Lecture 10: Transpor Layer
Principles of Reliable Data Transfer

COMP 332, Spring 2018

Victoria Manfredi

WESLEYAN
UNIVERSITY

Acknowledgements: materials adapted from Computer Networking: A Top Down Approach 7th edition: ©1996-2016, J.F Kurose and K.W. Ross, All Rights Reserved as well as from slides by Abraham Matta at Boston University, and some material from Computer Networks by Tannenbaum and Wetherall.

Today

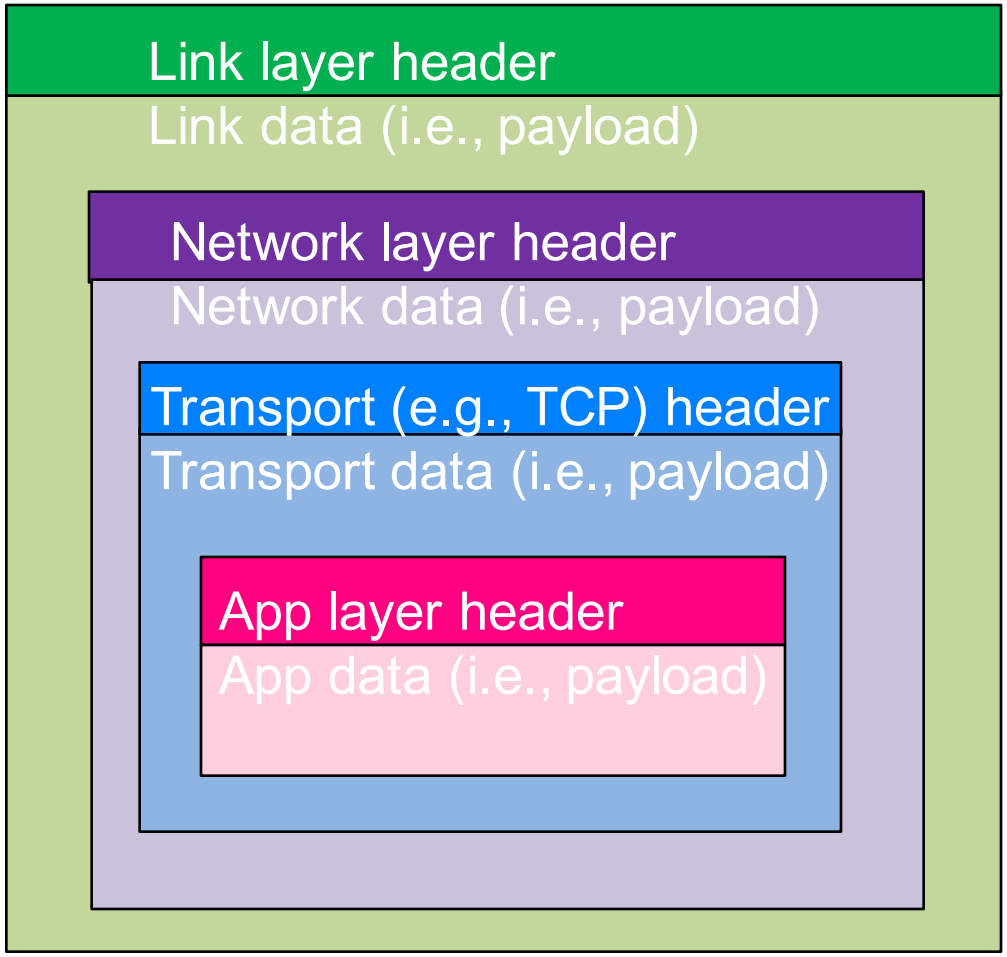
1. Announcements
   - homework 4 extension until Thursday at 11:59p
     - comments on how coding will be graded
   - homework 5 posted
2. Headers and payloads
   - recap
3. Reliable data transport
   - NAKs and ACKs
   - coping with garbled ACKs and NAKs
   - NAK-free protocol
   - channels with errors and loss
   - pipelined protocols
     - go-back-N and selective repeat
   - seq #s in practice

Why are we looking at?
To help you understand why TCP operates the way it does (we'll cover next week)

Headers and Payloads

RECAP

Headers and payloads



Each layer only looks at the header associated with that layer

Reliable Data Transport

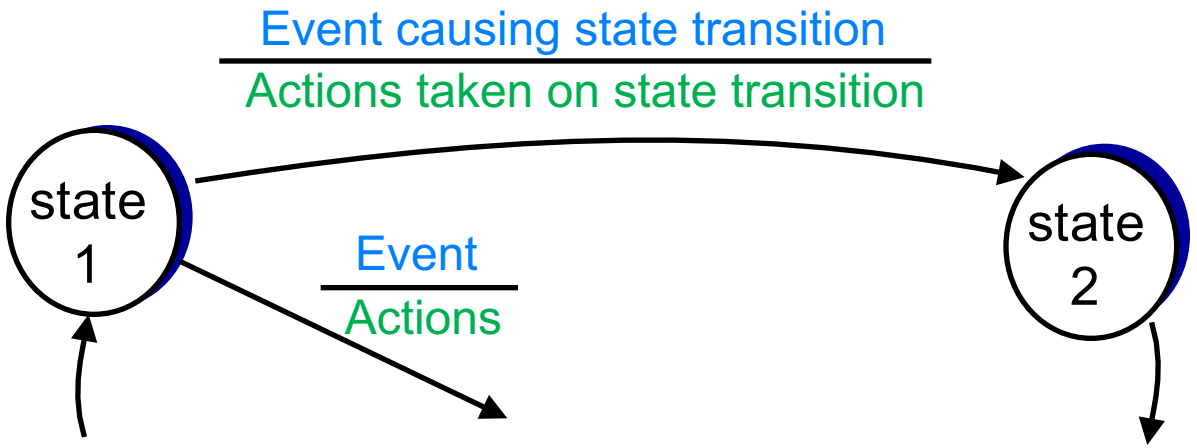
PRINCIPLES

Reliable data transfer: getting started

Our plan

- incrementally develop
  - sender, receiver sides of **r**eliable **d**ata **t**ransfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow in both directions!
- use finite state machines (FSM) to specify sender, receiver

State: when in this state, next state is uniquely determined by next event



Reliable Data Transport

PROTOCOL V1.0

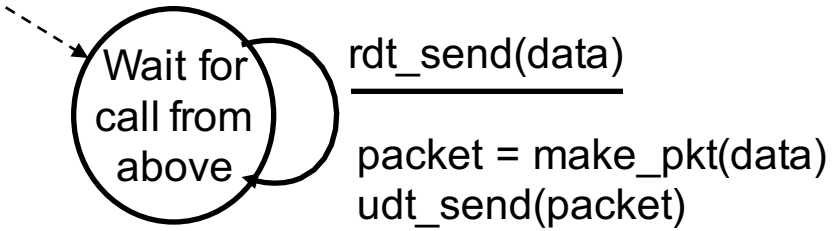
rdt1.0: reliable transfer over a reliable channel

Underlying channel perfectly reliable

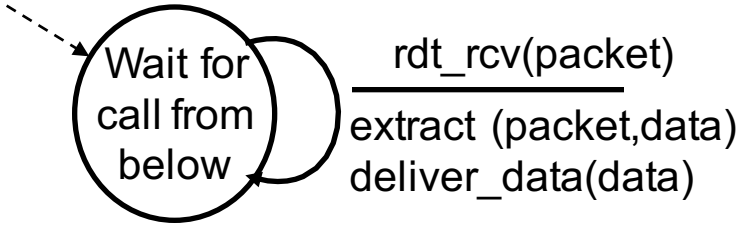
- no bit errors
- no loss of packets

Separate FSMs for sender, receiver:

- sender sends data into underlying channel
- receiver reads data from underlying channel



sender



receiver

Unreliable data transfer protocol would look the same

Reliable Data Transport

ACKS AND NAKS

rdt2.0: channel with bit errors

Underlying channel may flip bits in packet

- checksum to detect bit errors
- Q: how to recover from errors?

How do humans recover from “errors” during conversation?

rdt2.0: channel with bit errors

Problem: underlying channel may flip bits in packet

- how to detect and recover from errors?

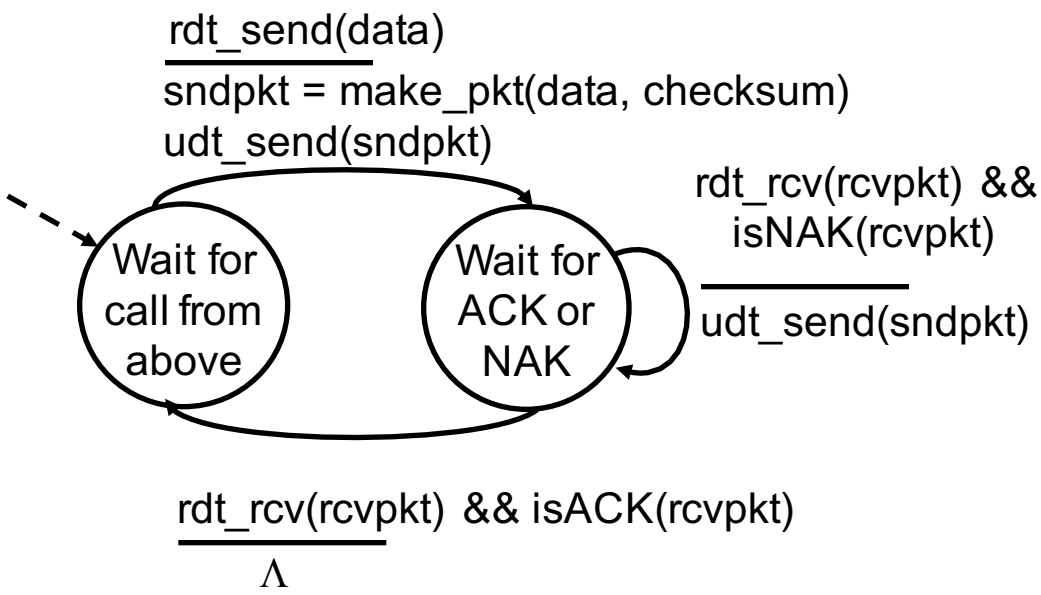
Solution

- Checksum
  - to detect bit 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

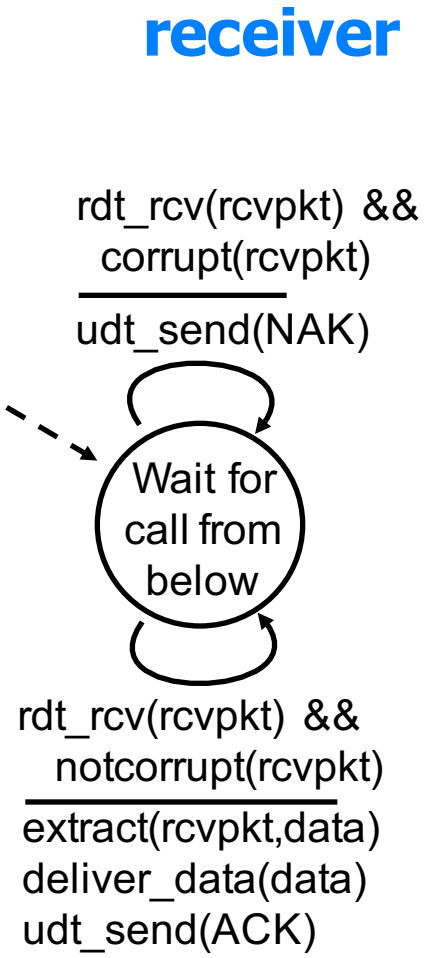
New mechanisms in rdt2.0 (beyond rdt1.0)

- error detection
- feedback: control msgs (ACK,NAK) from receiver to sender

rdt2.0: FSM specification



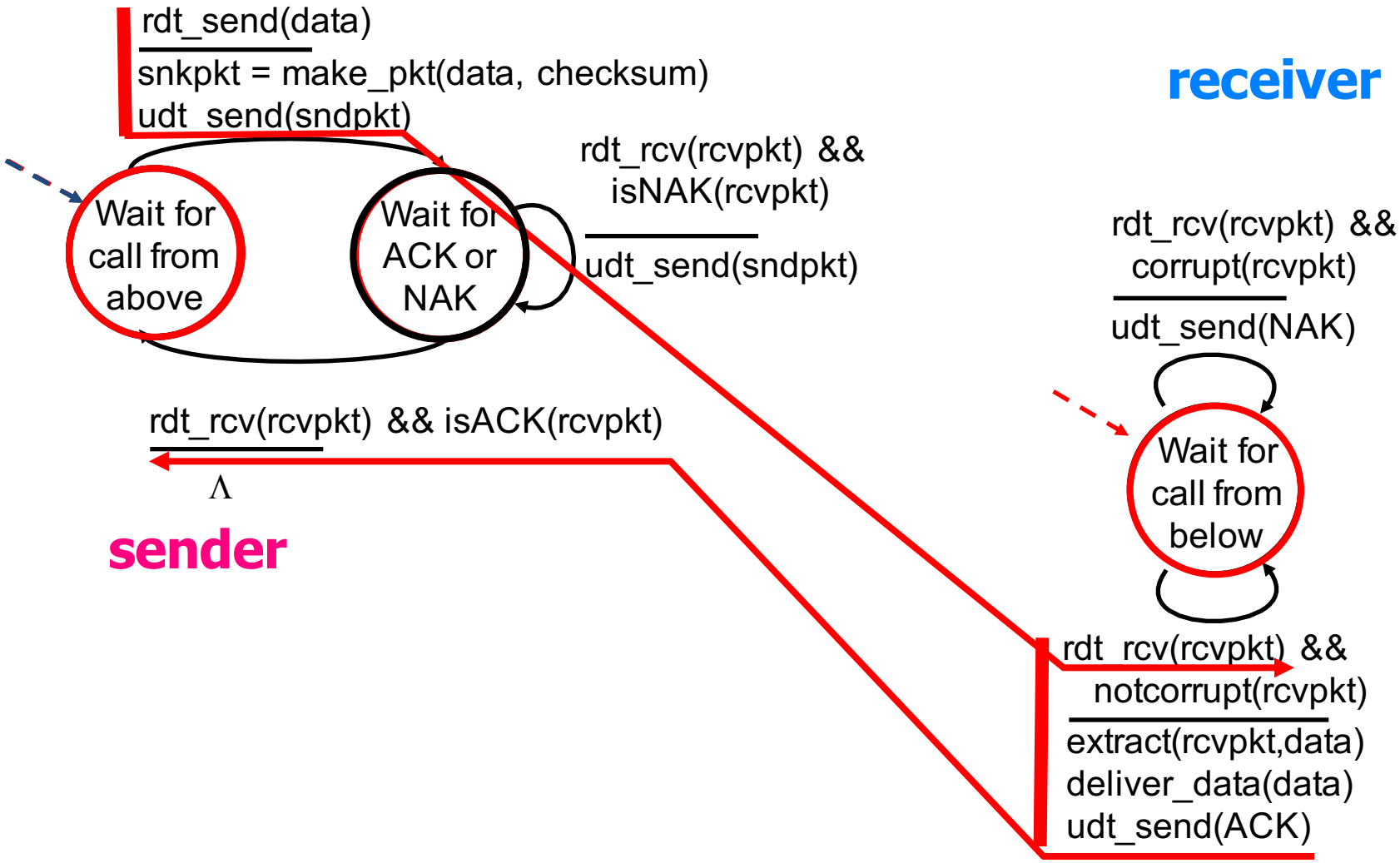
sender



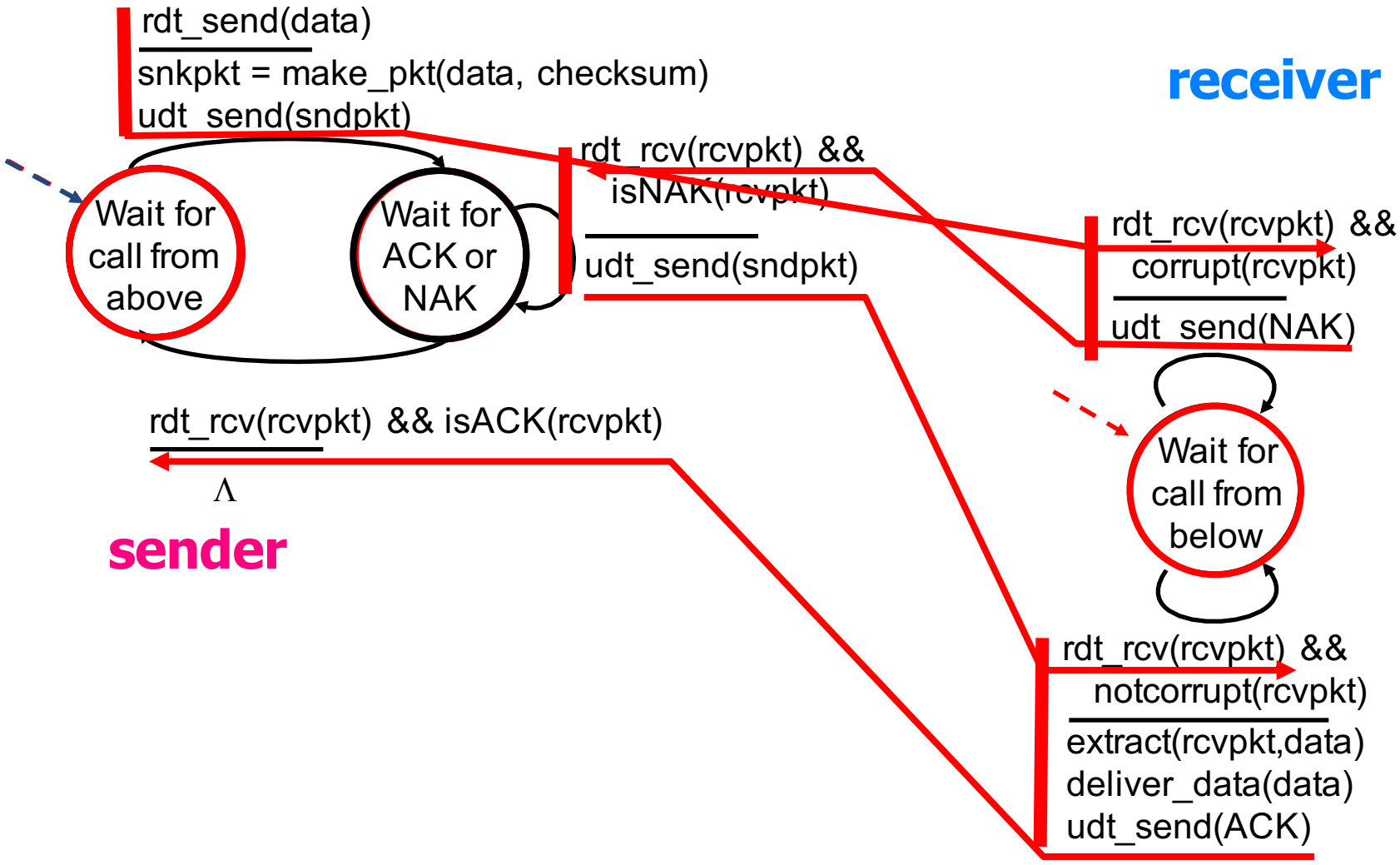
Stop and wait

Sender sends one packet, then waits for receiver response

rdt2.0: operation with no errors



rdt2.0: error scenario



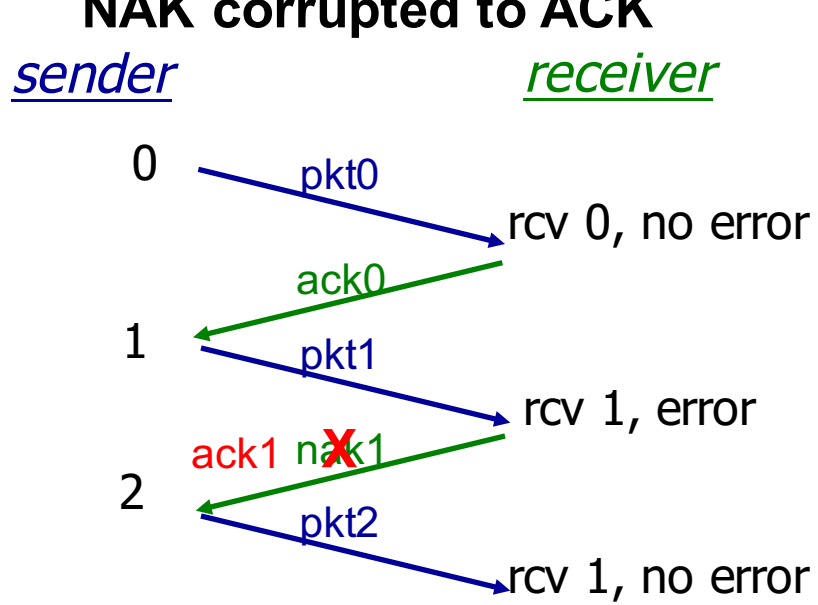
rdt2.0 has a fatal flaw!

(no seq #s on pkt/ack just yet, using only for clarity)

What if ACK/NAK corrupted?

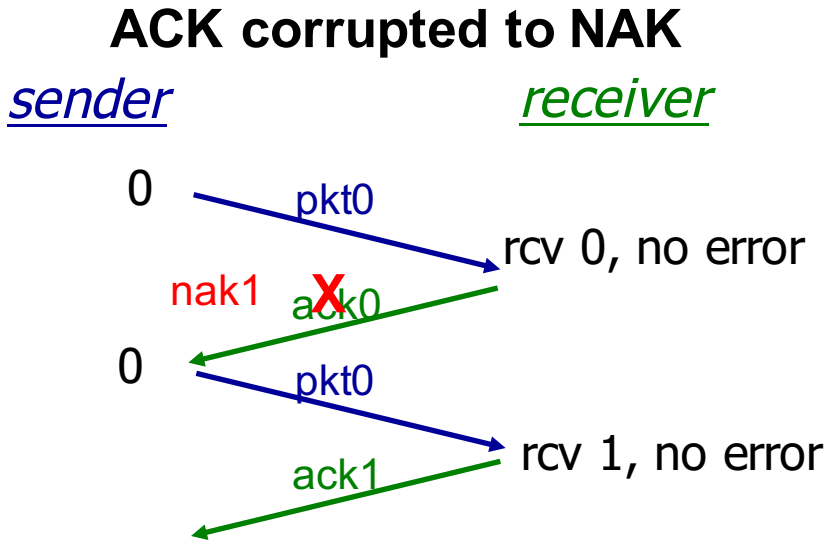
NAK corrupted to ACK

- sender may not retransmit when actually needed
- new packet seen as duplicate



ACK corrupted to NAK

- sender may retransmit packet when not needed
- duplicate seen as new packet



ACK/NAK just corrupted

- What do you do? Just assume NAK and resend?

Reliable Data Transport

GARBLED ACKS AND NAKS

rdt2.1: channel with bit errors, garbled N/ACKs

Problems

- underlying channel may flip bits in packet
  - packets, ACKs, NAKs may be garbled

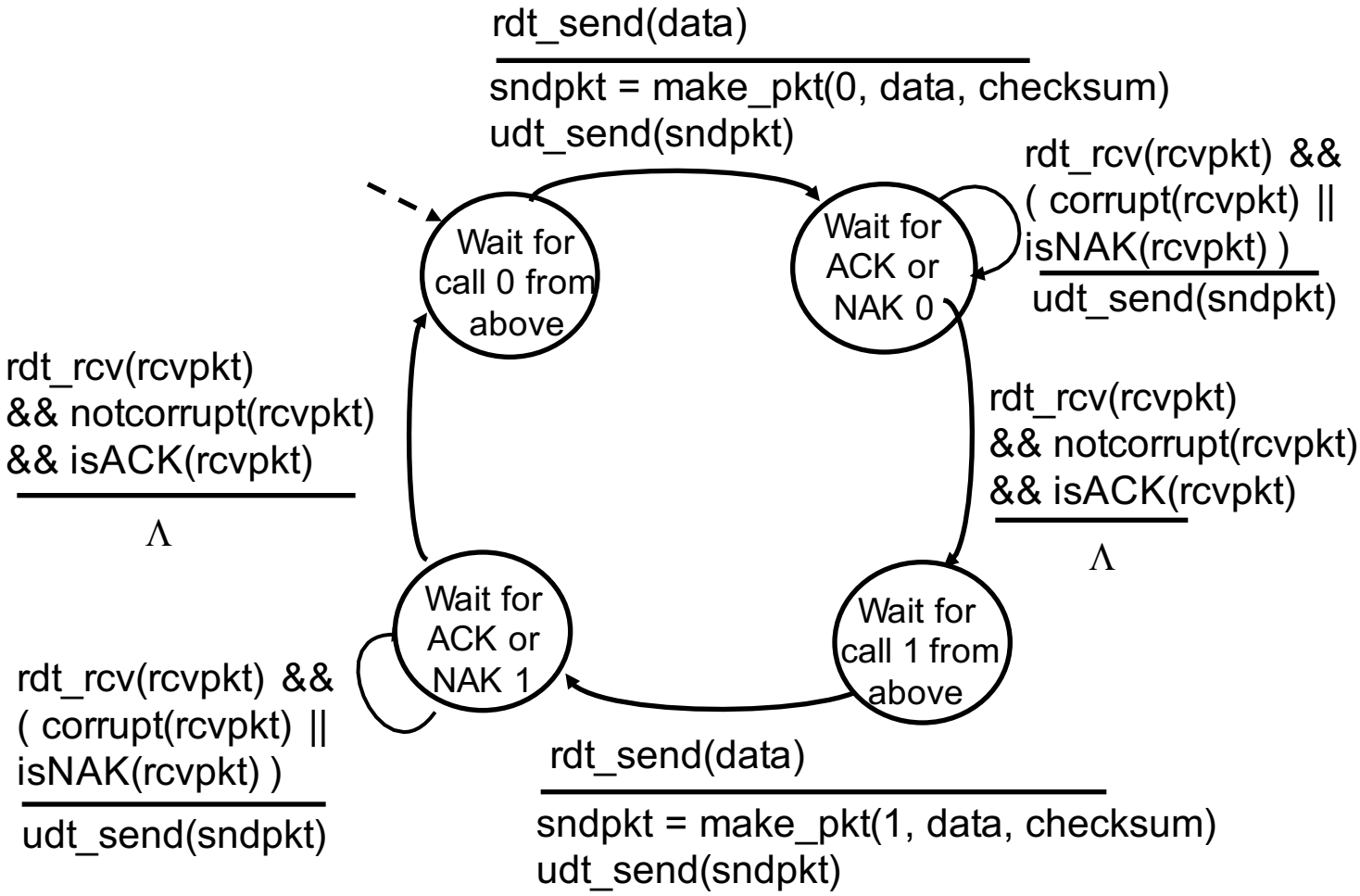
Solution

- Checksum for both packets and ACKs/NAKs
  - to detect bit 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
- Sender retransmits current pkt if ACK/NAK corrupted
  - adds sequence # to each pkt
  - receiver discards duplicate pkt

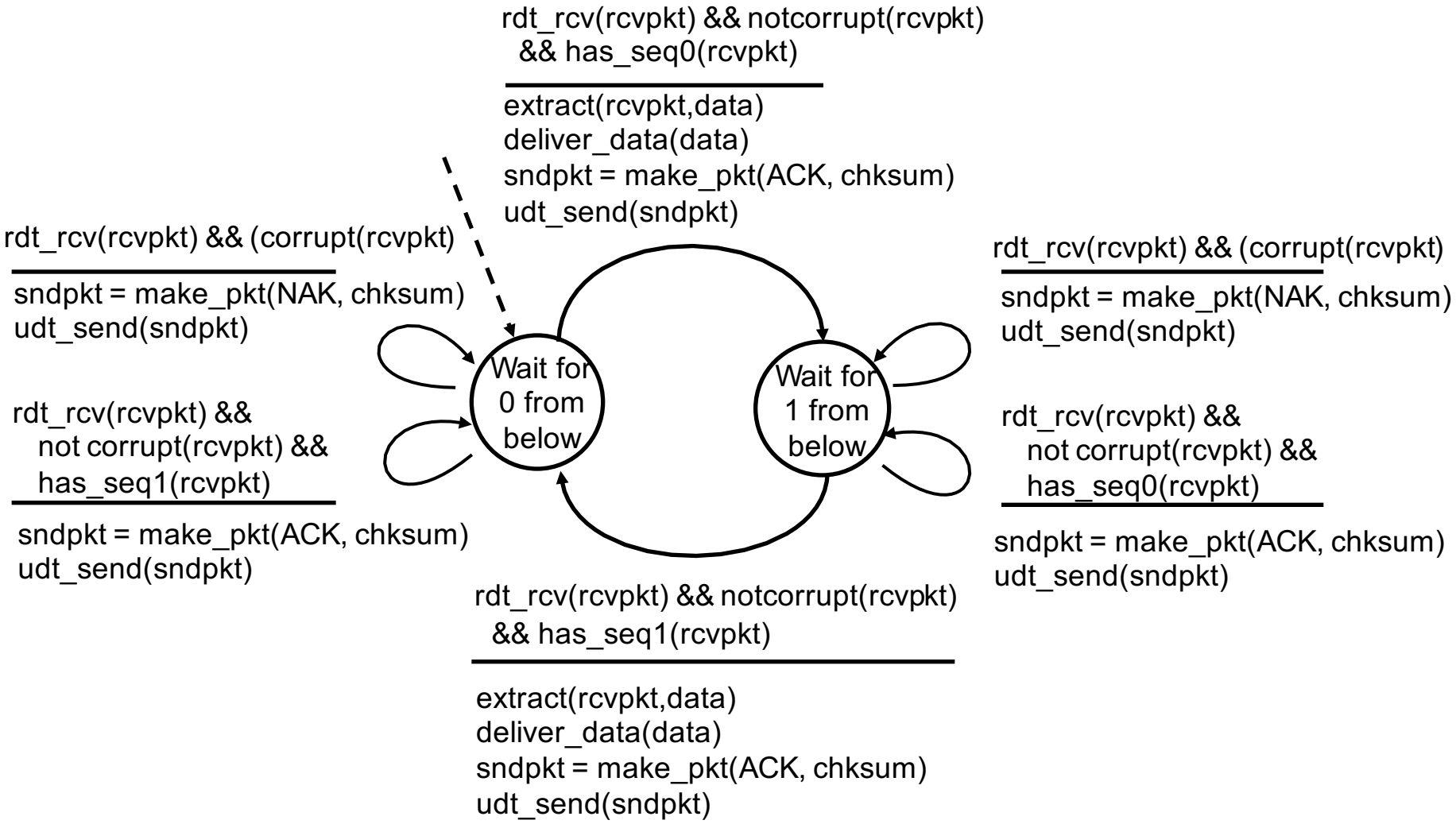
rdt2.1: sender, handles garbled ACK/NAKs

Only 2 seq #s: 0, 1

No channel reordering of pkts



rdt2.1: receiver, handles garbled ACK/NAKs



rdt2.1: discussion

Sender

Seq # added to pkt

- 2 seq #'s (0,1) suffice
- Q: Why?
- Stop-and-wait protocol

Checks

- if received ACK/NAK corrupted

Twice as many states

- state must remember whether expected pkt should have seq # 0 or 1

Receiver

- Checks
  - if received packet is corrupted or duplicate
  - state indicates whether 0 or 1 is expected pkt seq #
- Note
  - receiver cannot know if its last ACK/NAK received OK at sender

Reliable Data Transport

A NAK-FREE PROTOCOL

rdt2.2: a NAK-free protocol

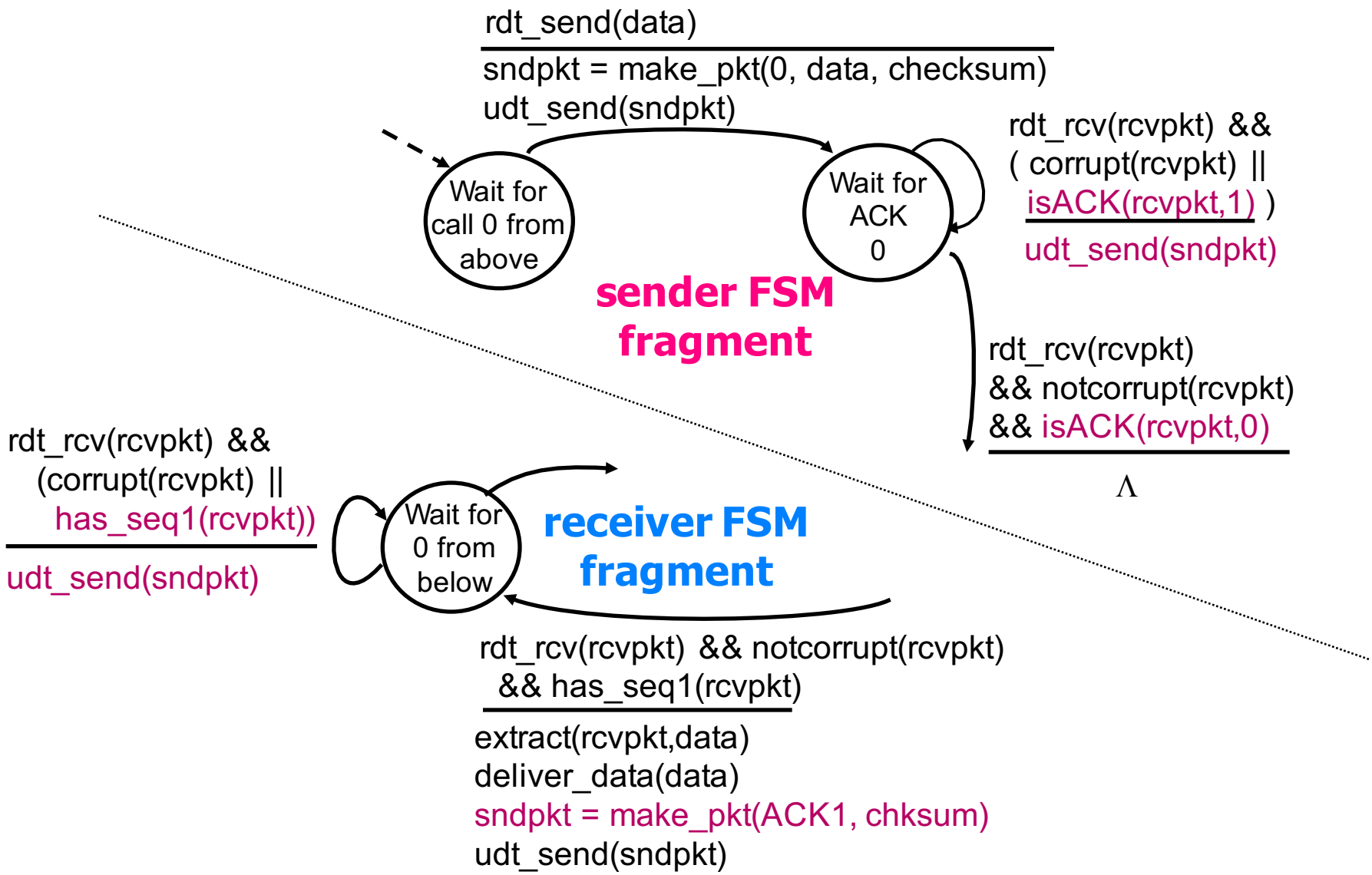
Same functionality as rdt2.1, using ACKs only

- 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



Reliable Data Transport

CHANNELS WITH ERROR AND LOSS

rdt3.0: channels with errors and loss

Problems

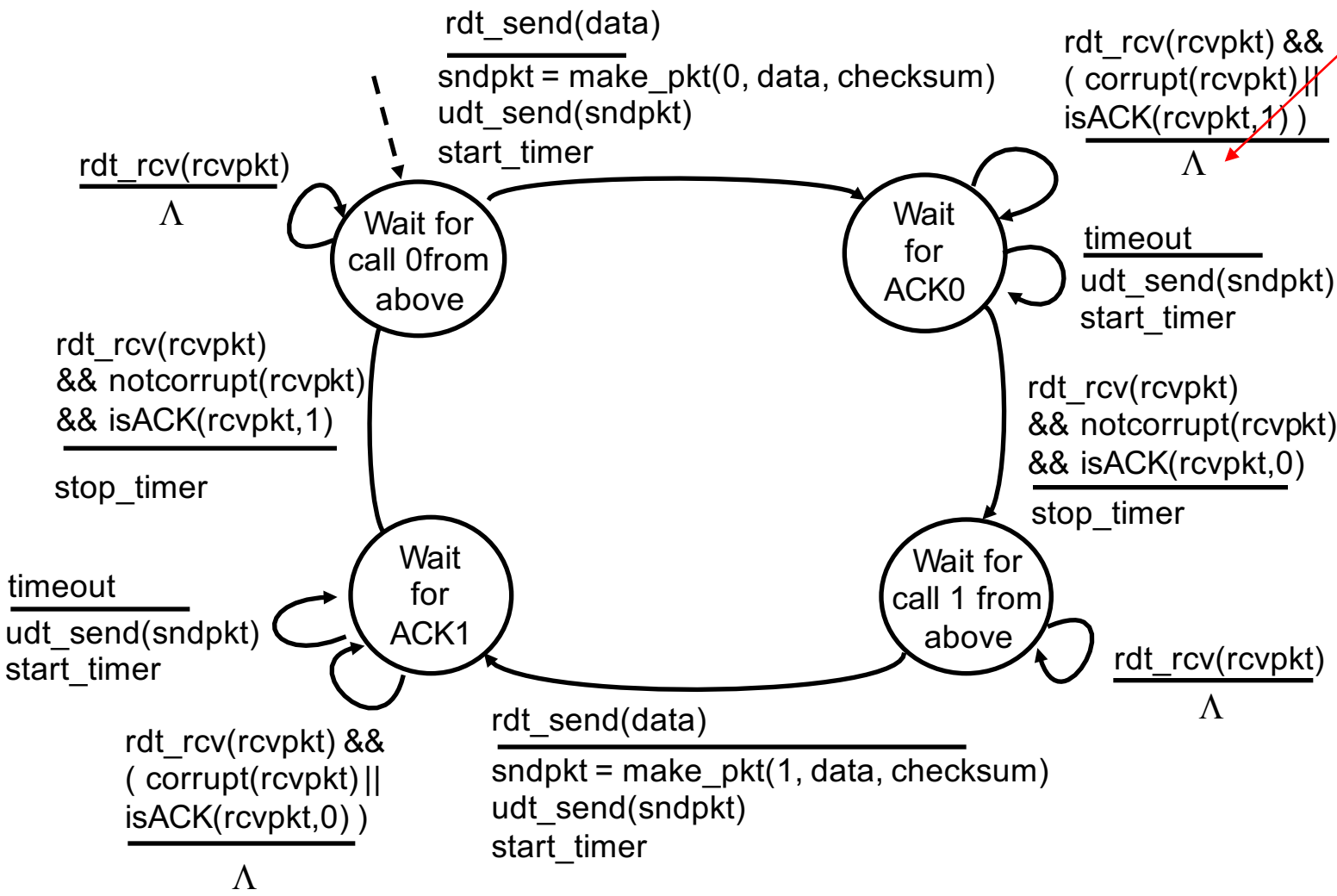
- underlying channel may flip bits in packet
  - both data and ACKs may be garbled
- underlying channel can also lose packets
  - both data and ACKs
- checksum, seq. #, ACKs, retransmissions will be of help
  - … but not enough

Solution: add countdown timer

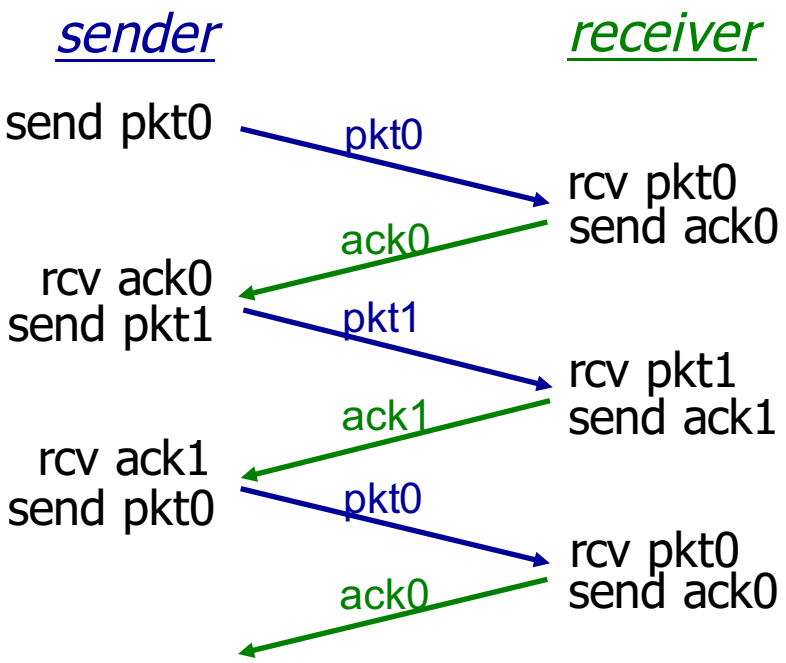
- sender waits "reasonable" amount of time for ACK
  - retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost)
  - retransmission will be duplicate, but seq #'s already handles this
- receiver must specify seq # of pkt being ACKed

rdt3.0 sender

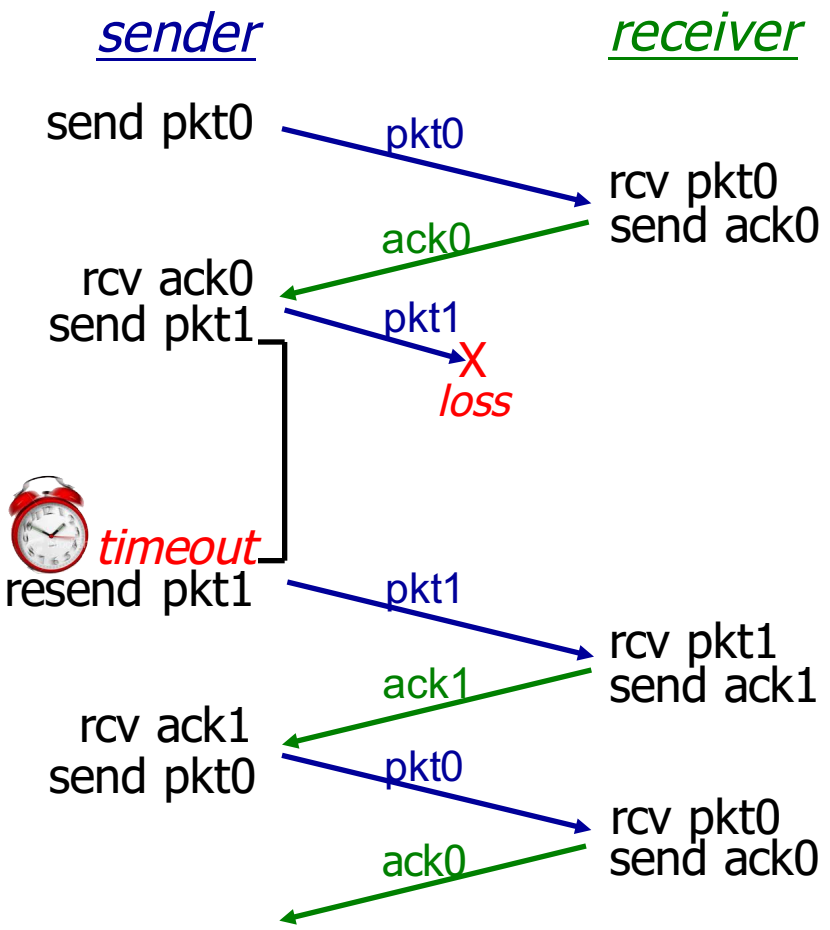
Why do nothing ? Why not resend pkt0? Because sender doesn't know whether ack1 means pkt 0 garbled or pkt 1 duplicate received By not resending pkt 0, sender doesn't introduce potentially unnecessary (even if valid) traffic: saves bandwidth



rdt3.0 in action

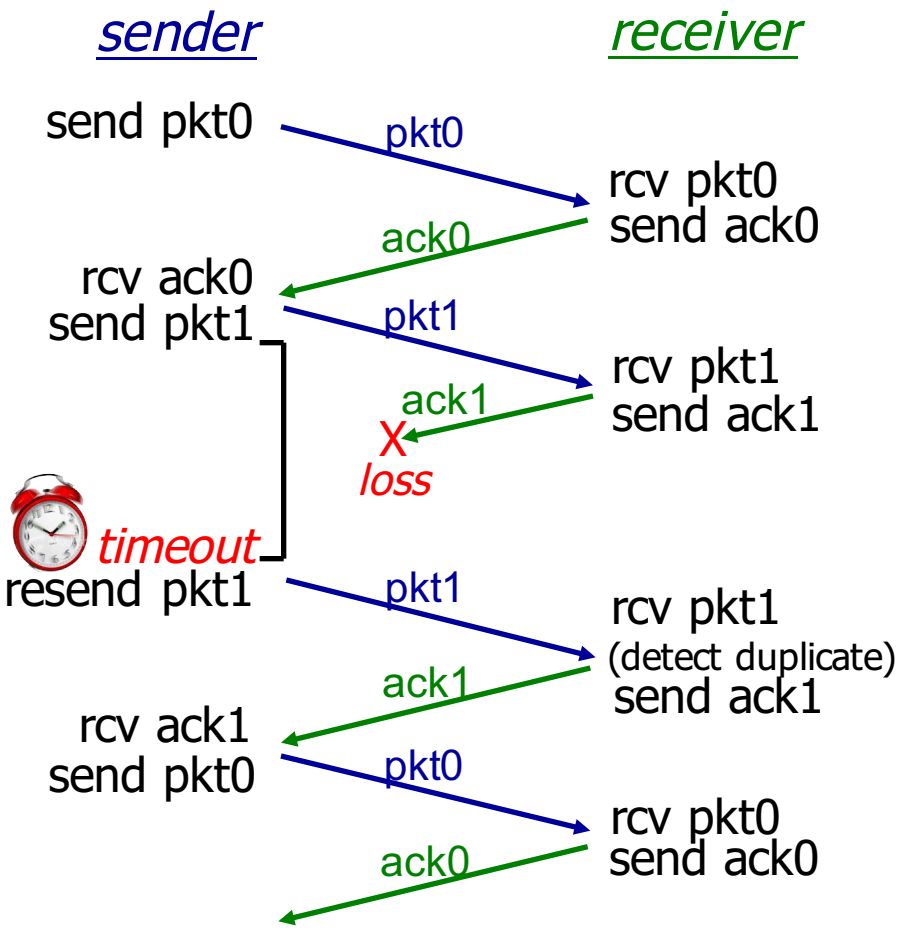


(a) no loss

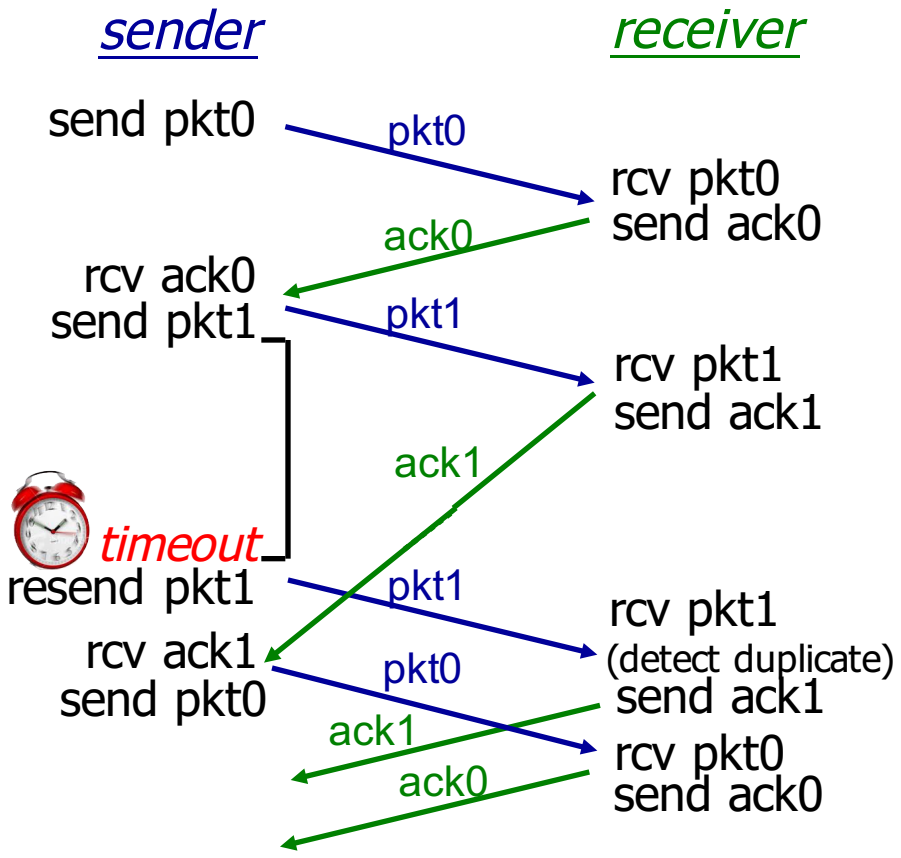


(b) packet loss

rdt3.0 in action

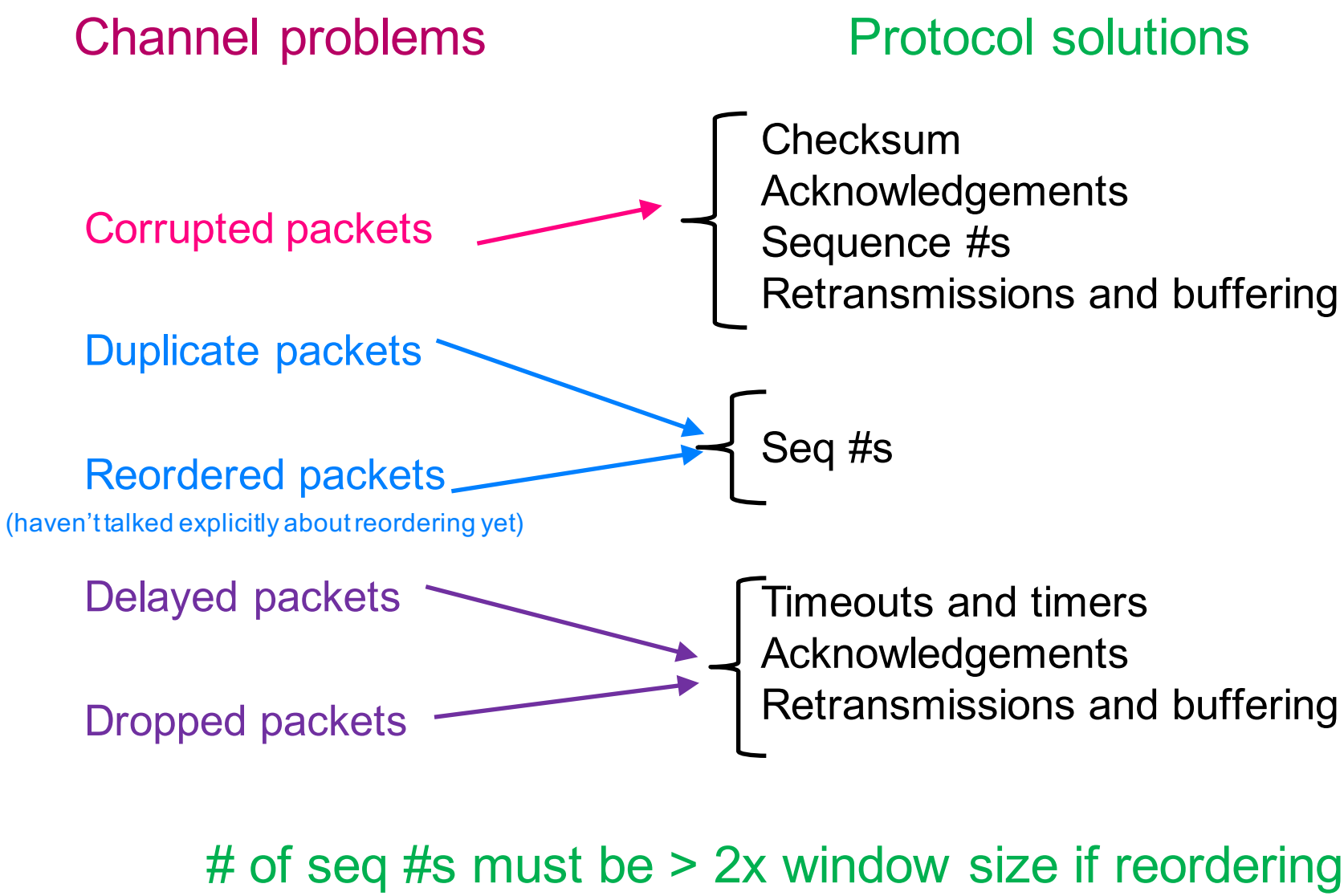


(c) ACK loss



(d) premature timeout/ delayed ACK

Summary of techniques and uses



# of seq #s must be > 2x window size if reordering

Reliable Data Transport

PIPELINED PROTOCOLS

Performance of rdt3.0

rdt3.0 is correct, but awful performance

- e.g., 1 Gbps link, 15 ms prop. delay, 8000 bit (=1KB) packet

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

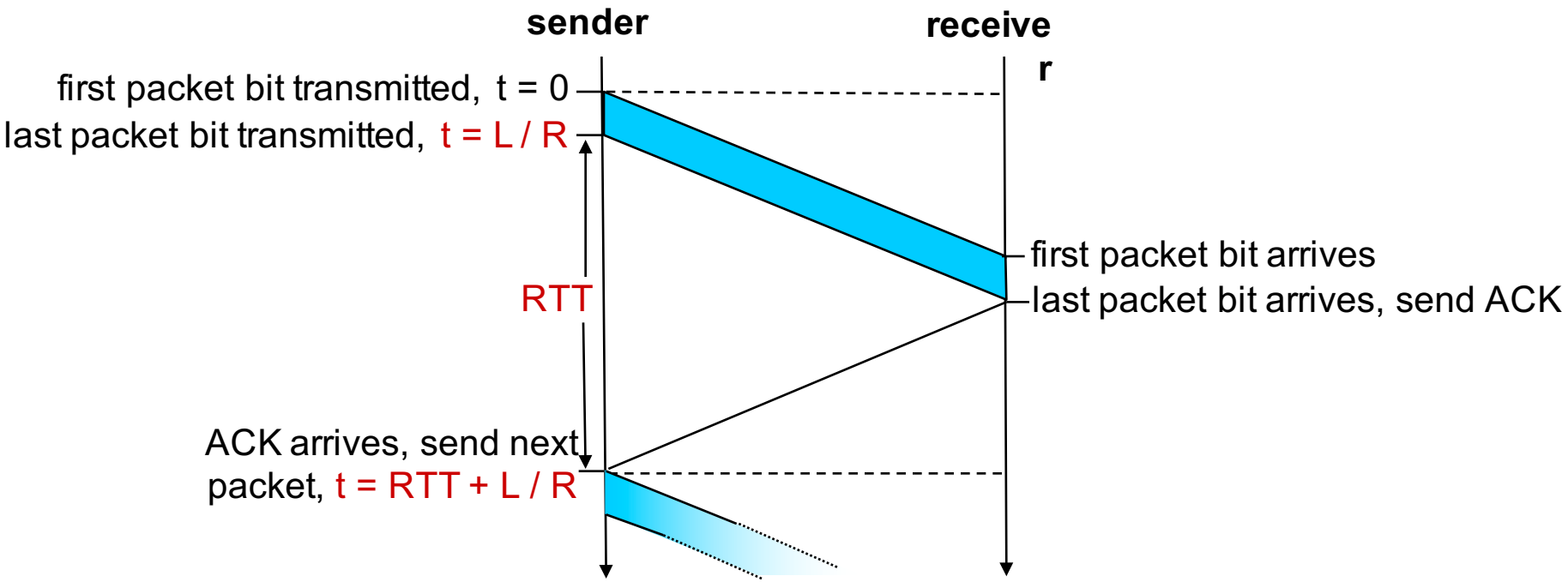
U_{sender}: utilization is fraction of time sender busy sending

$$U_{sender} = \frac{\overbrace{L / R}^{\text{Time spent sending stuff}}}{\underbrace{RTT + L / R}_{\text{Total time we're considering}}} = \frac{.008}{30.008} = 0.00027$$

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



Time spent sending stuff

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

Total time we're considering

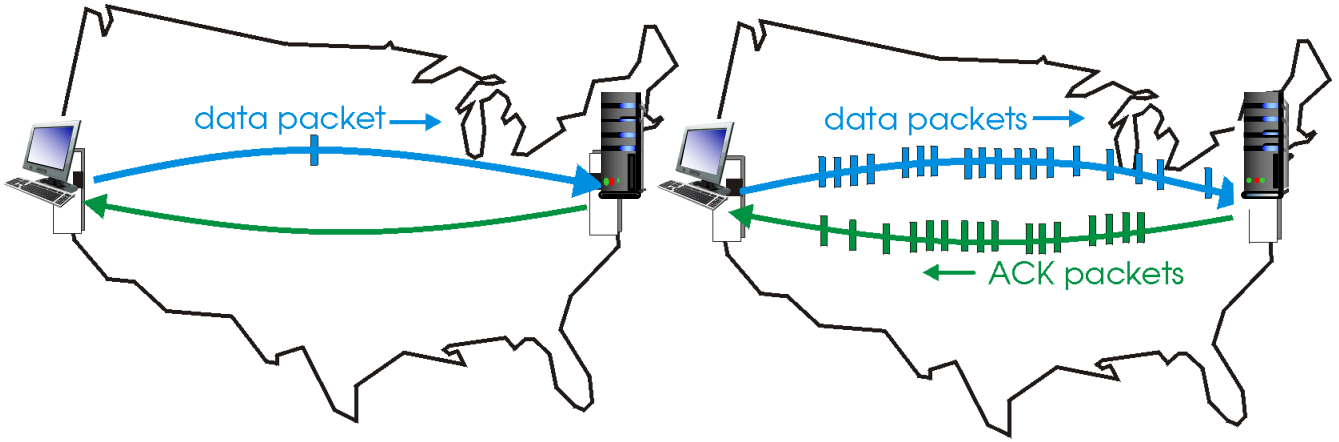
Problem: Keeping the pipe full (i.e. maintain high link utilization)

Creating a more efficient protocol

How? get rid of stop-and-wait

Instead: pipelining (also called sliding-window protocols)

- sender allows multiple, in-flight, yet-to-be-acknowledged pkts
  - send up to N packets at a time: N packets in flight, unacked
  - range of seq #s must be increased
  - sender needs more memory to buffer outstanding unacked packets



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

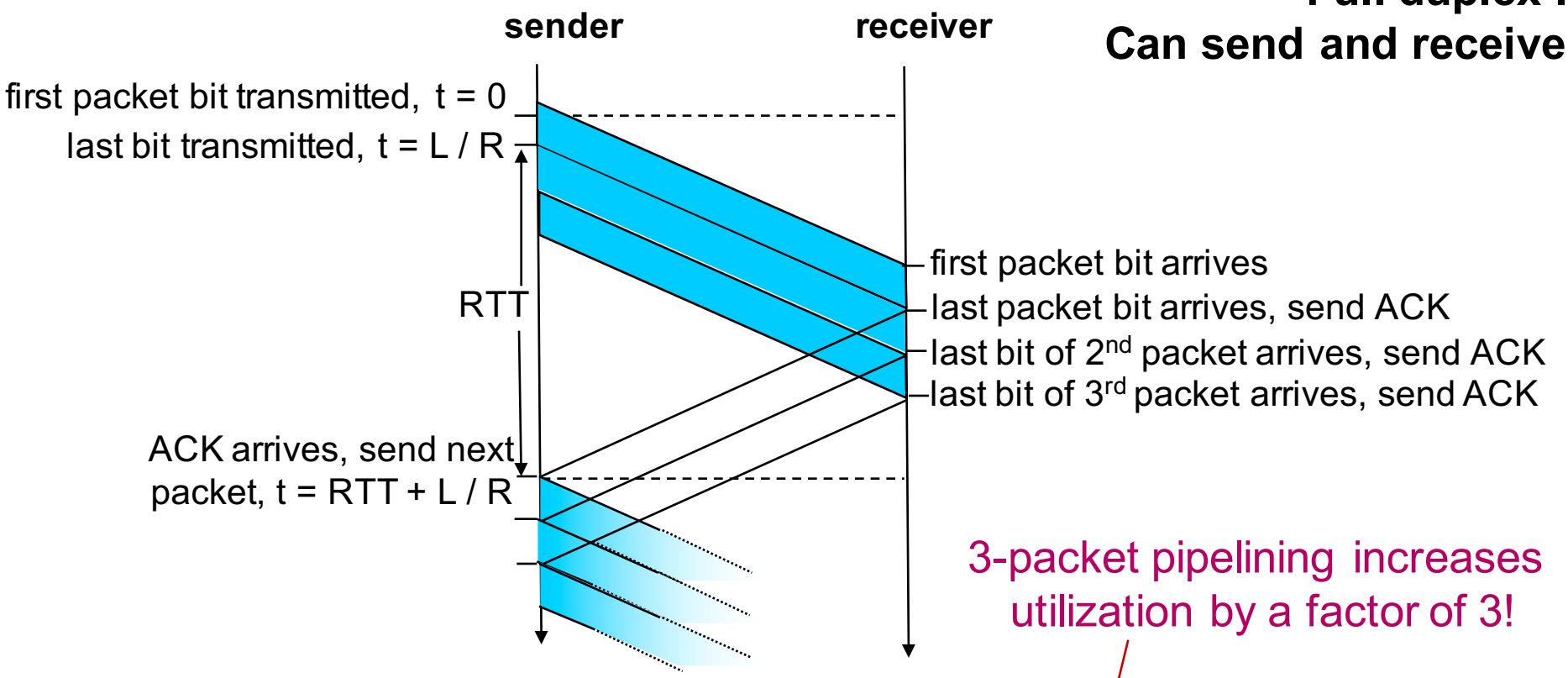
(b) a pipelined protocol in operation

Achieves higher link utilization than stop-and-wait

Pipelining: increased utilization

3-packet pipelining example

Full duplex li
Can send and receive a



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

Time spent sending stuff

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

Total time we're considering

Pipelined protocols

Sent N packets without receiving ACKs

- Q: How does receiver ACK packets now?

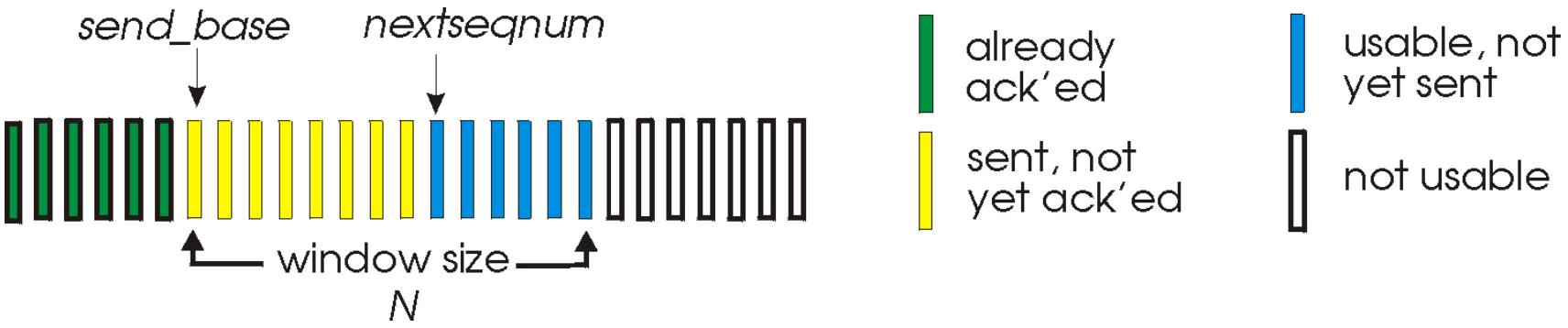
1. Cumulative ACKs: used in Go-Back-N protocol
   - receiver only sends cumulative ack
     - doesn't ack packet if there's a gap
   - sender has timer for oldest unacked packet
     - when timer expires, retransmit all unacked packets
     - packets received correctly may be retransmitted

2. Selective ACKs: used in Selective Repeat protocol
   - receiver sends individual ack for each packet
   - sender has timer for each unacked packet
     - when timer expires, retransmit only that unacked packet
     - only corrupted/lost packets are retransmitted

How pipelining/sliding window protocols work

Sliding window

- how sender keeps track of what it can send
- window: set of N adjacent seq #s
  - only send packets in window



If window large enough, will fully utilize link