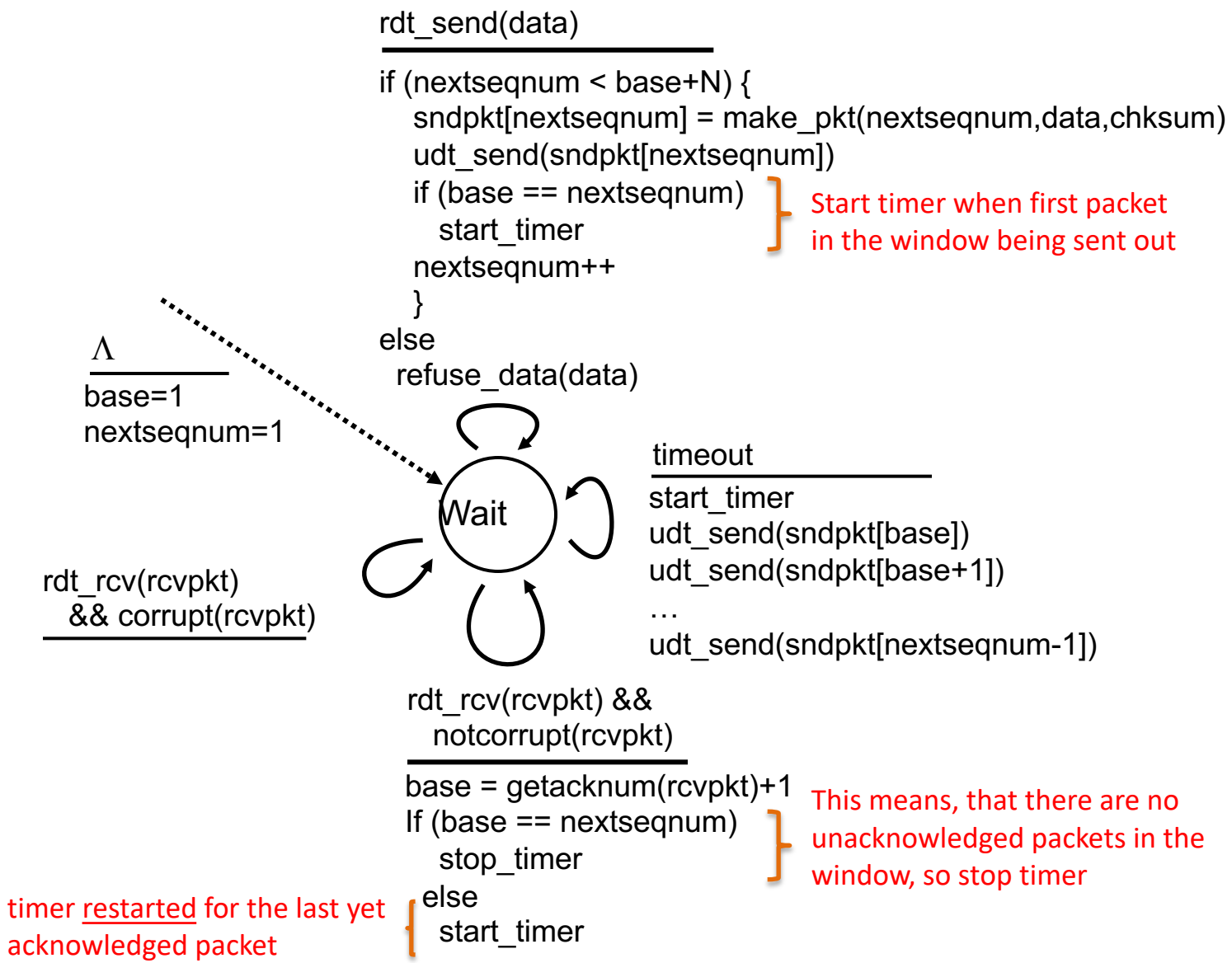
# Go-Back-N: sender

- k-bit seq # in pkt header (modulo $2^k$ arithmetic for seq #)
- "window" of up to N, consecutive unack'ed pkts allowed

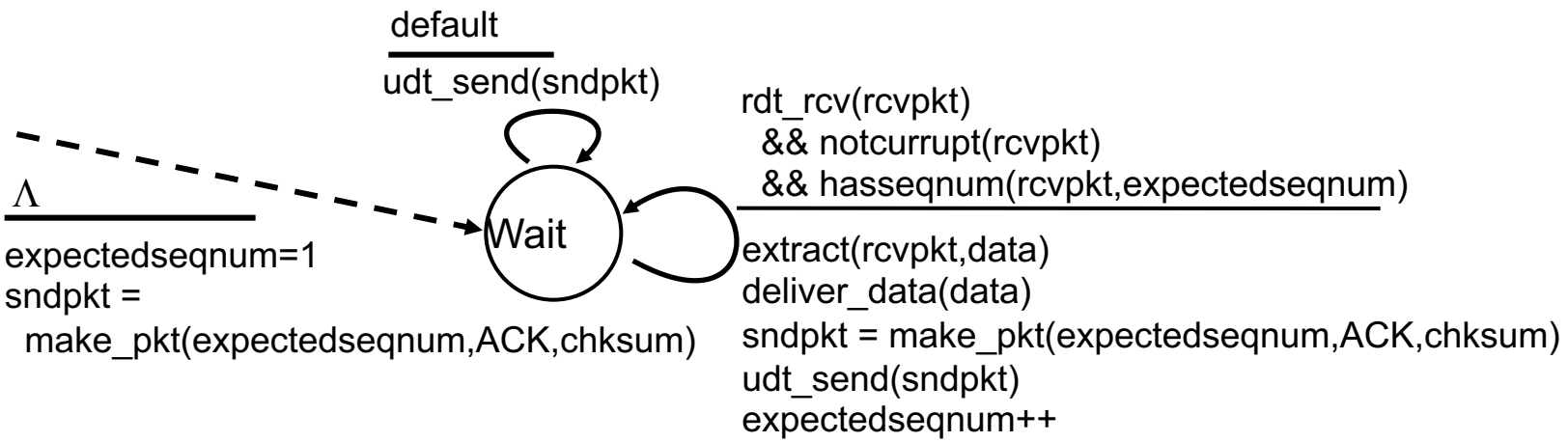



- ACK(n): ACKs all pkts up to, including seq # n - *"cumulative ACK"*
  - may receive duplicate ACKs (see receiver)
- timer for oldest in-flight pkt
- *timeout(n):* retransmit all packets that have been sent but not acknowledged yet

# GBN: sender extended FSM

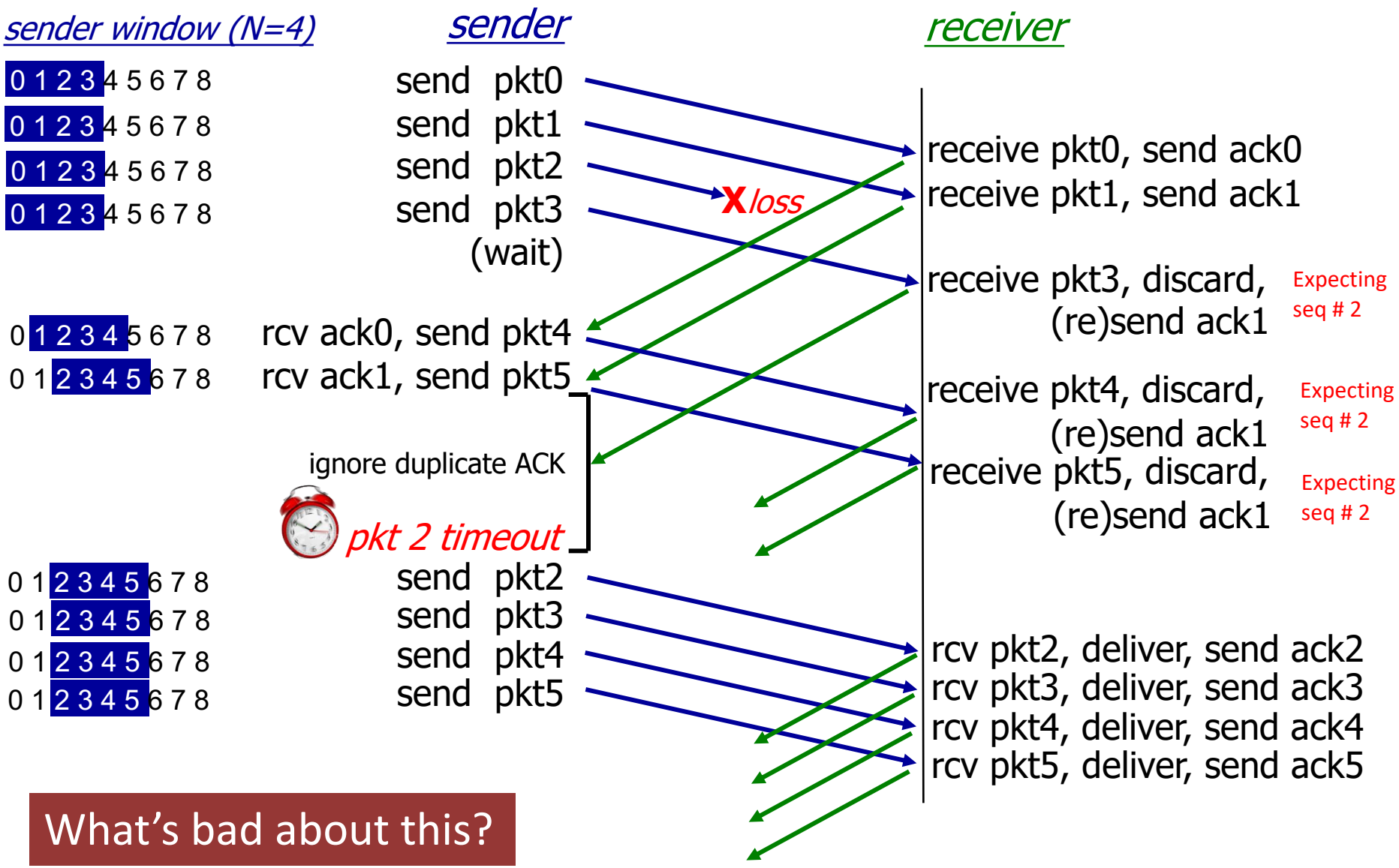


# GBN: receiver extended FSM




- Maintain variable **`expectedseqnum`**
- always send ACK if packet received with expected sequence number
- out-of-order pkt:
  - discard (don't buffer): *no receiver buffering! --- WHY?*
  - re-ACK last in order seq # packet
  - may generate duplicate ACKs

# GBN in action




What's bad about this?


That's all Folks!