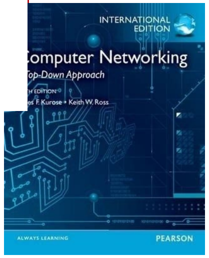
Mic!

Principles of Reliable Transport

Chapters
3.4 & 3.5



Geoff Coulson
Week 15 Lecture 1

Preamble: we are being very selective

- we'll focus on *principles of reliability* than on the details of TCP
  - although we will look at TCP as well
- we'll be using slides from Kurose and Ross
  - I hereby acknowledge their copyright!
  - indication as we go of topics we're omitting from the book
  - but it's very well worthwhile reading and understanding the omitted sections!

Chapter 3 outline (from the book)

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer (omit: selective resend)

3.5 connection-oriented transport: TCP

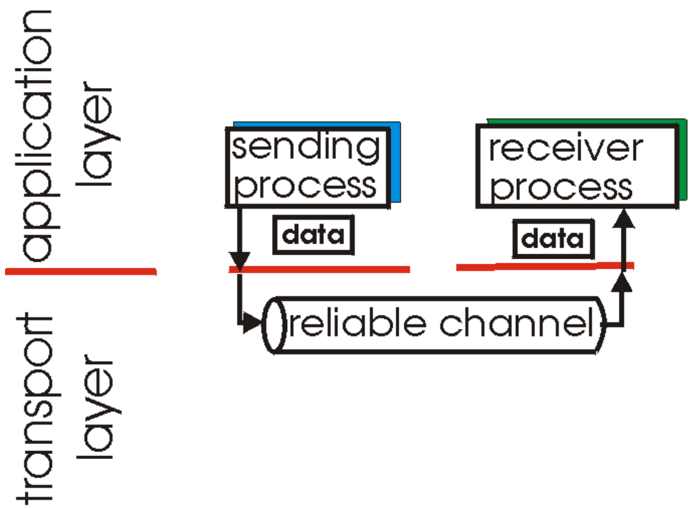
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

Principles of reliable data transfer

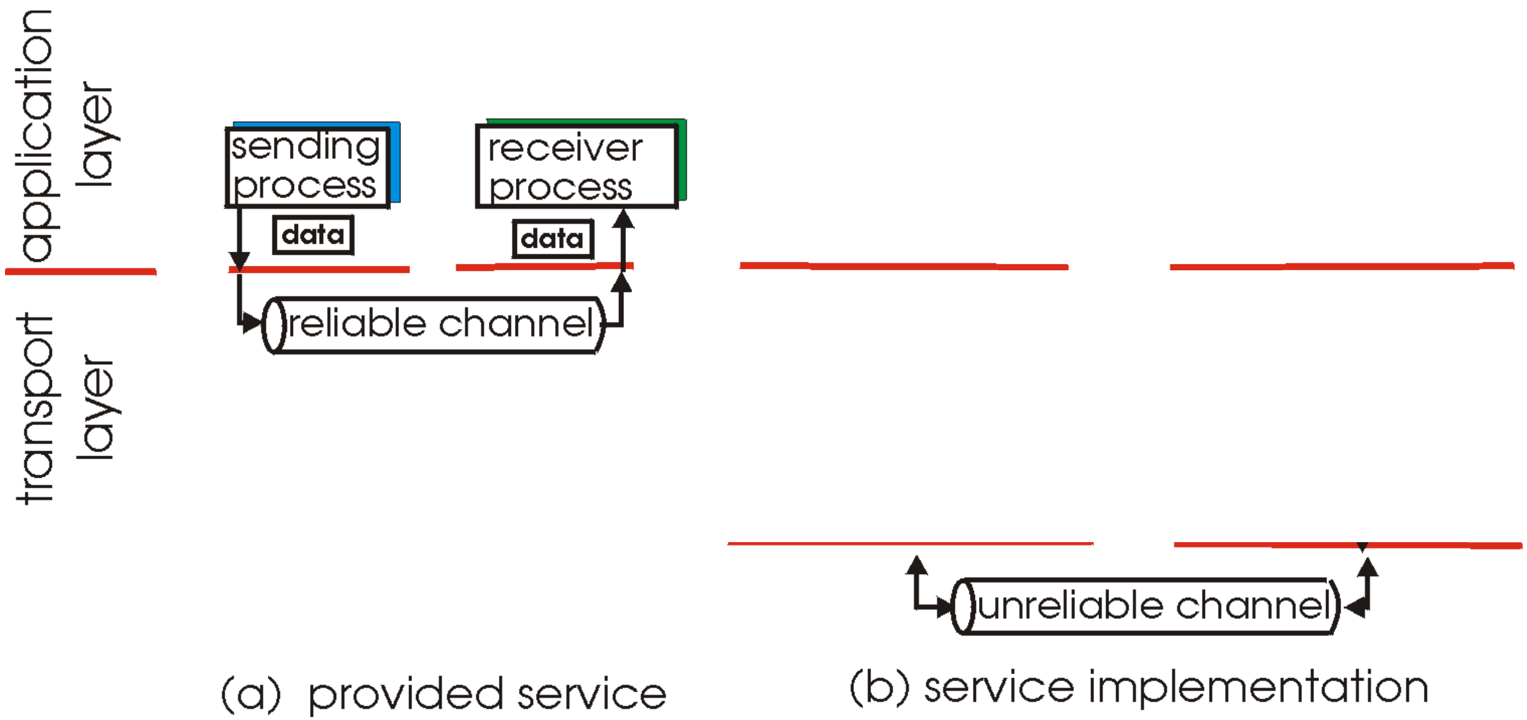
- important in application, transport, link layers
  - top-10 list of important networking topics!



(a) provided service

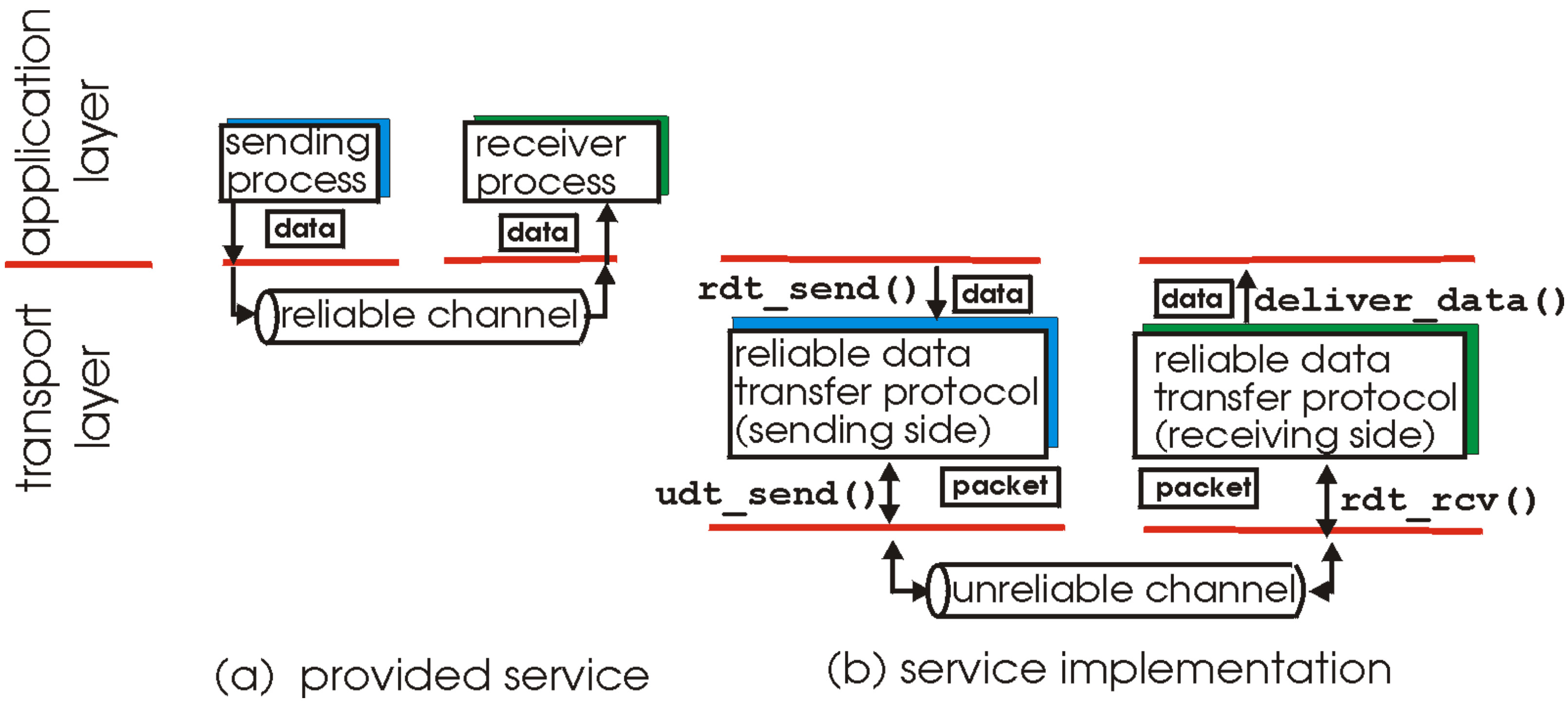
Principles of reliable data transfer

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



Principles of reliable data transfer

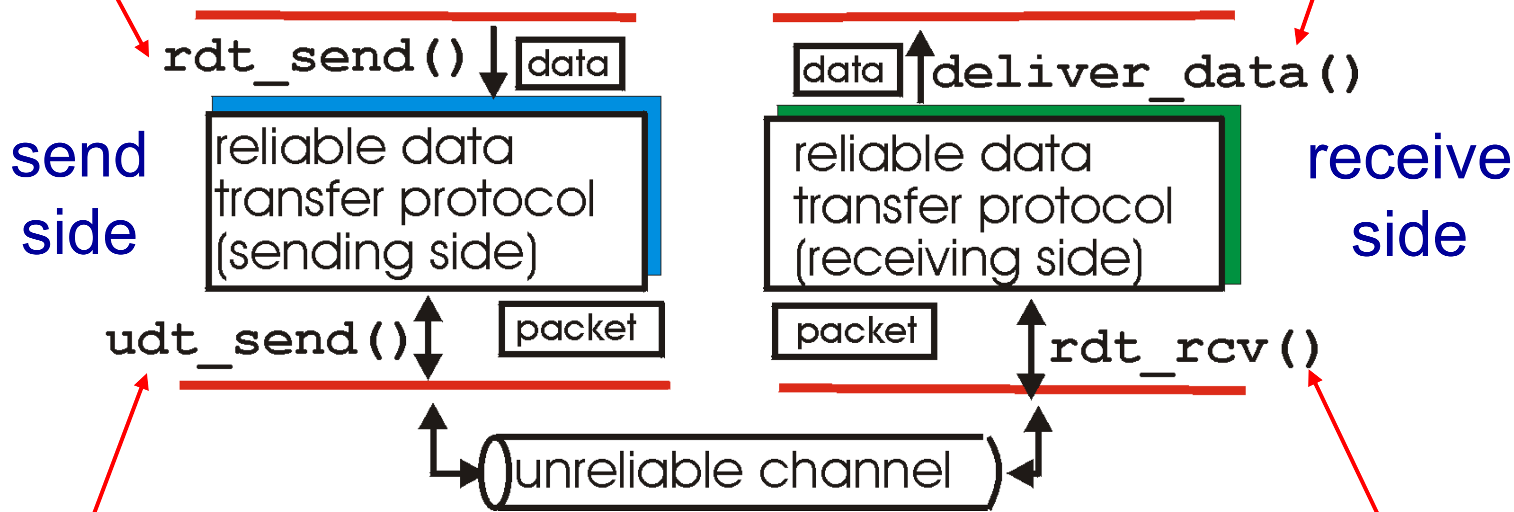
- important in application, transport, link layers
  - top-10 list of important networking topics!



Reliable data transfer: getting started

rdt_send(): called from above, (e.g., by app.). Passed data to be delivered to receiver's upper layer

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



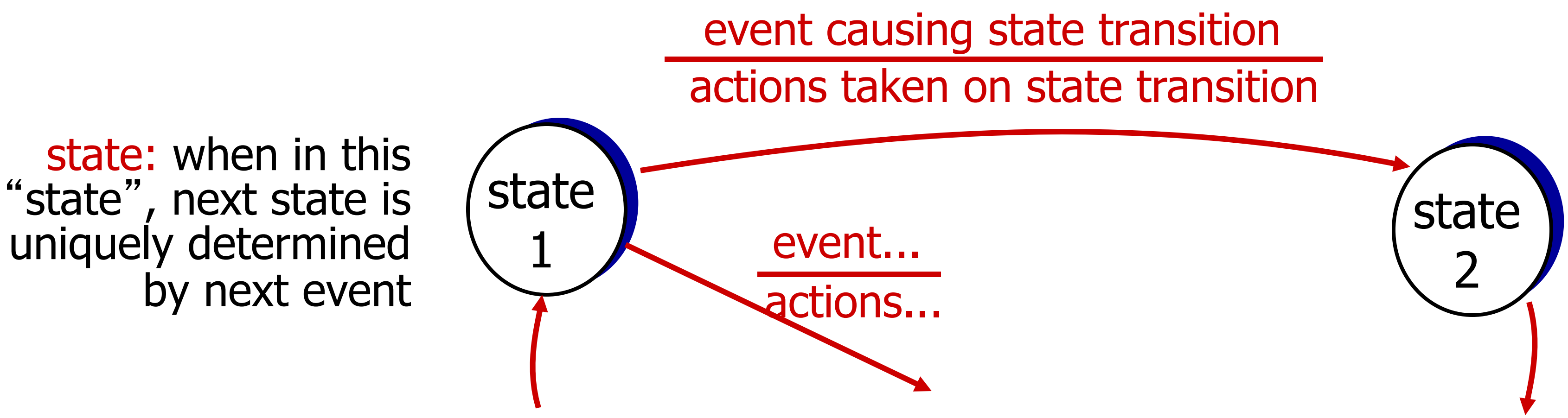
udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

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

Reliable data transfer: getting started

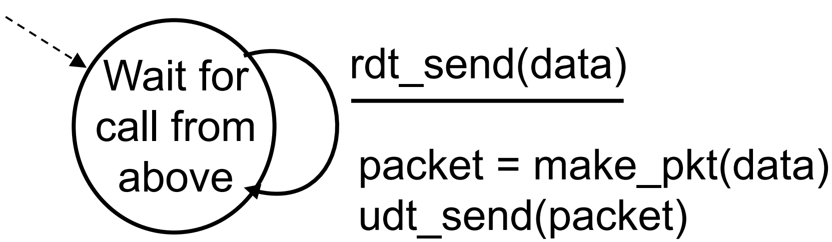
we'll:

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

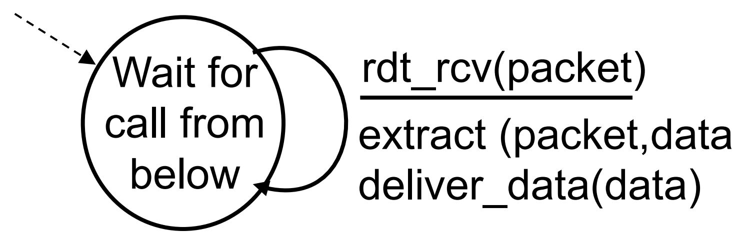


rdt1.0: reliable transfer over a reliable channel

- underlying channel assued to be perfectly reliable
  - no bit errors
  - no loss of packets
- we define 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

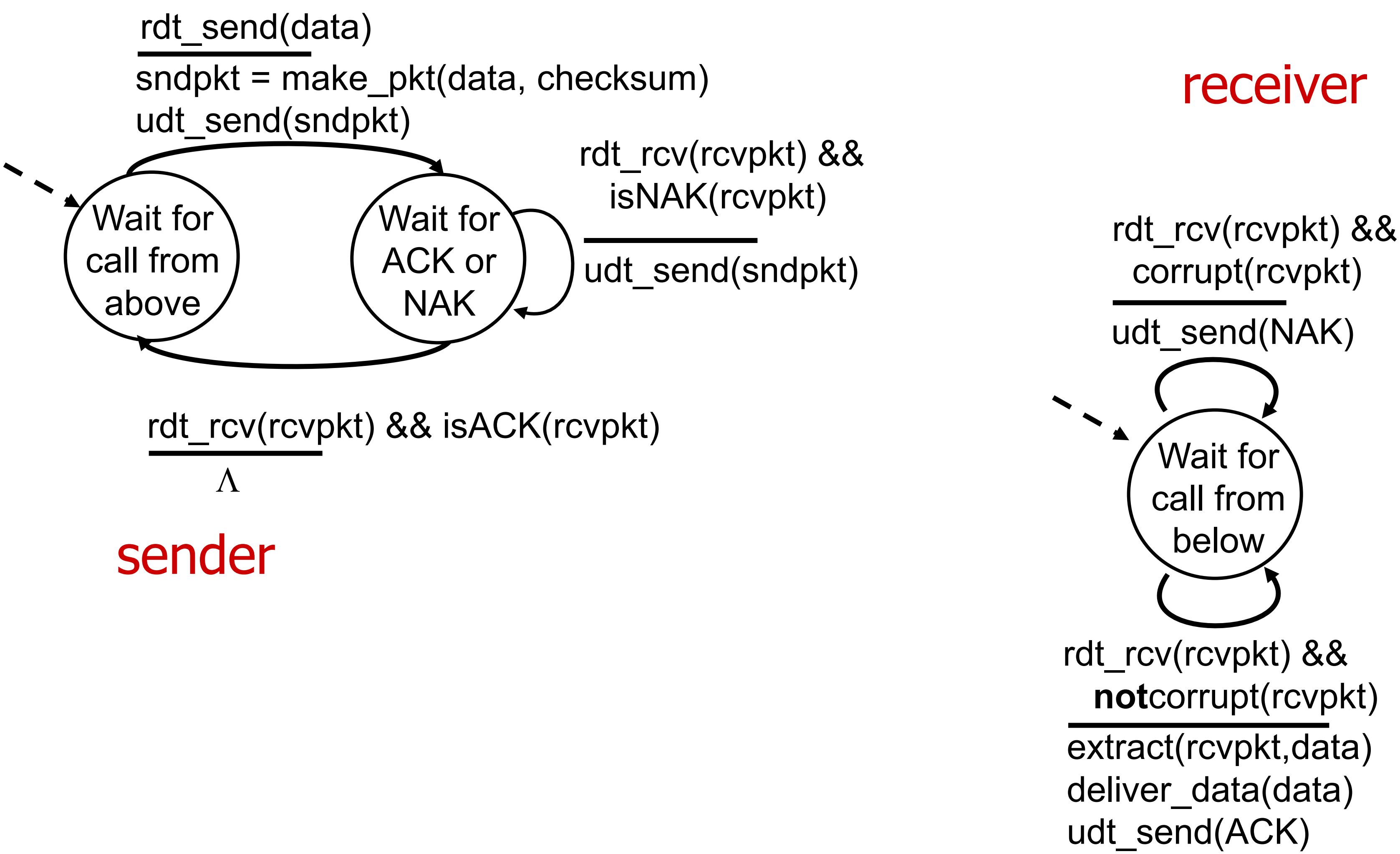
- this time, underlying channel may flip bits in packet
  - employ *checksum* to detect bit errors
- *the* question: how to recover from errors:

How do humans recover from "errors" during conversation?

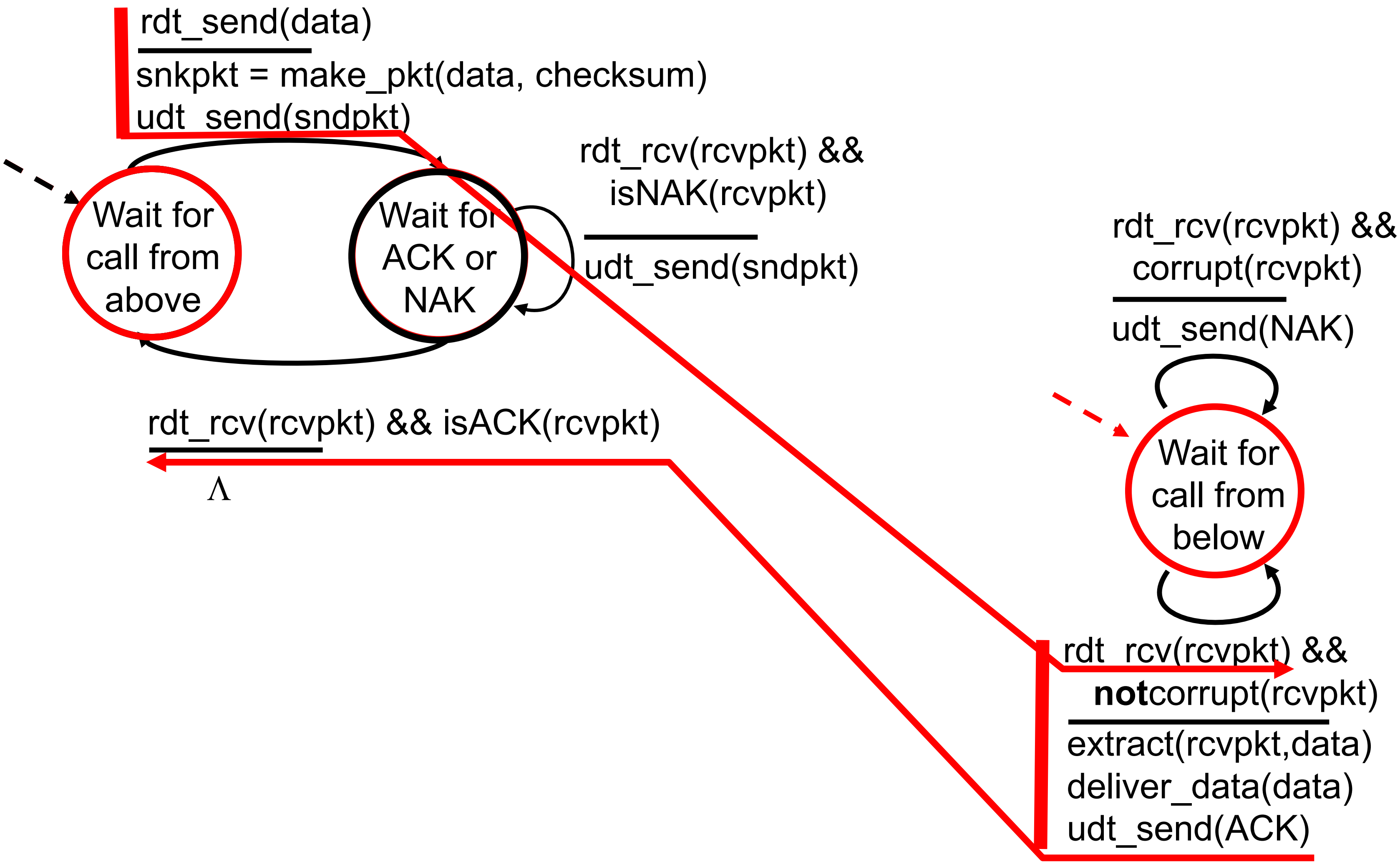
rdt2.0: channel with bit errors

- this time, underlying channel may flip bits in packet
  - employ *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

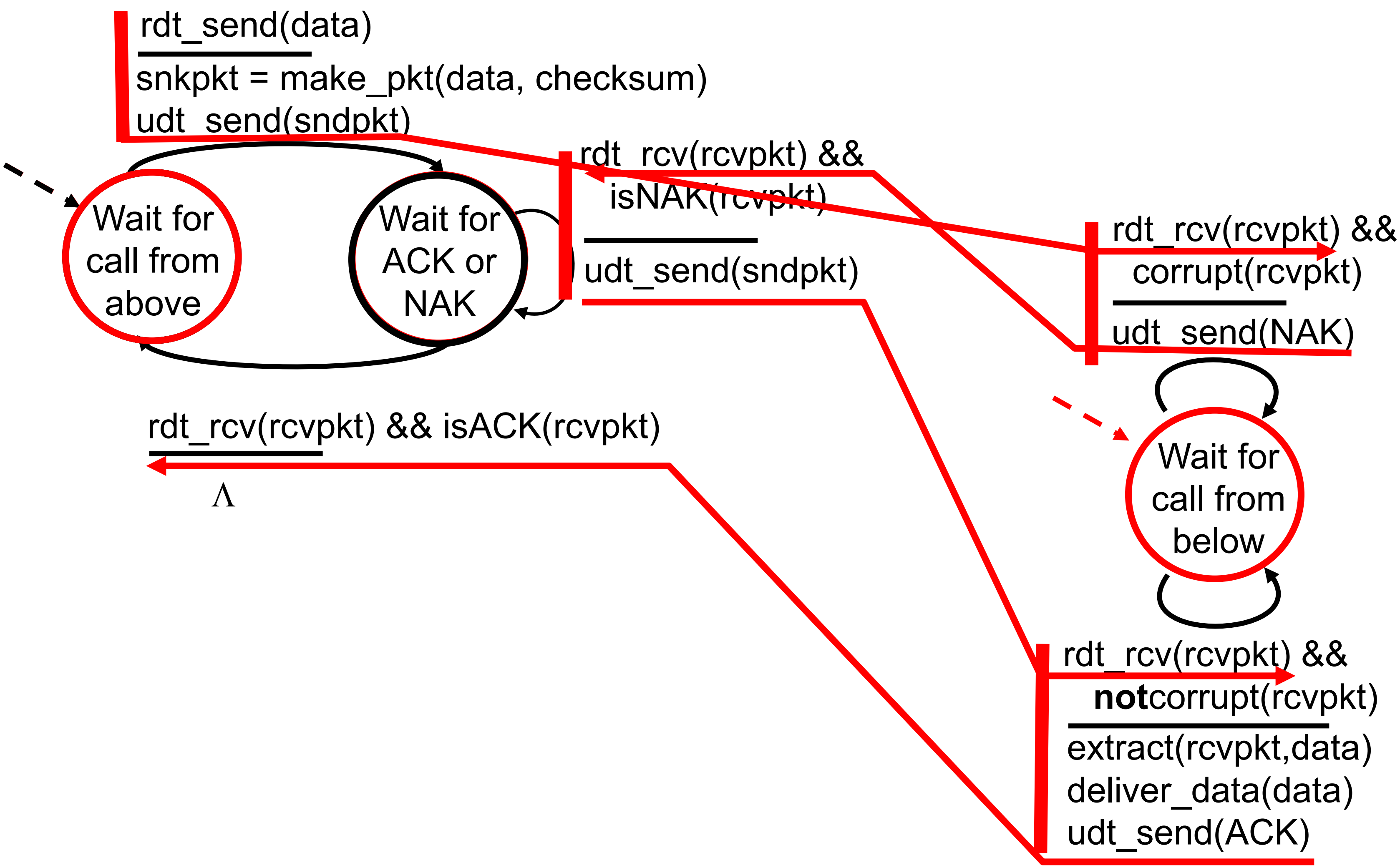
rdt2.0: FSM specification



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!
- sender can't just retransmit: possible **duplicate**

how can we handle duplicates?

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

stop and wait

sender sends one packet, then waits for receiver response

rdt2.1: outline

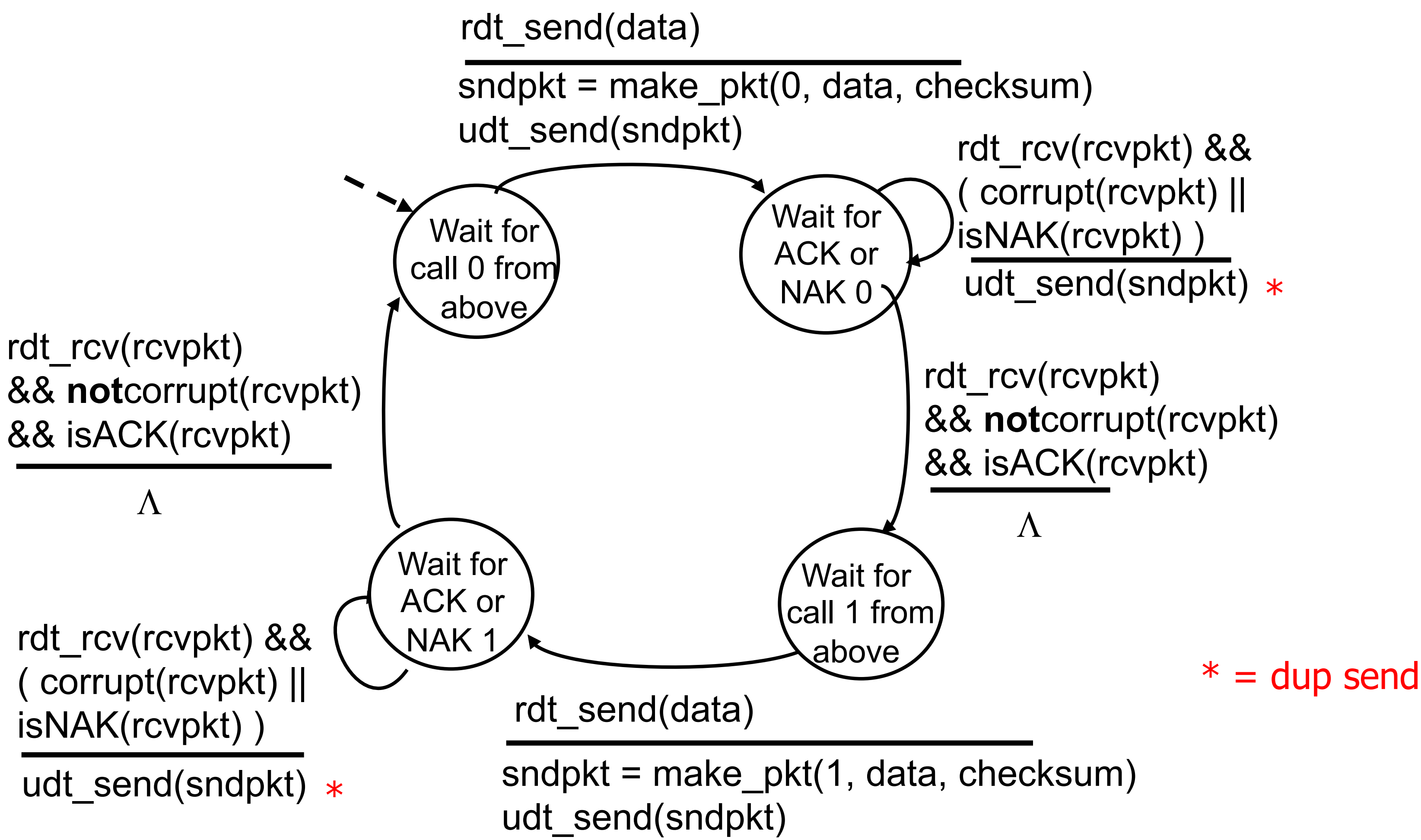
sender:

- seq # added to pkt
- two seq. #'s (0,1) will suffice. Why?
- must check if received ACK/NAK corrupted
- twice as many states
  - states must "remember" whether an incoming ACK/NAK packet should relate to seq # 0 or 1

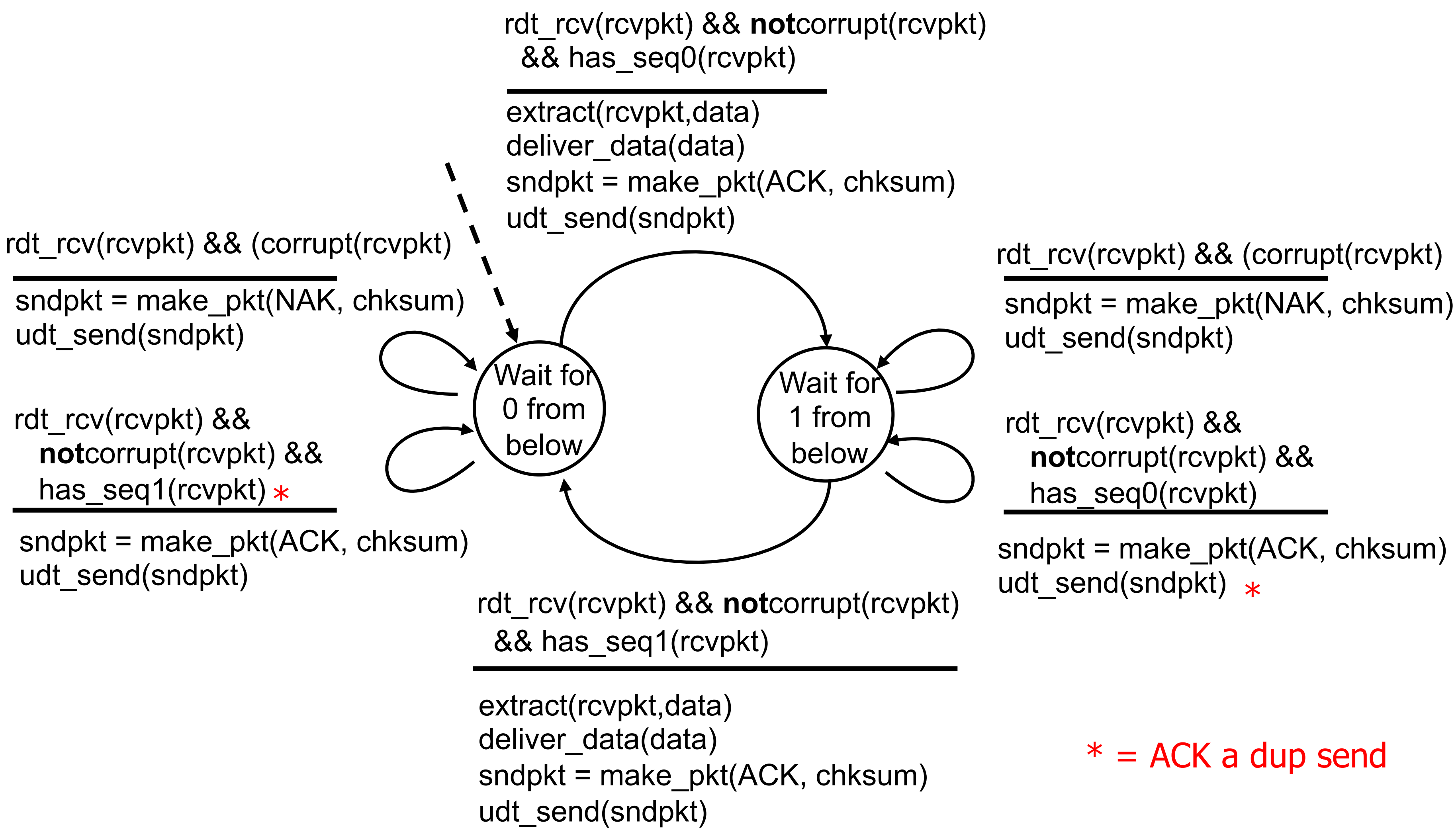
receiver:

- twice as many states here, too
- must check if received packet is a duplicate
  - state indicates whether seq # 0 or 1 is expected

rdt2.1: sender; handles garbled ACK/NAKs



rdt2.1: receiver; handles garbled ACK/NAKs



* = ACK a dup send

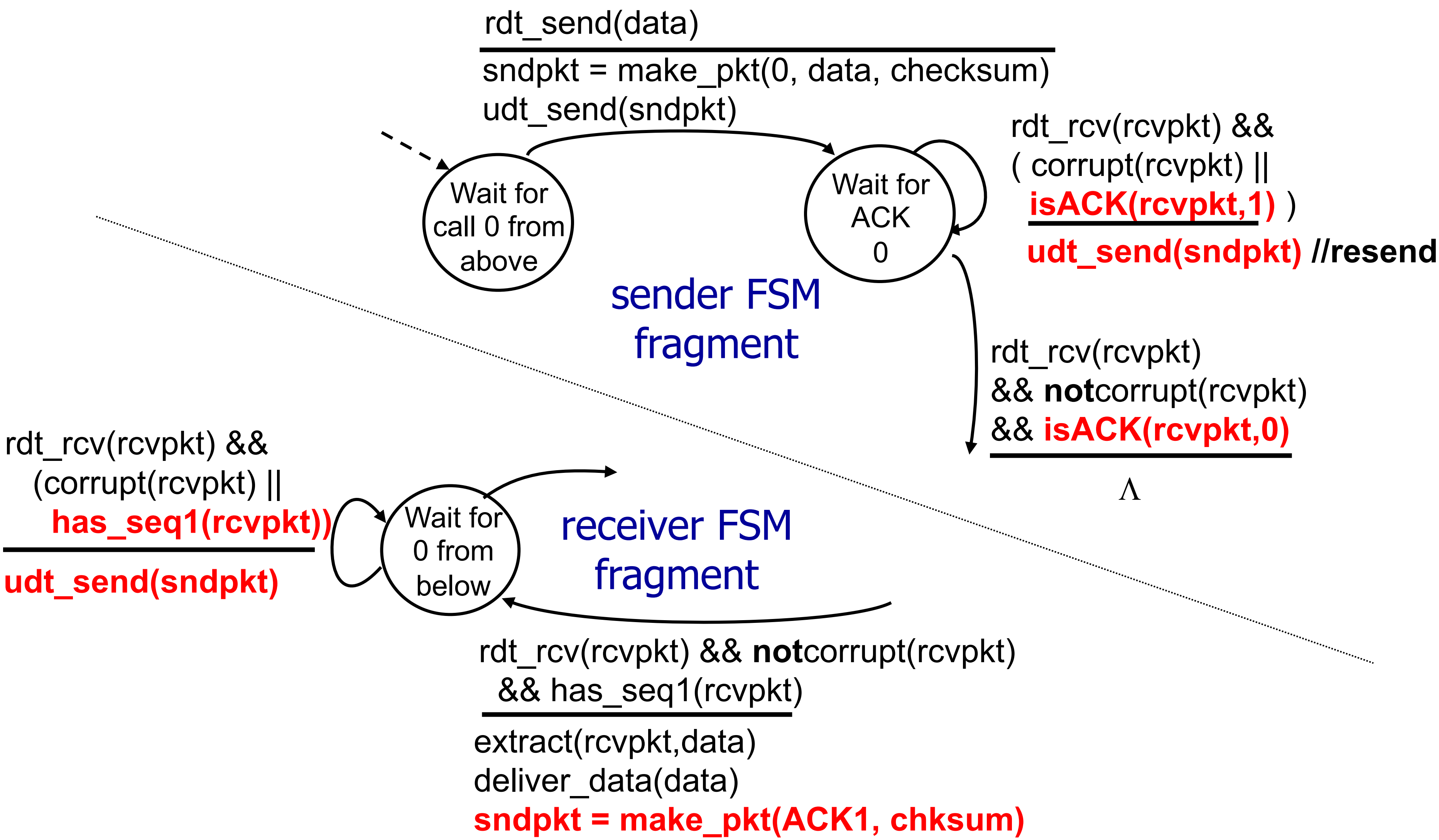
rdt2.2: a NAK-free protocol

- same functionality as rdt2.1, but uses ACKs only
- instead of a NAK, on receipt of a bad packet, the receiver sends an *ACK for last pkt correctly received*
  - so, receiver ACK must explicitly include seq # of pkt being ACKed
- duplicate ACK received at sender results in same action as NAK: *retransmit current pkt*

Why?

simplifies; eases further extensions; it's what TCP does

rdt2.2: sender, receiver fragments



rdt3.0: channels with errors *and* loss

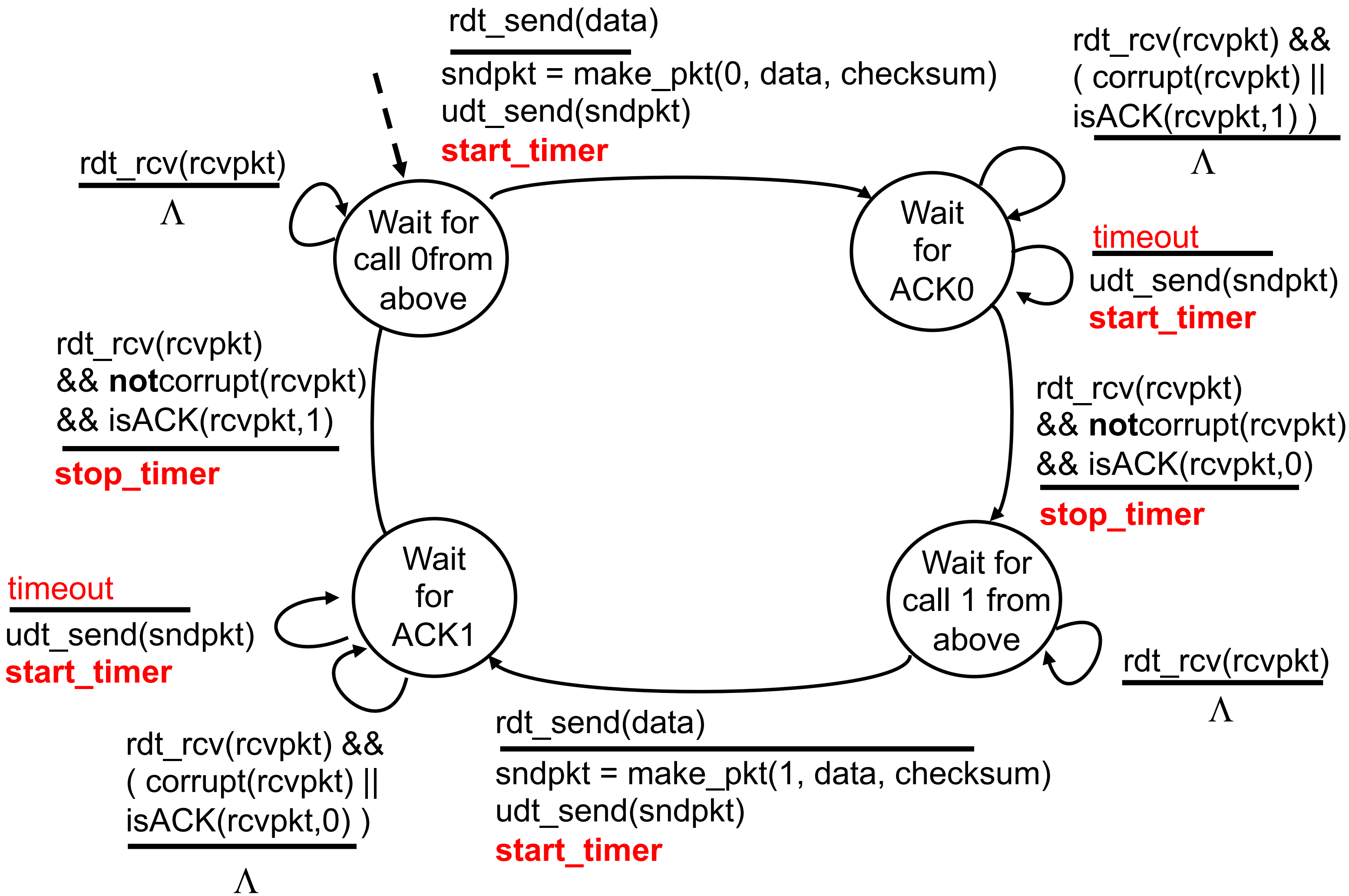
new assumption: as well as corrupting packets, the underlying channel may now also *lose* packets (both data and ACKs)

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

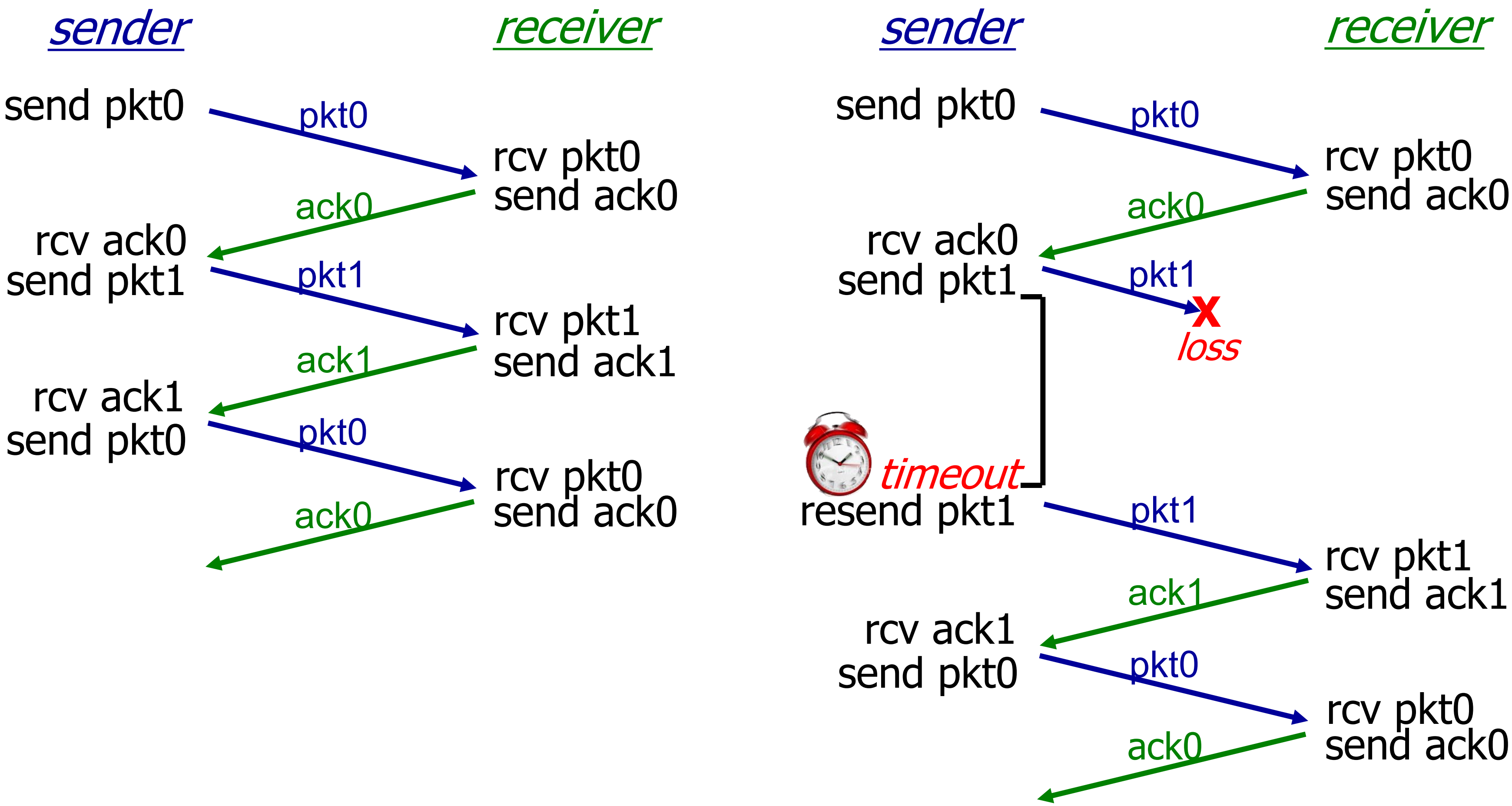
approach: sender waits "reasonable" amount of time for ACK

- retransmits if no ACK received in this time
- if ACK was just delayed (not lost):
  - retransmission will be received as a duplicate; but seq. #'s already handle this
- requires *countdown timer*

rdt3.0 sender



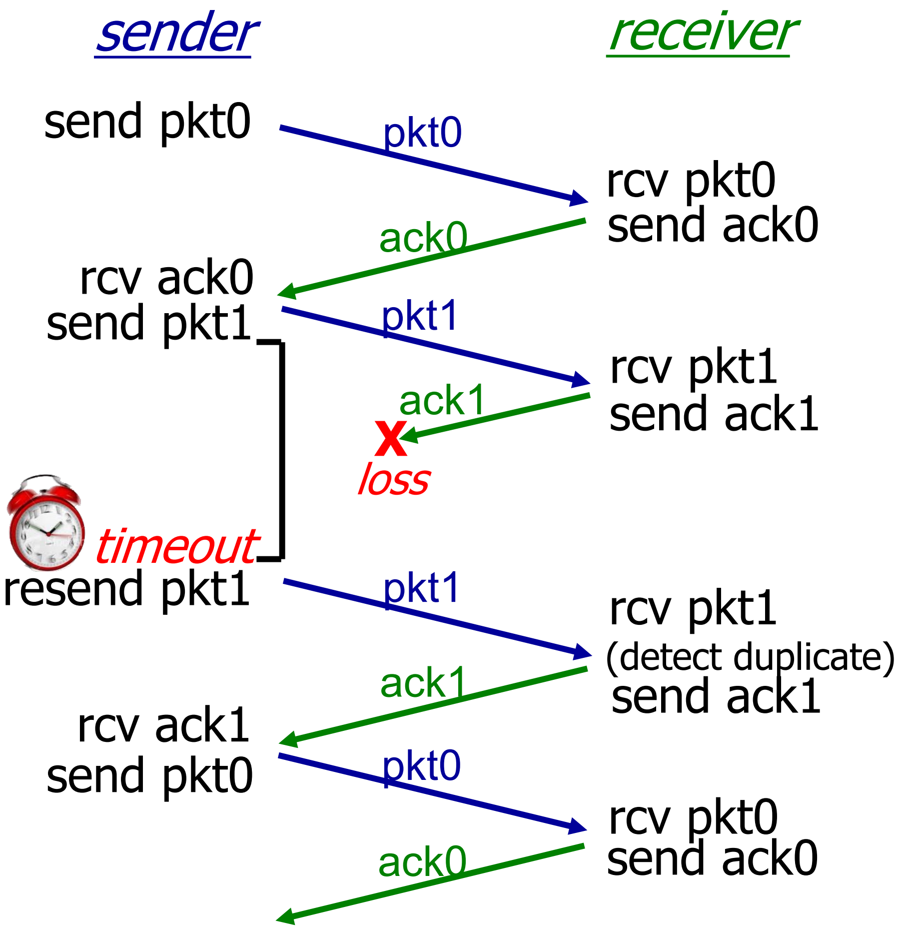
rdt3.0 in action



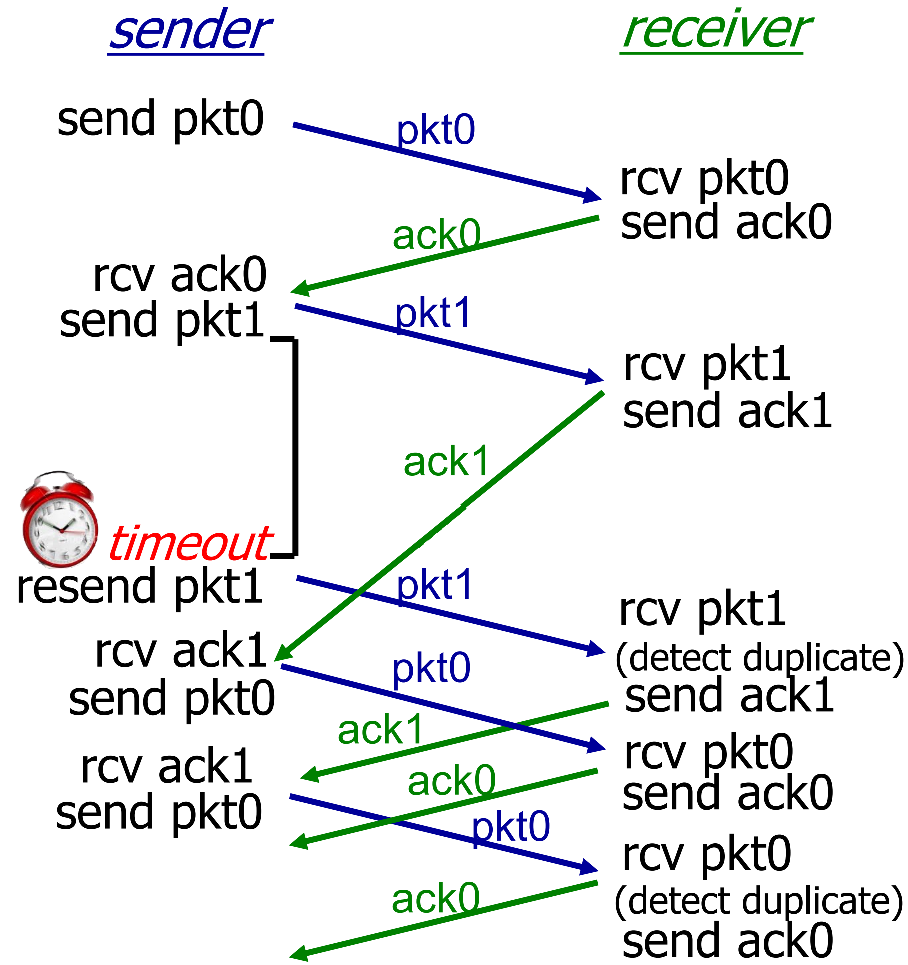
(a) no packet loss

(b) packet loss

rdt3.0 in action



(c) ACK loss



(d) premature timeout (=delayed ACK)
(please work though this yourself and convince yourself it works as it should!)

Performance of rdt3.0

- rdt3.0 is **correct**, but performance is terrible
- e.g.: 1 Gbps link, 15 ms prop. delay, 1KByte packet:

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

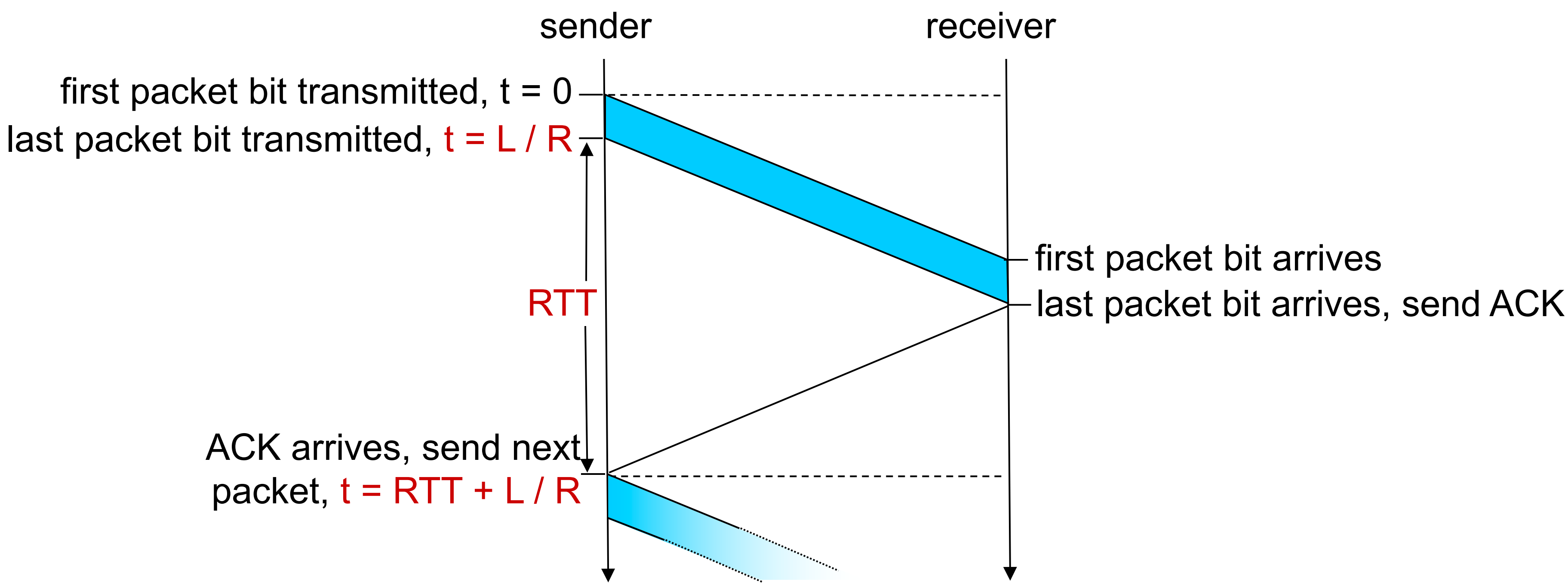
  - U_{sender}: *utilization* (i.e., fraction of time sender busy sending)

(n.b. RTT=30ms)

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

- so, 1KByte pkt every 30.008 msec
  - = 33kB/sec (approx) over a 1 Gbps link!
- protocol severely limits potential of network!

rdt3.0: stop-and-wait performs poorly

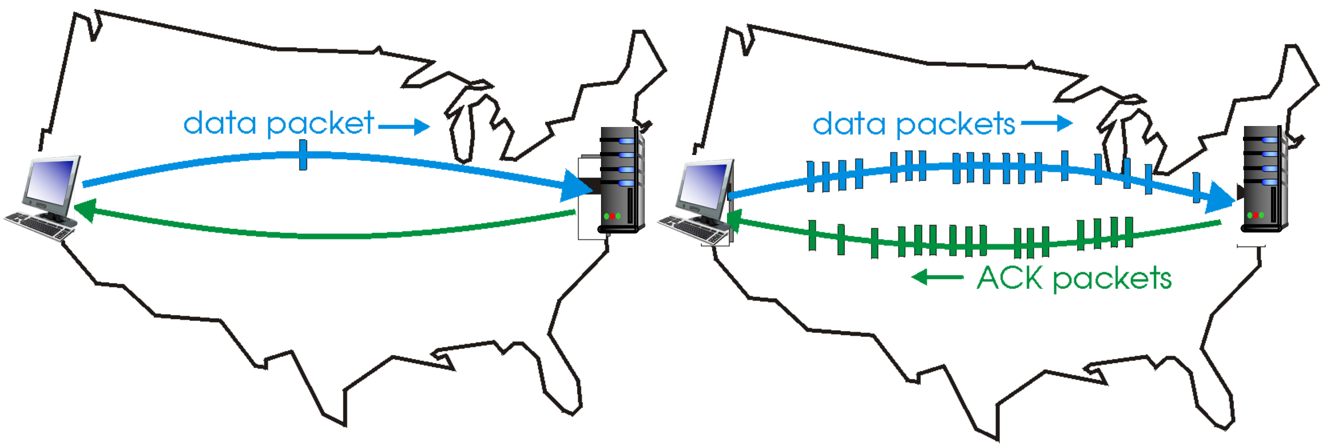


$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

Pipelined protocols

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

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

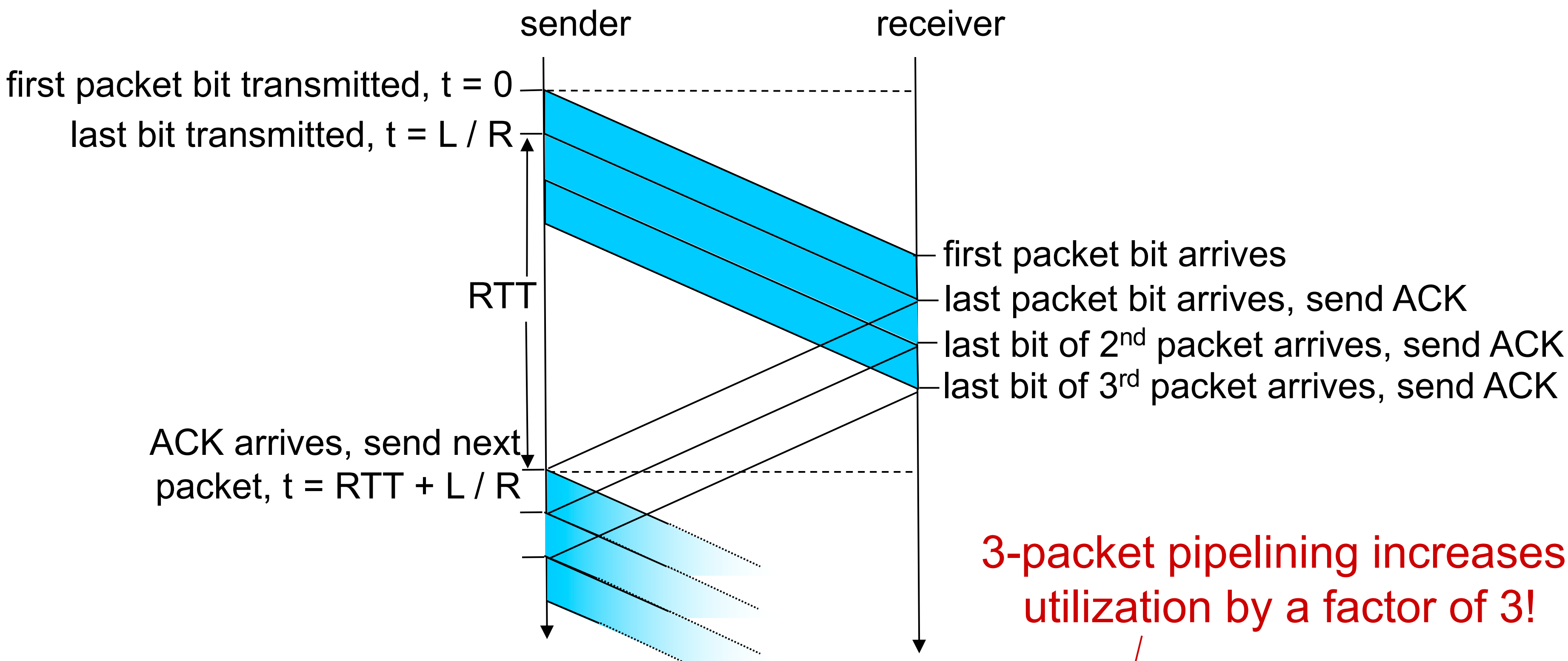


(a) a stop-and-wait protocol in operation

(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 may push up to N unacked packets into the pipeline ("window")
- receiver only sends *cumulative ack*
  - doesn't ack packet if there's a gap
- sender maintains a 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)

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

- segment structure (omit: RTT estimation)
- reliable data transfer
- flow control
- connection management

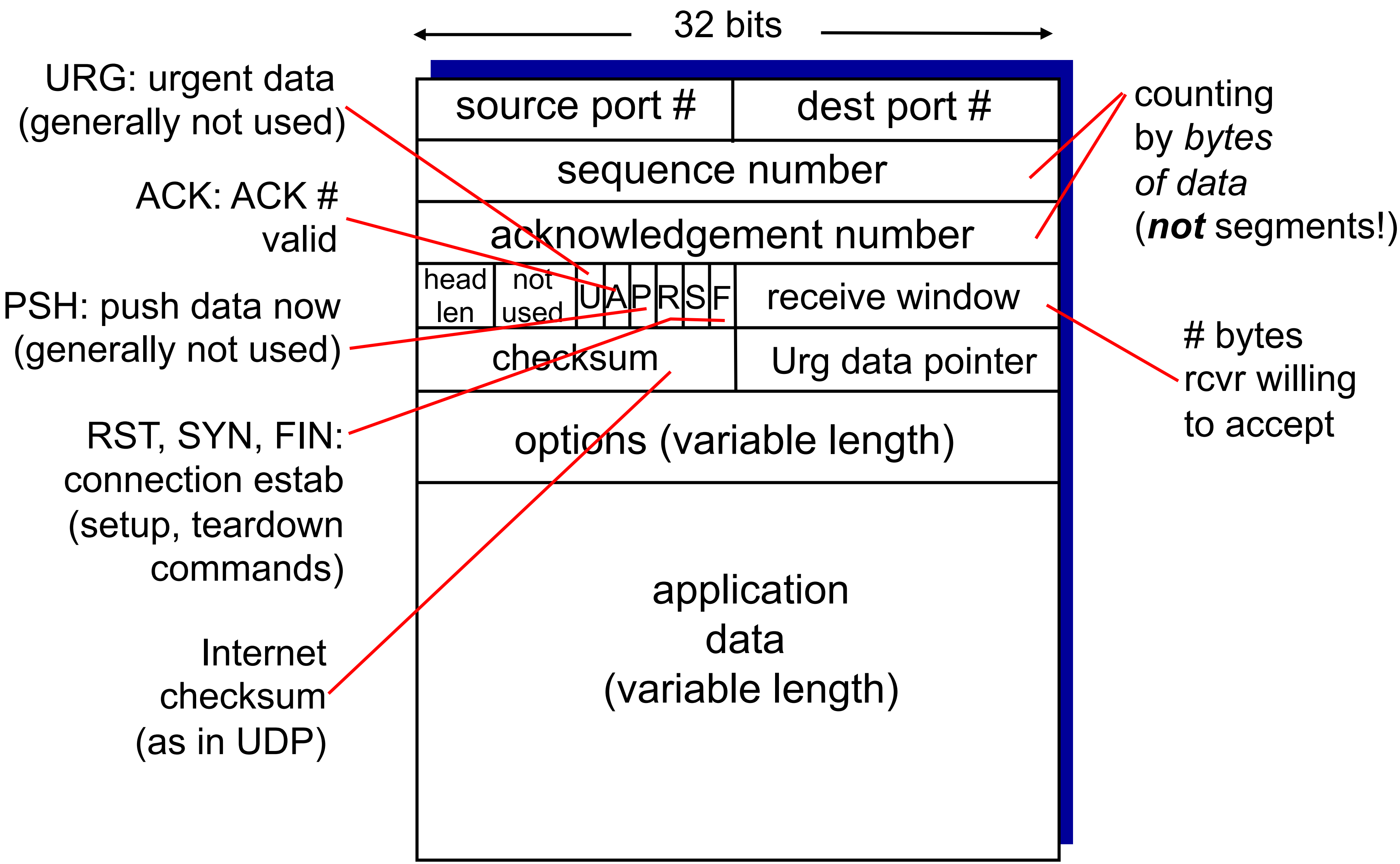
3.6 principles of congestion control

3.7 TCP congestion control

TCP: Overview RFCs: 793,1122,1323, 2018, 2581

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size
- full duplex data:
  - bi-directional data flow in same connection
- connection-oriented:
  - handshaking (exchange of control msgs) inits sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

TCP segment structure

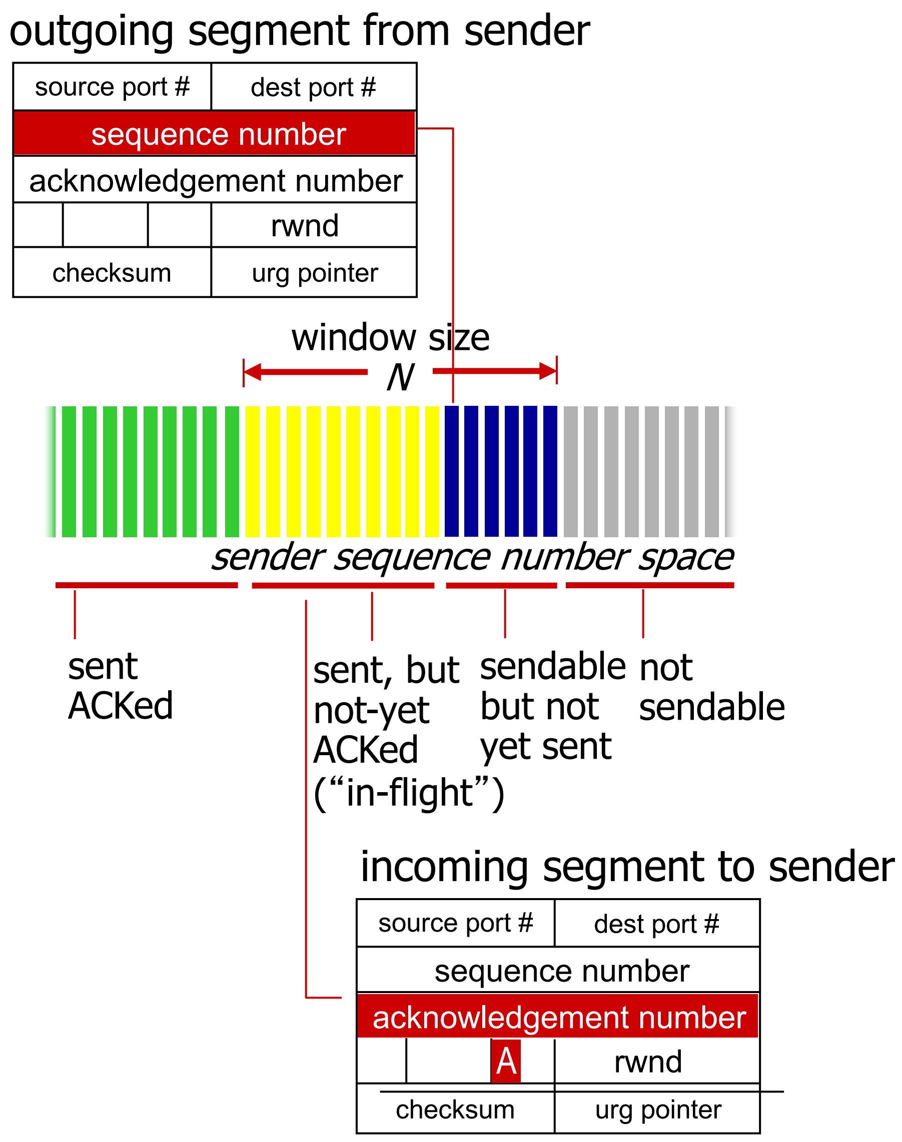


TCP seq. numbers, ACKs

sequence numbers:
- "index" of the first byte of a segment's data in the ongoing byte stream

acknowledgements:
- seq # of next byte expected from send side
- cumulative ACK

Q: how does receiver handle out-of-order segments?
- A: TCP spec doesn't say, - up to implementer



TCP round trip time, timeout

Q: how best to set TCP timeout value?

- *too short:* premature timeout, unnecessary retransmissions
- *too long*: slow reaction to segment loss
- certainly set it longer than RTT
  - but RTT varies…

Q: how to estimate RTT?

- **`SampleRTT`**: measured time from segment transmission until ACK receipt
- **`SampleRTT`** will vary; ideally, we want a "smoother" estimated RTT average
- so, incorporate several *recent* measurements, not just current **`SampleRTT`**

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

TCP fast retransmit (a quick example of one TCP optimization)

- time-out period is often relatively "long":
  - so, long delay before resending lost packet
- as we know, we detect corrupt/lost segments via duplicate ACKs
  - sender often sends many segments back-to-back
  - if segments are lost, gaps will likely cause many duplicate ACKs

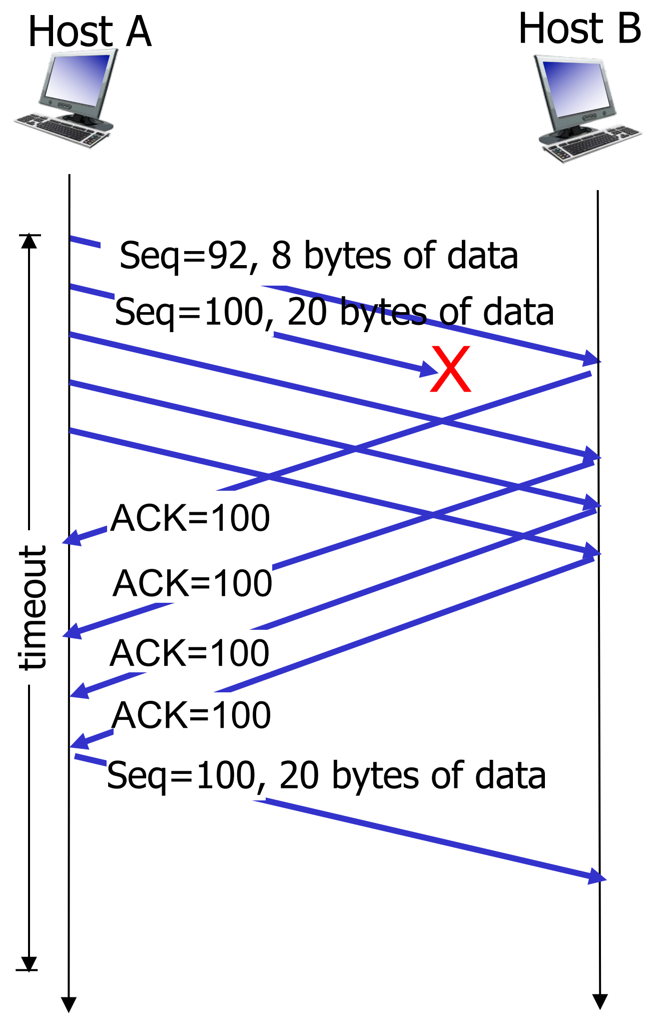
TCP fast retransmit

if sender receives 3 ACKs for same data ("triple duplicate ACKs"),

=> immediately resend unACKed segment with smallest seq #

- likely that unacked segment was lost, so don't wait for timeout

TCP fast retransmit



fast retransmit after sender receipt of triple duplicate ACK

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control (OMITTED)
- connection management (OMITTED)

3.6 principles of congestion control (OMITTED)

3.7 TCP congestion control (OMITTED)

Brief word on the omitted topics in section 3.5

- TCP connection management
  - establishing the basis for communication – two sides exchange initial state, and each becomes aware that the other is connected
  - employs 3-way (not 2-way) handshake
- TCP flow control
  - preventing a sender swamping a receiver
  - use of rwnd field
- principles of congestion and TCP congestion control
  - different from flow control – congestion control is about avoiding swamping the network
  - "slow start" mechanism
  - big topic!

Summary

- we've seen the development of an abstract protocol that ensures 100% reliability in the face of corrupted and/or lost packets
- TCP builds on these ideas and adds many practical features
  - connection management
  - flow and congestion control
  - fast retransmit
- there's a lot to TCP that we have not yet seen!
  - see sections 3.5, 3.6 and 3.7 of the book