Course on Computer Communication and Networks

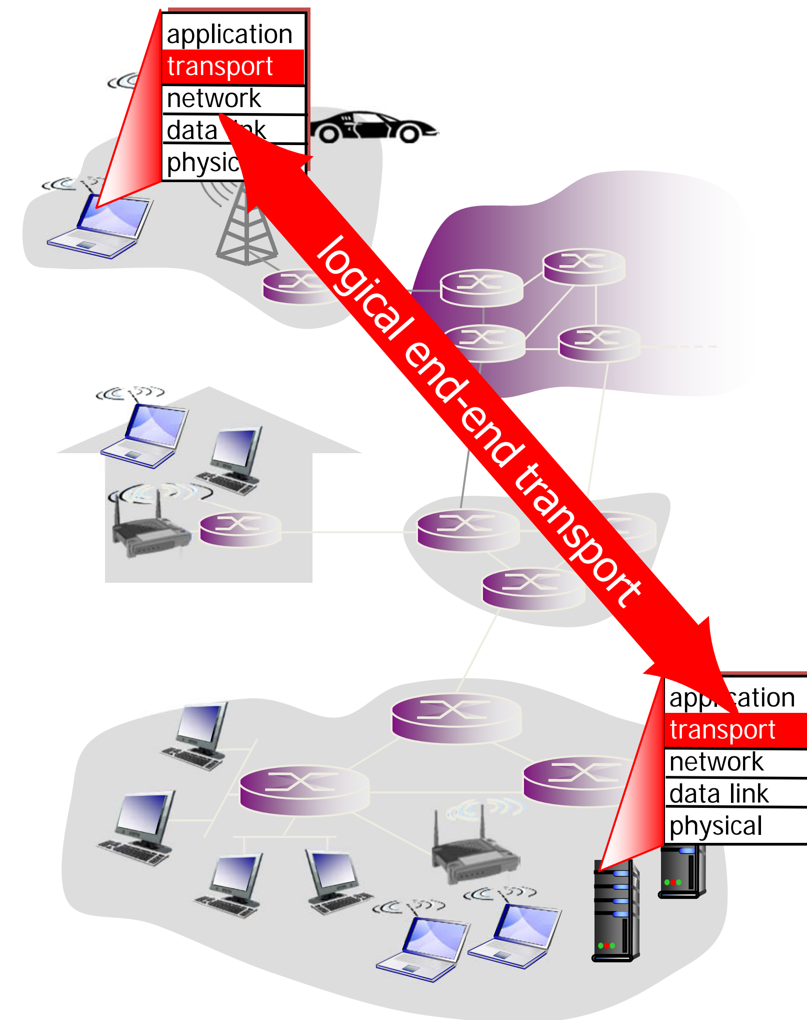
Lecture 4
Chapter 3; Transport Layer, Part A

EDA344/DIT 420, CTH/GU

Based on the book Computer Networking: A Top Down Approach, Jim Kurose, Keith Ross, Addison-Wesley.

Transport services and protocols

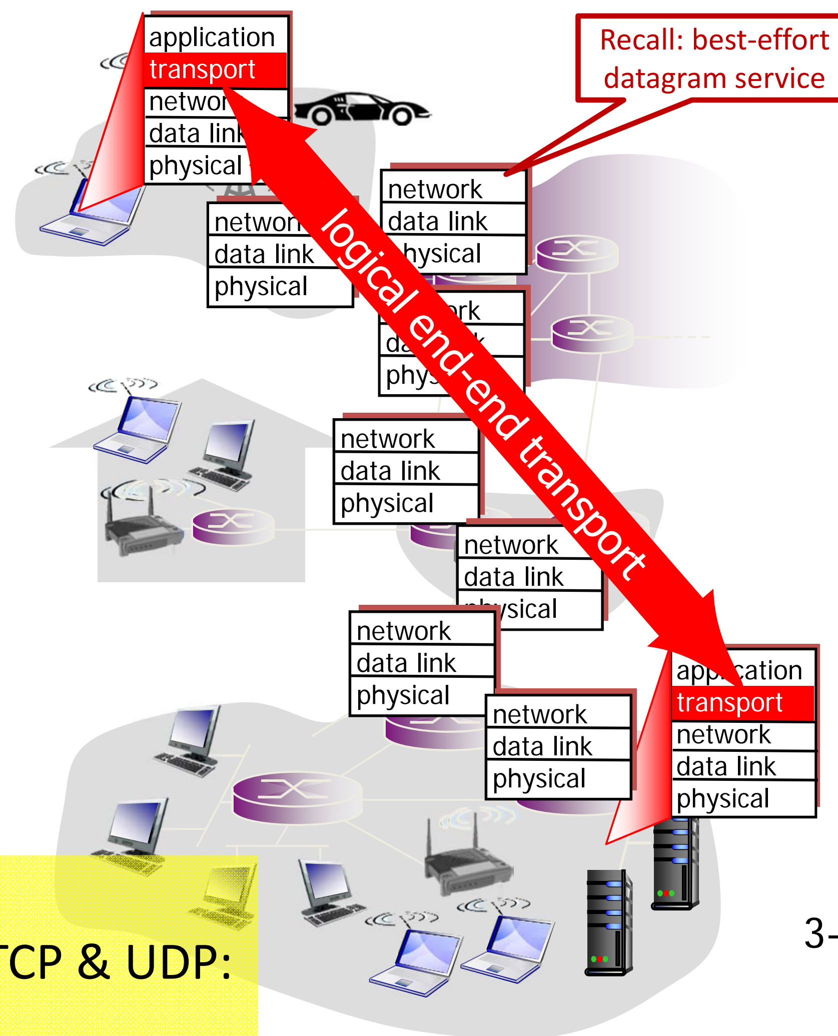
- provide logical communication to app processes
- transport protocols run in end systems
  - send side: breaks app messages into segments, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
- transport protocols available to apps
  - Internet: TCP and UDP



Internet transport-layer protocols

- reliable, in-order delivery: **TCP**; also provides
  - flow control
  - congestion control
  - connection setup

- unreliable, unordered delivery: **UDP**
  - no-frills extension of "best-effort" IP



Both support addressing (multiplexing)
Transport Layer services **not available** in TCP & UDP:
Delay, bandwidth guarantees

Transport Layer: Learning goals:

- understand principles behind transport layer services:
  - Addressing, multiplexing/demultiplexing
  - reliable data transfer
  - flow control
  - congestion control (no;t strictly Transport layer issue --*some study now; more in connection with RealTime traffic*)
- instantiation and implementation in the Internet

Roadmap

- Transport layer services in Internet
- Addressing, multiplexing/demultiplexing
- Connectionless, unreliable transport: UDP
- principles of reliable data transfer

- *Next lecture: connection-oriented transport: TCP*
  - *reliable transfer*
  - *flow control*
  - *connection management*
  - *TCP congestion control*

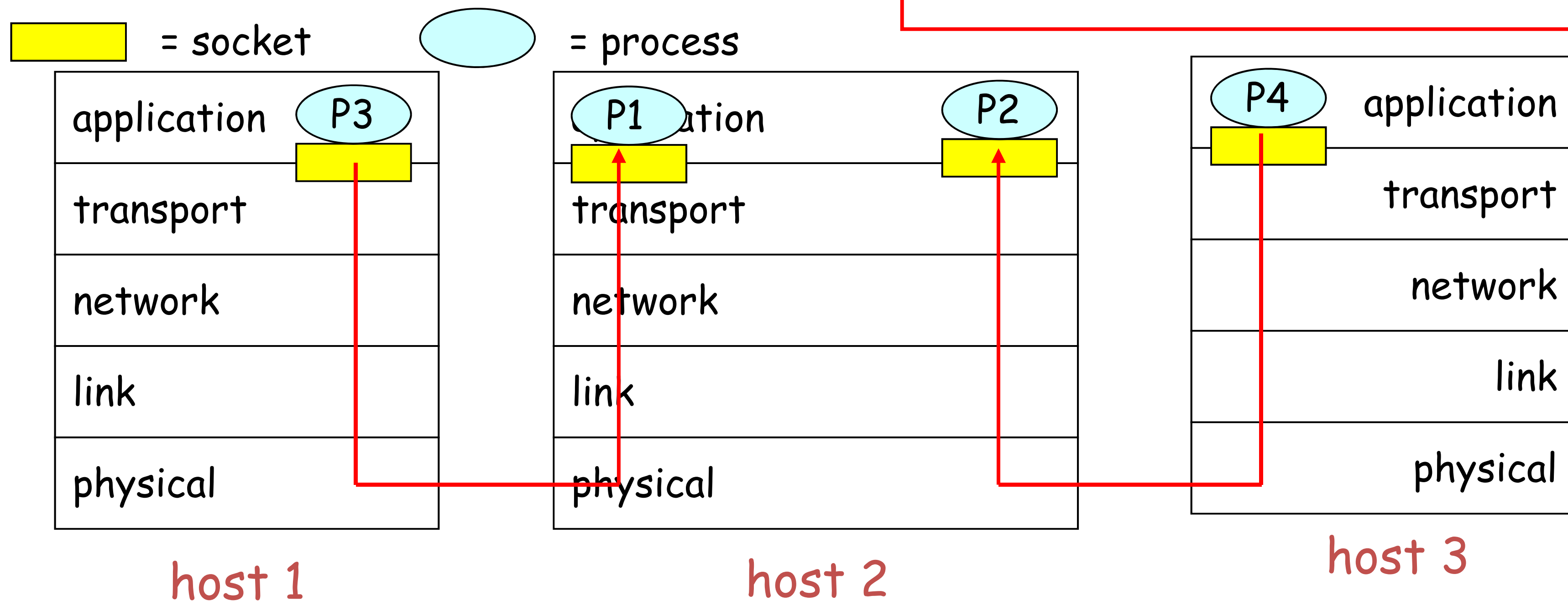
Addressing: Multiplexing/demultiplexing

Demultiplexing at rcv host:

delivering received segments to correct socket

Multiplexing at send host:

gathering data, enveloping data with header (later used for demultiplexing)

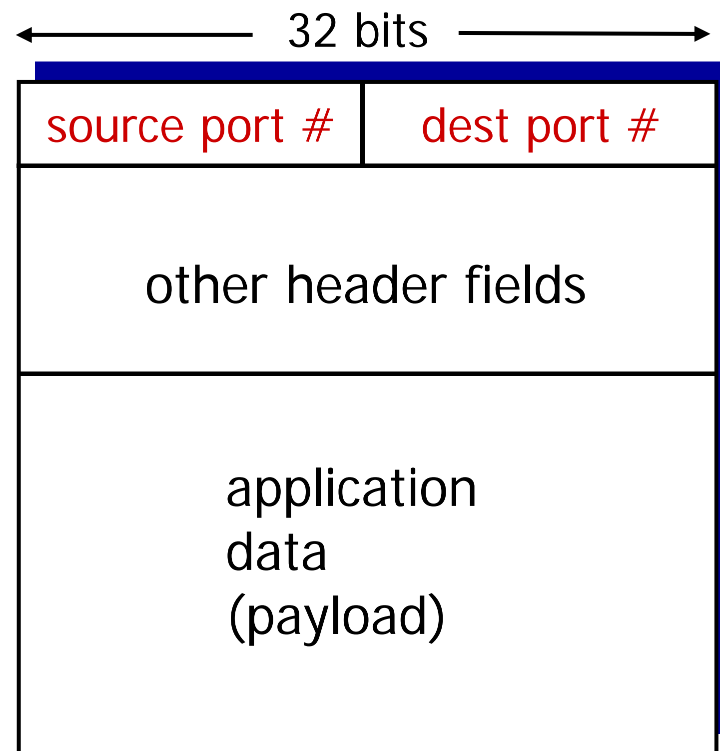


Recall: *segment* - unit of data exchanged between transport layer entities

aka TPDU: transport protocol data unit

Addressing

- Host receives IP datagrams
  - datagram has source IP address, destination IP address
  - datagram carries transport-layer segment
  - segment has source, destination port number
- Host uses *IP addresses & port numbers* to direct segment to appropriate socket



TCP/UDP segment format

UDP addressing - demultiplexing

- when creating datagram to send, must specify:
  - destination IP address
  - destination port #

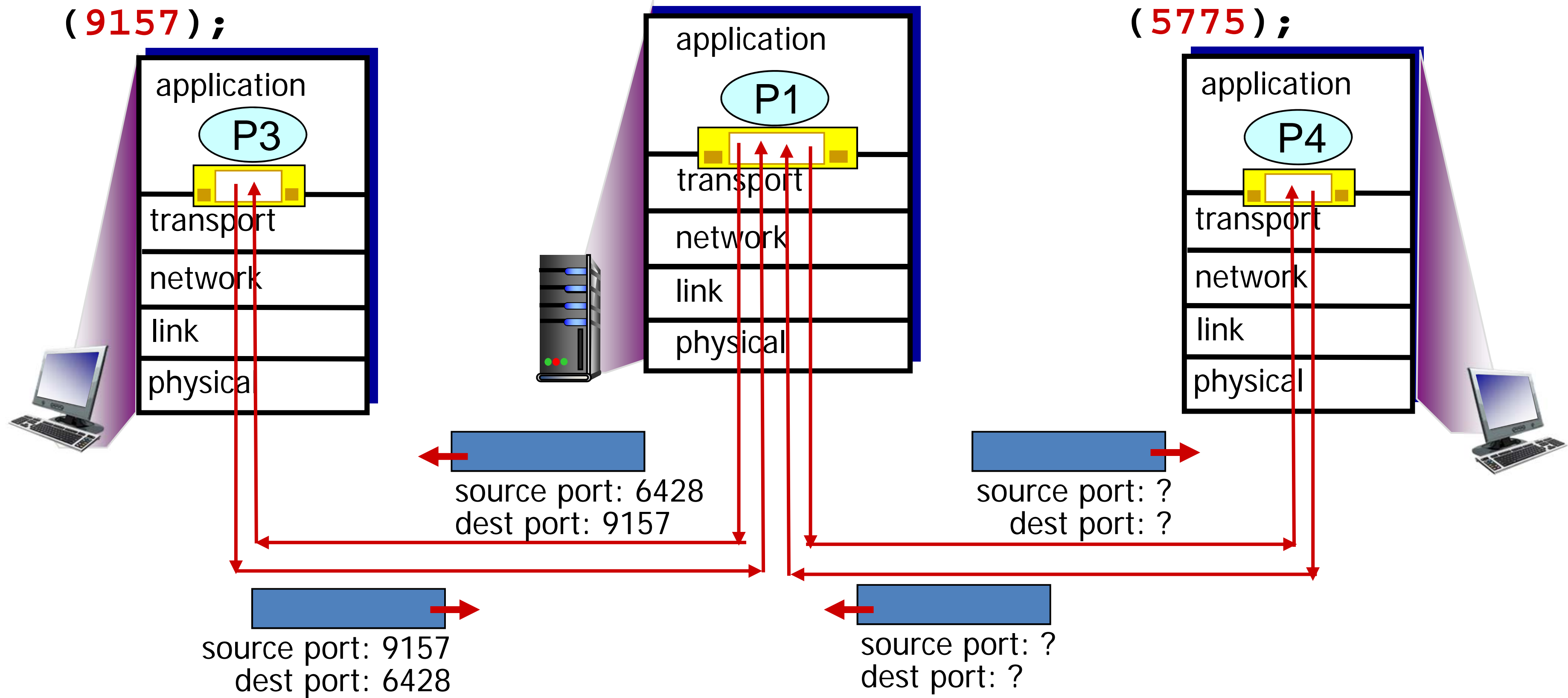
created socket has host-local port #, eg:

```
DatagramSocket mySocket1 = new
 DatagramSocket(12534);
```

- when host receives UDP datagram:
  - checks destination port # in datagram
  - directs UDP datagram to socket with that port #

IP datagrams with *same dest. port #,* but different source IP addresses and/or source port numbers will be directed *to the same socket*

UDP : example

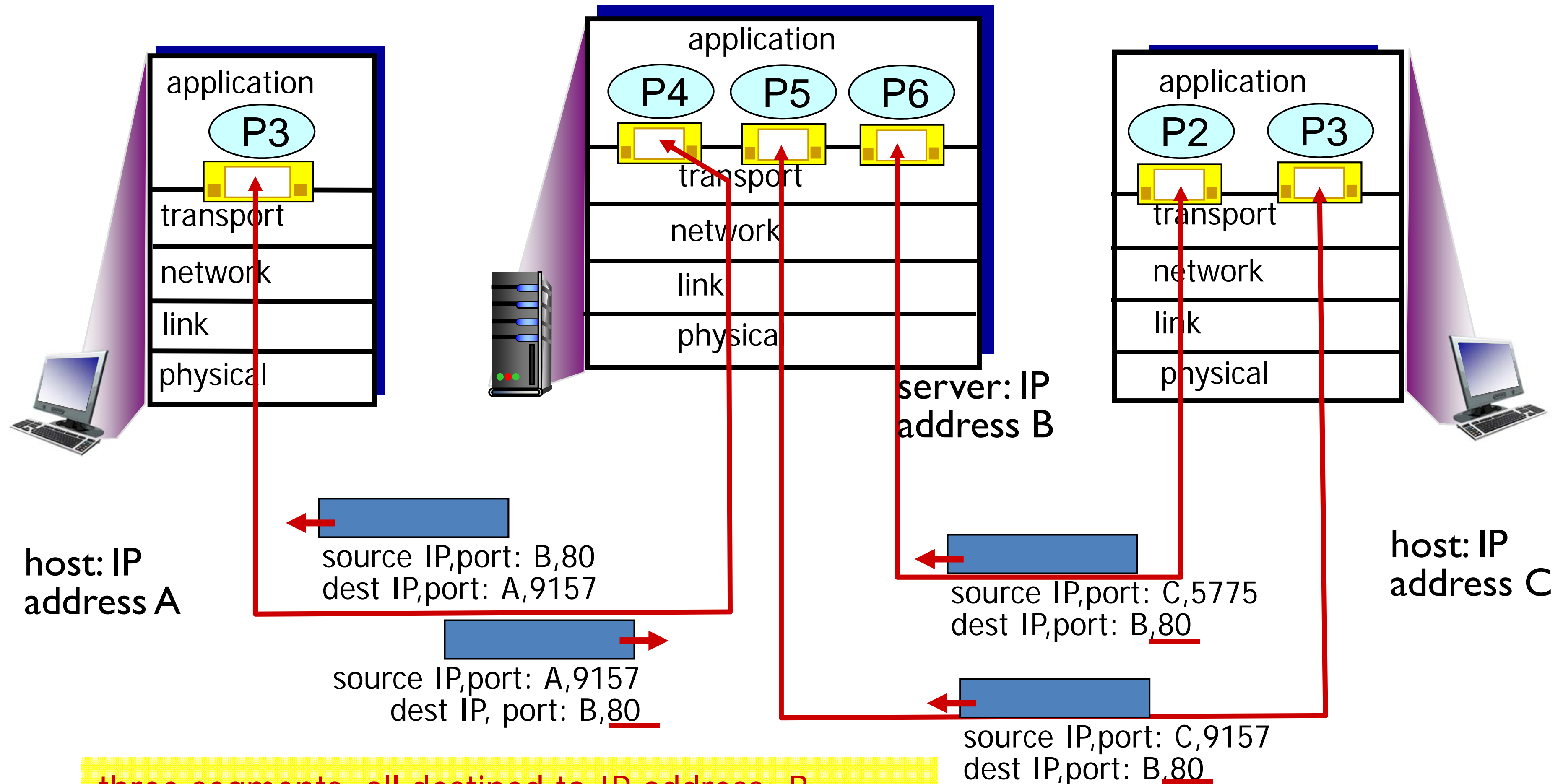


Connection-oriented (TCP) addressing/demux

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver uses all four values to direct segment to appropriate socket

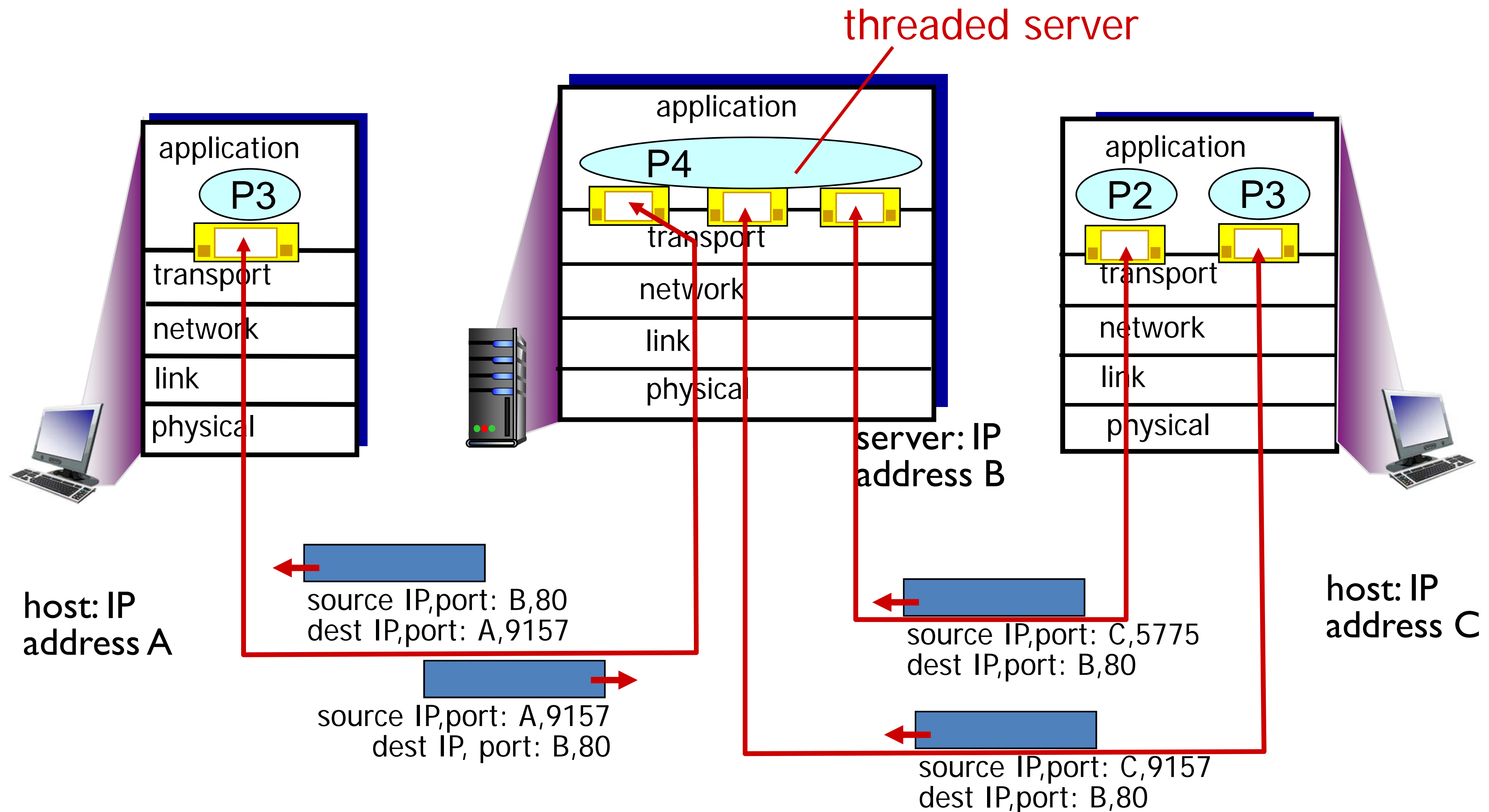
- server host may support many simultaneous TCP sockets:
  - one socket per connection
  - each socket identified by its own 4-tuple
- web servers have different sockets for each connecting client
  - non-persistent HTTP will even have different sockets for each request

TCP: example



three segments, all destined to IP address: B,
dest port: 80 are demultiplexed to *different* sockets

TCP demux: Threaded web server



Roadmap

- Transport layer services
- Addressing, multiplexing/demultiplexing
- Connectionless, unreliable transport: UDP
- principles of reliable data transfer

- *Next lecture: connection-oriented transport: TCP*
  - *reliable transfer*
  - *flow control*
  - *connection management*
  - *TCP congestion control*

UDP: User Datagram Protocol [RFC 768]

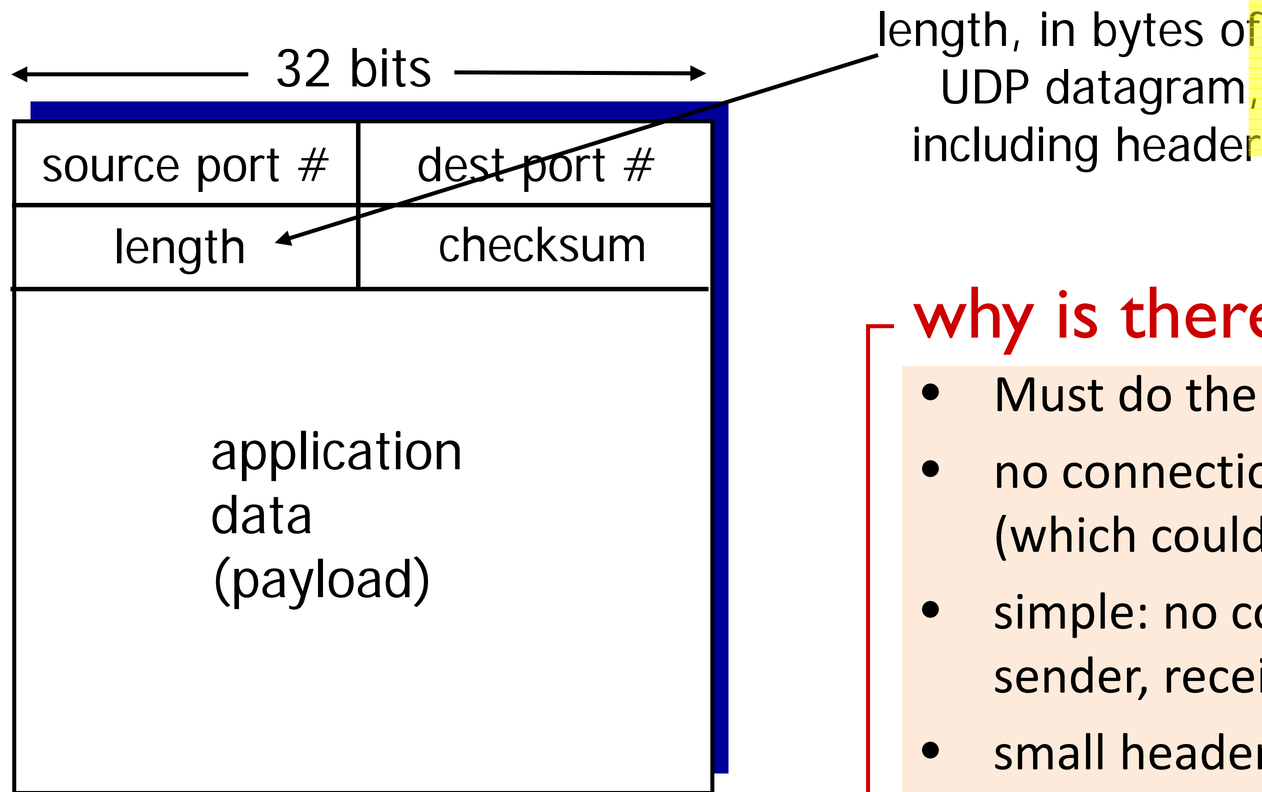
"best effort" service, UDP segments may be:

- lost
- delivered out-of-order

- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

UDP: segment header

UDP use:
- DNS
- SNMP
- More discussion later…



UDP datagram format

why is there a UDP?

- Must do the addressing job
- no connection establishment (which could add delay)
- simple: no connection state at sender, receiver
- small header size
- no congestion control: UDP can blast away as fast as desired

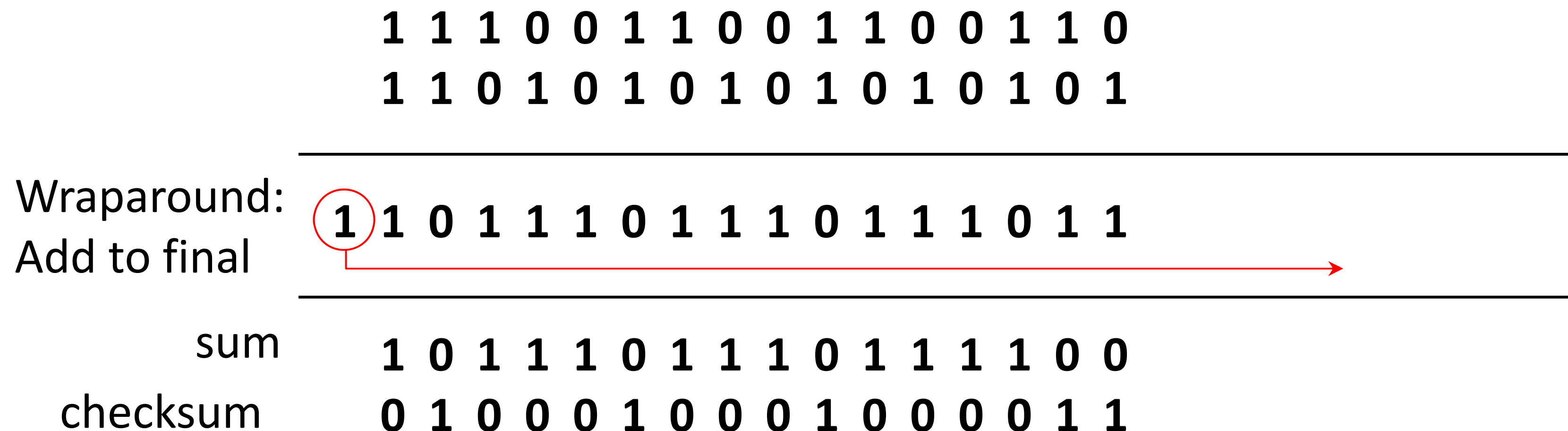
UDP Checksum[RFC 1071]: check bit flips

Sender:

- treat segment contents as sequence of 16-bit integers
- checksum: addition (1's complement sum) of segment contents
- sender puts checksum value into UDP checksum field

Receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected (*report error to app or discard*)
  - YES - no error detected.
    - *But maybe (rarely) errors nonethless?* More later ….



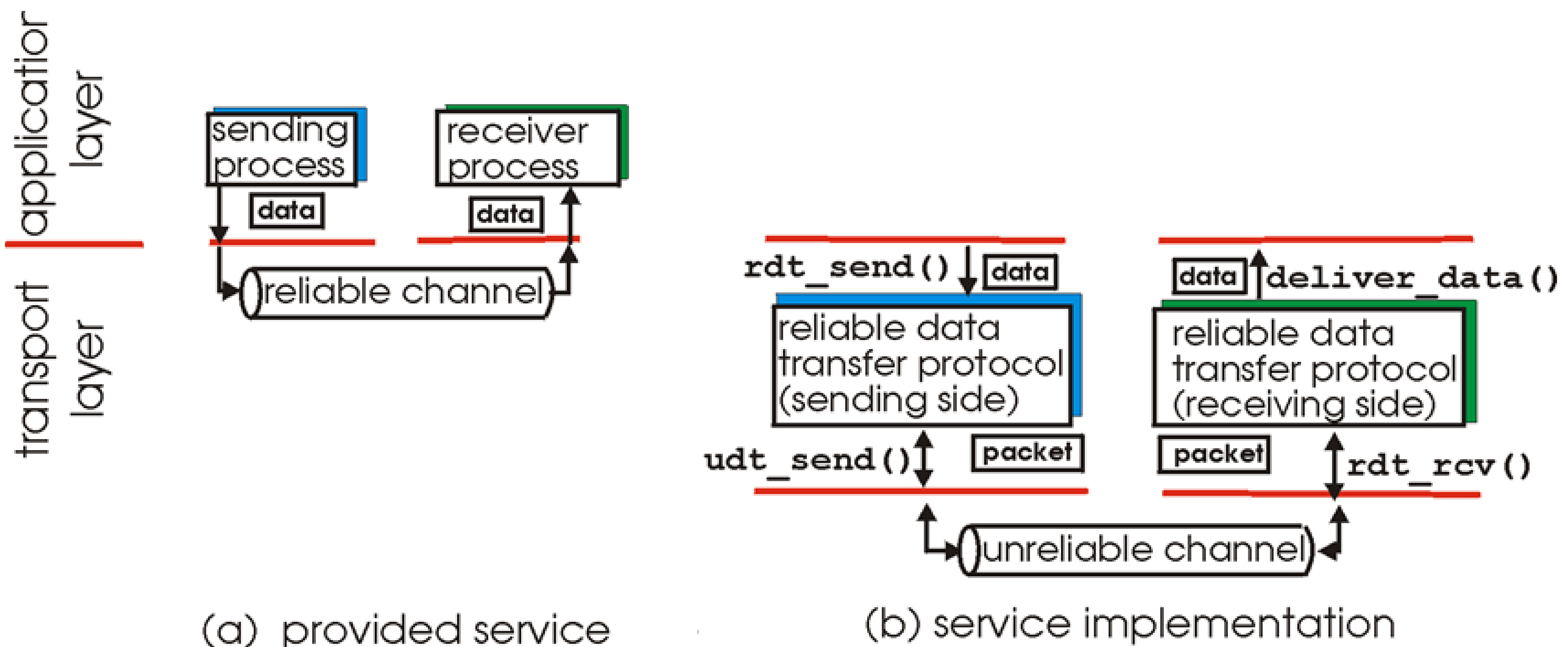
Roadmap

- Transport layer services
- Addressing, multiplexing/demultiplexing
- Connectionless, unreliable transport: UDP
- principles of reliable data transfer

- *Next lecture: connection-oriented transport: TCP*
  - *reliable transfer*
  - *flow control*
  - *connection management*
  - *TCP congestion control*

Principles of reliable data transfer

top-10 list of important networking topics!



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

Is it possible to have reliable transfer over UDP?

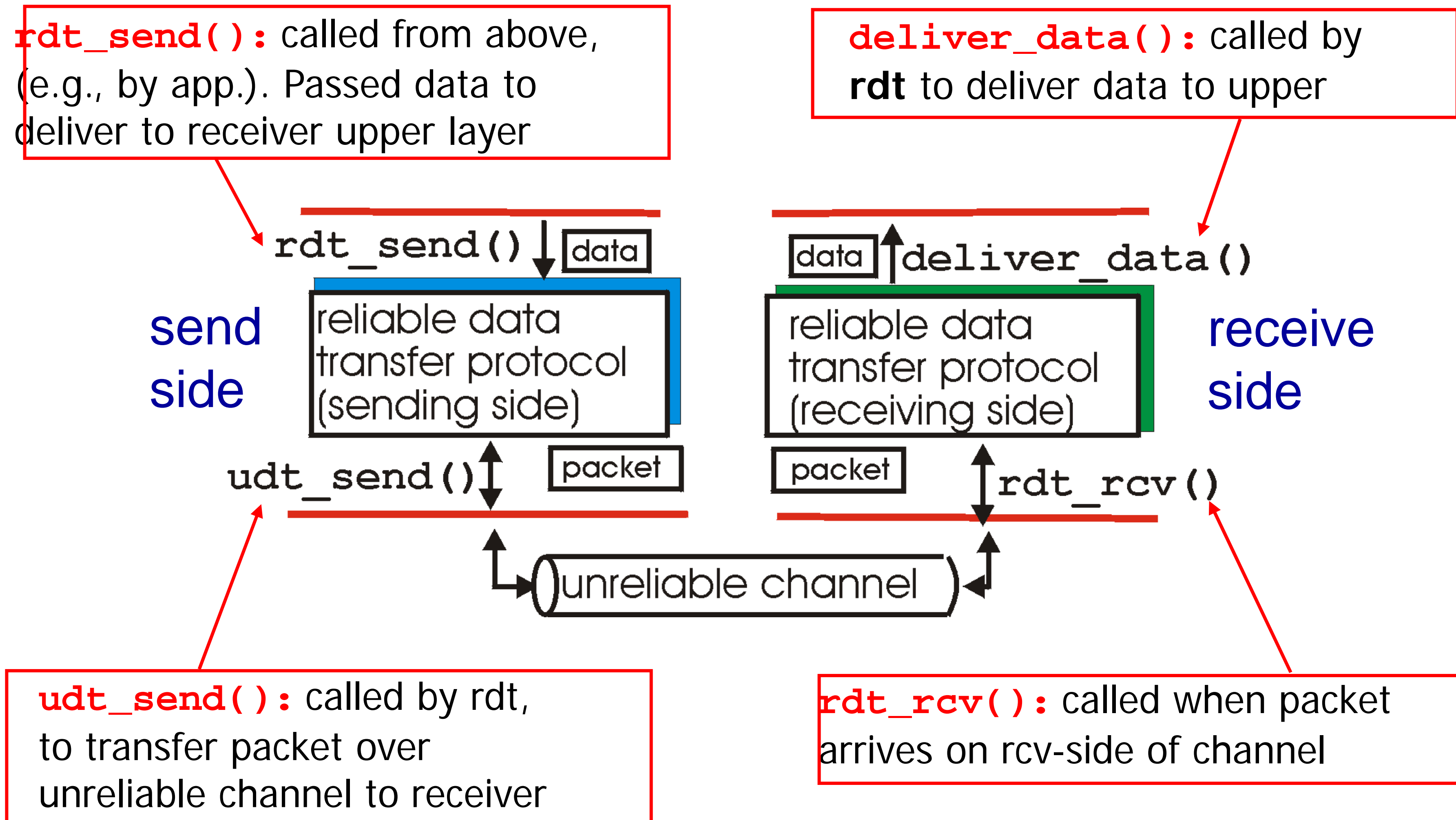
http://poll.fm/54om5

reliable transfer over UDP?

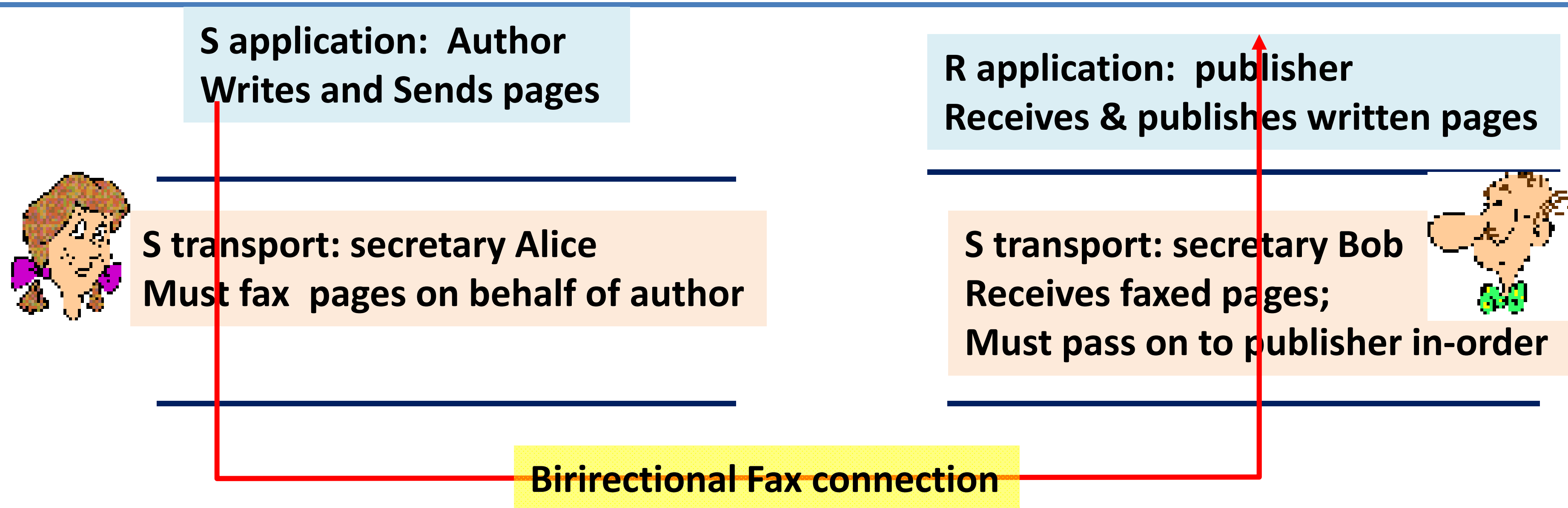
- add reliability at application layer
- application-specific error recovery

- But best to: not "reinvent "TCP on top of UDP
  - If you need TCP, use TCP

Reliable data transfer (RDT): getting started



RDT:



A sends one page at a time;

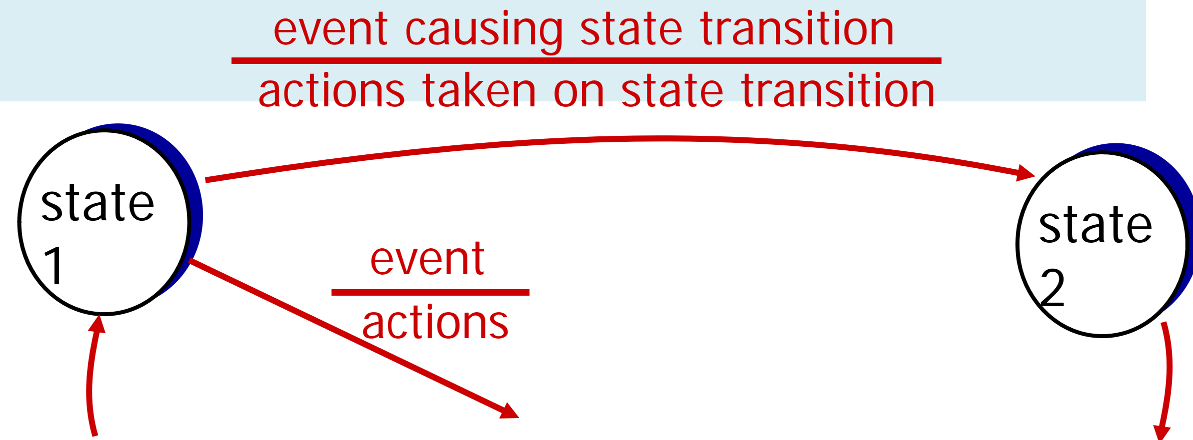
How do A & B do their job if fax connection…

- …is reliable?
- …might introduce errors?
- …might lose fax-messages?

Reliable data transfer: getting started

we'll:

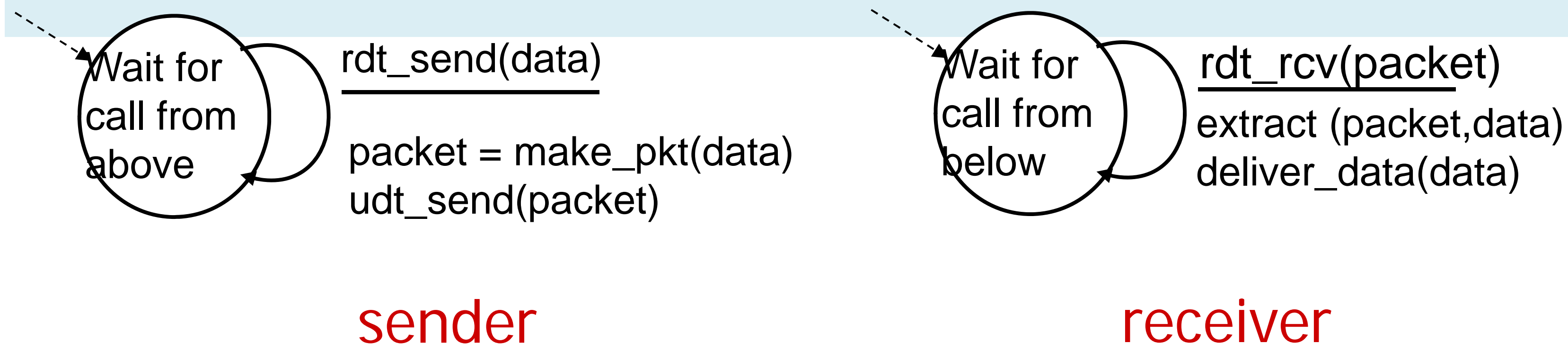
- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow on both directions!
- use **finite state machines (FSM)** to specify sender, receiver



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

rdt1.0: reliable transfer & reliable channel

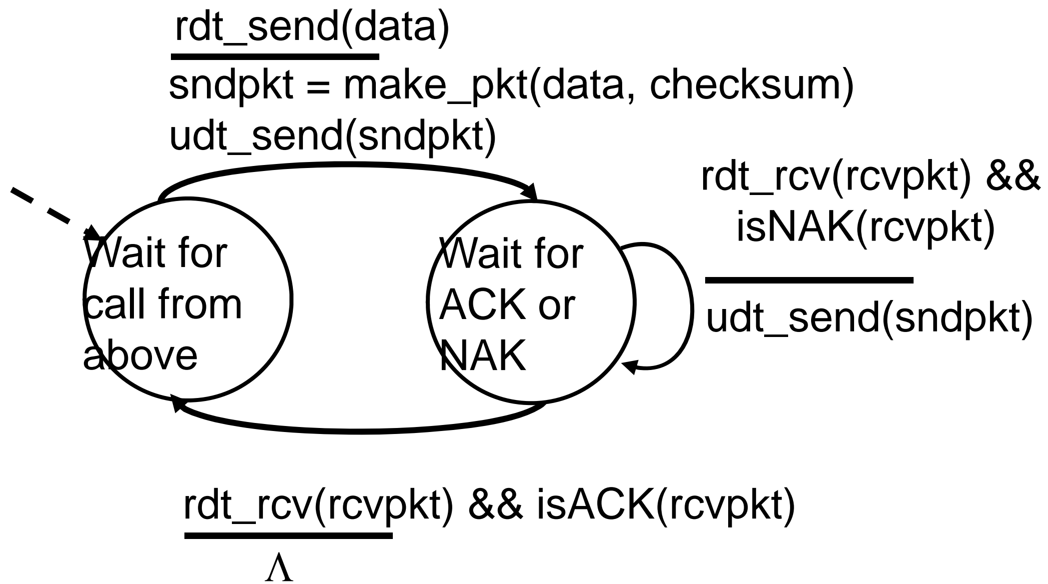
- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver reads data from underlying channel



rdt2.0: channel with bit errors

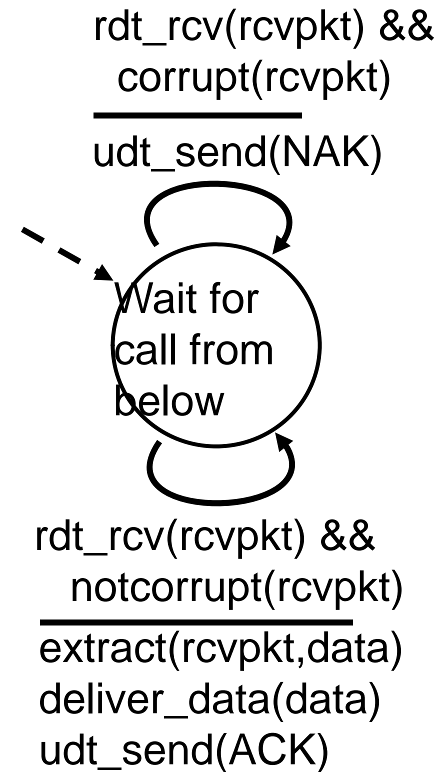
- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors:
  - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors
  - 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

rdt2.0: FSM specification

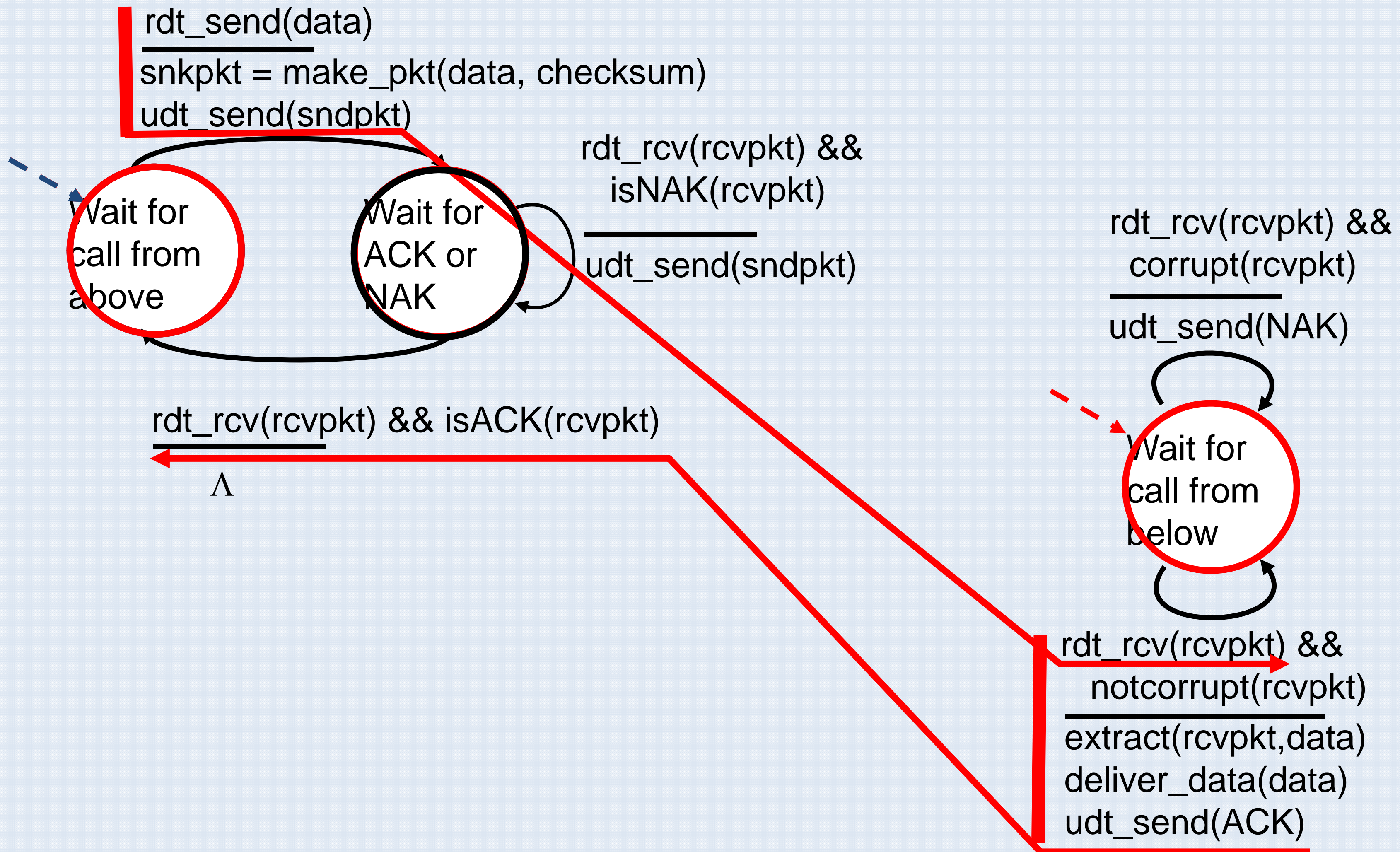


sender

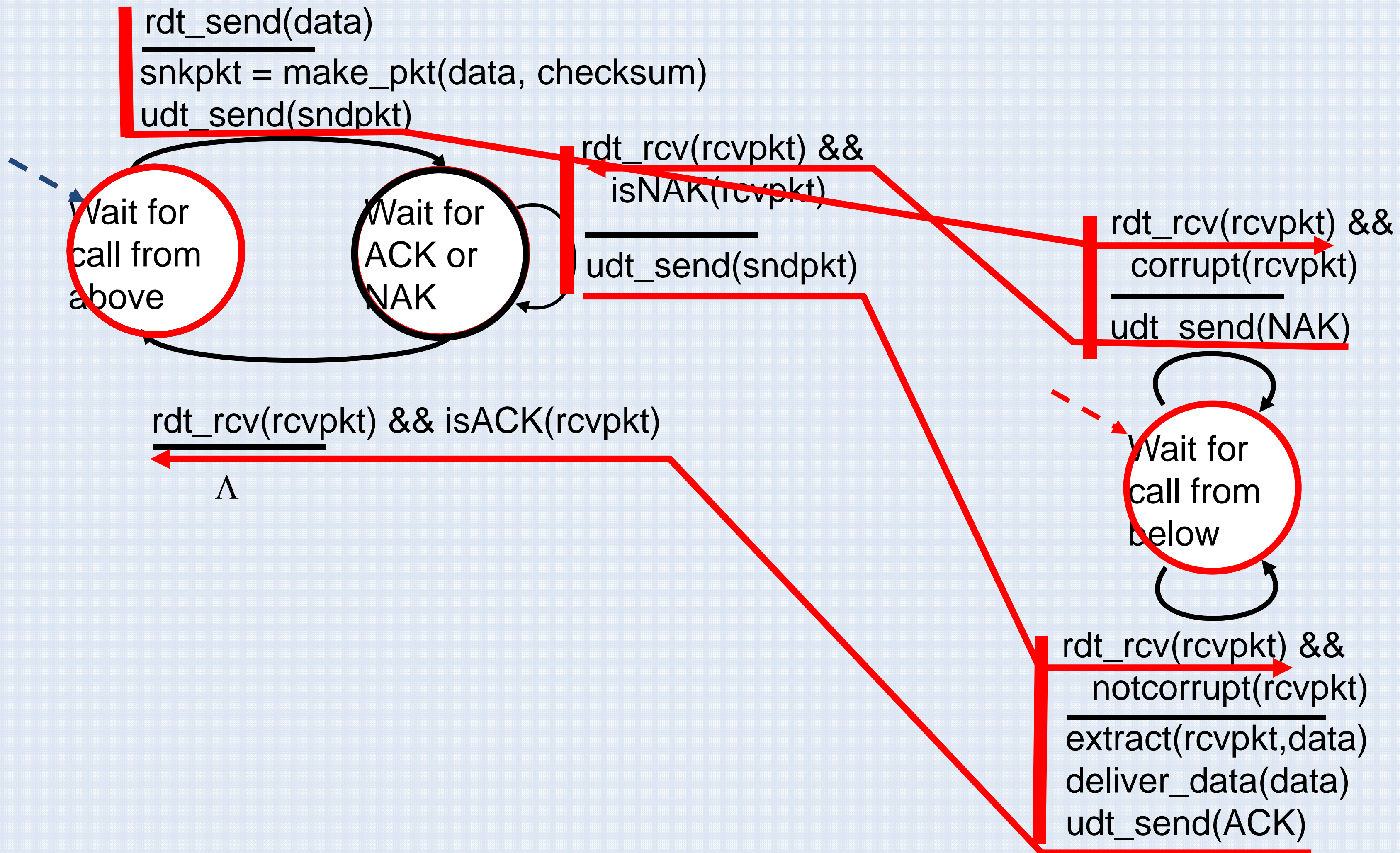
receiver



rdt2.0: operation with no errors



rdt2.0: error scenario



rdt2.0 has an issue ....

what happens if ACK/NAK corrupted?

- sender uncertain what happened at receiver!
- **Can retransmit : BUT WATCH: possible duplicates!**

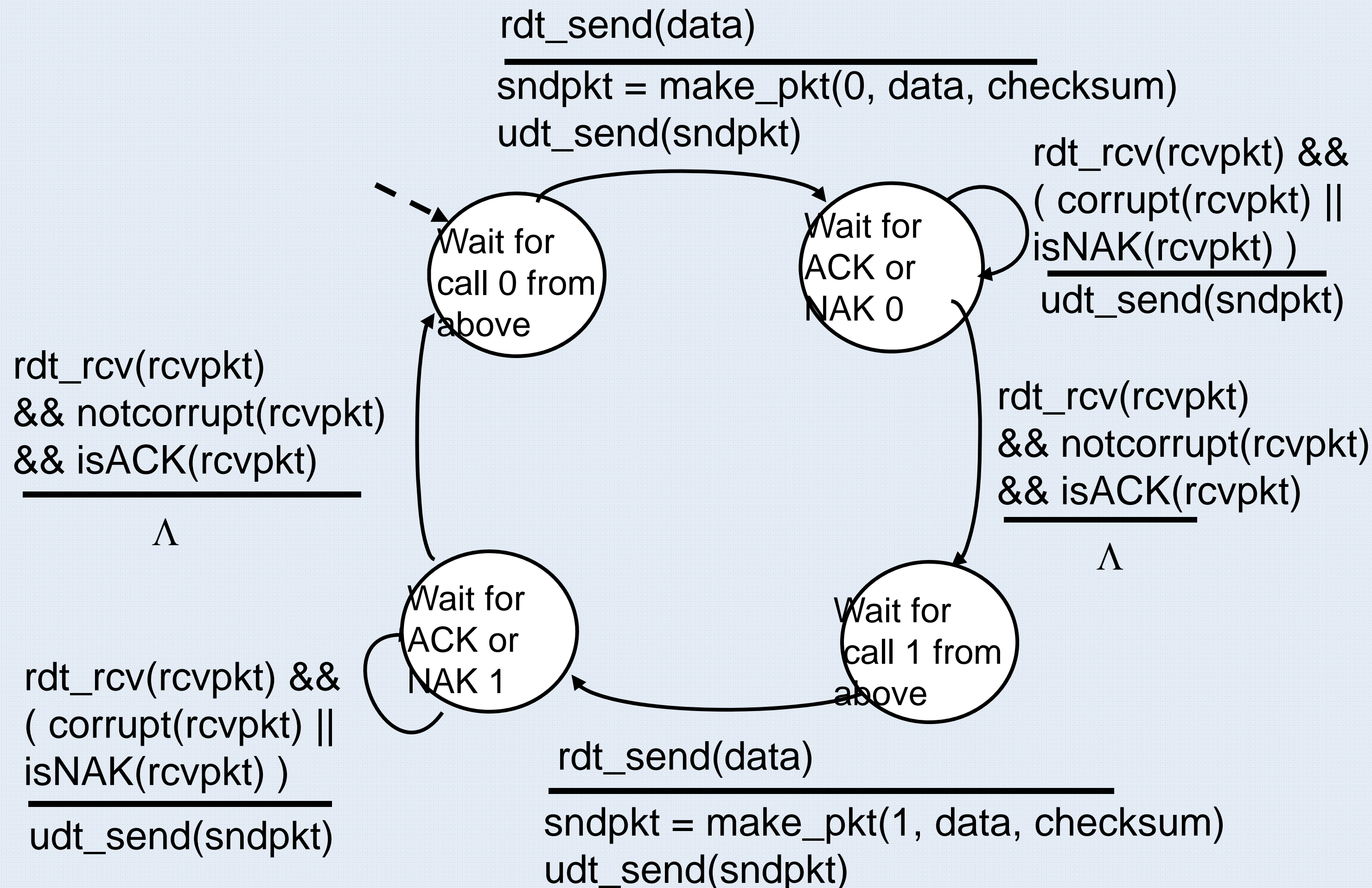
handling duplicates:

- sender retransmits current pkt if hears nothing
- sender adds *sequence number* to each pkt
- receiver discards (doesn't deliver up) duplicate pkt

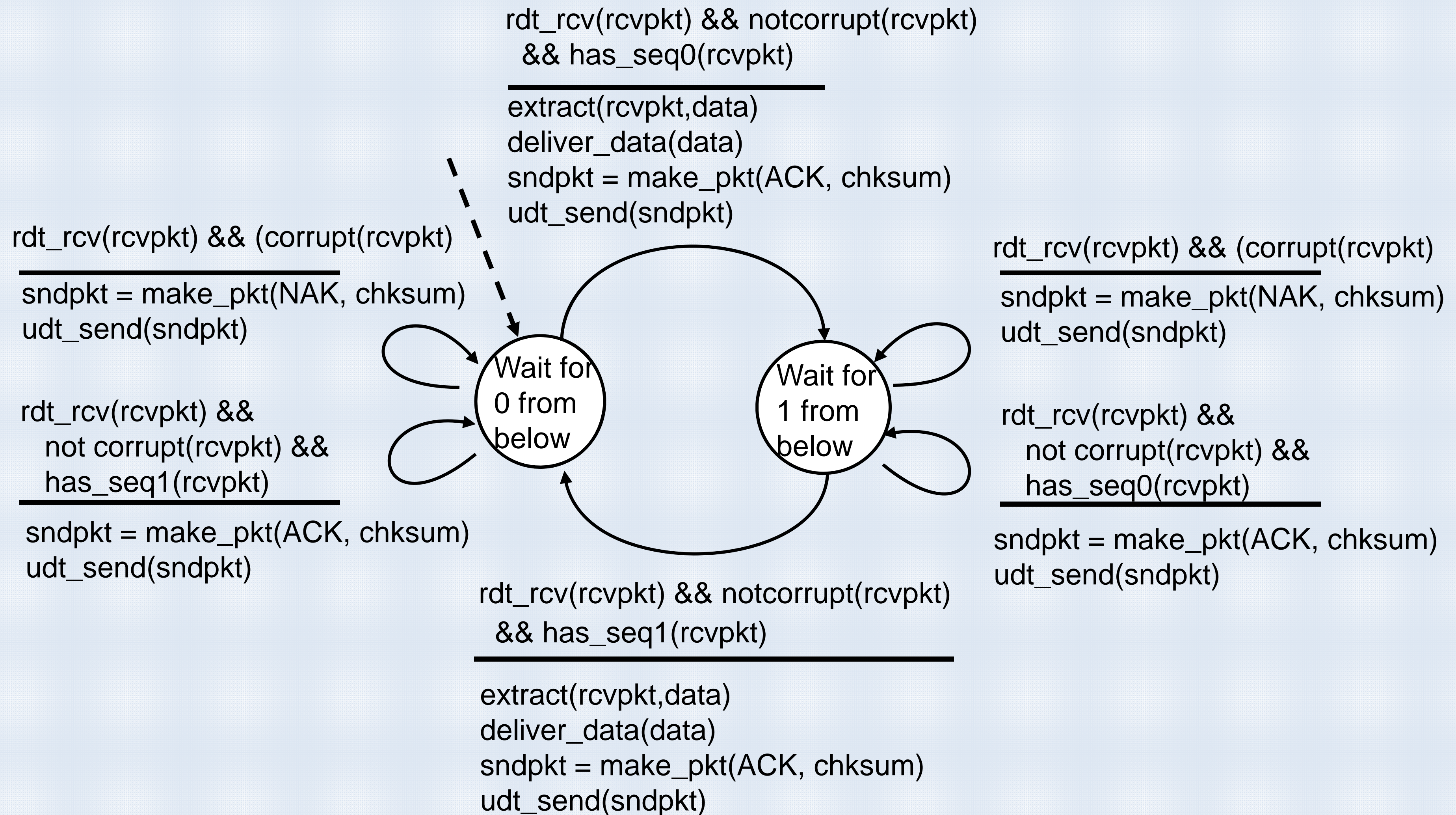
stop and wait

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 "expected" pkt should have seq # 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

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*

rdt3.0: channels with errors *and* loss

new assumption: underlying channel can also **lose packets** (data, ACKs)

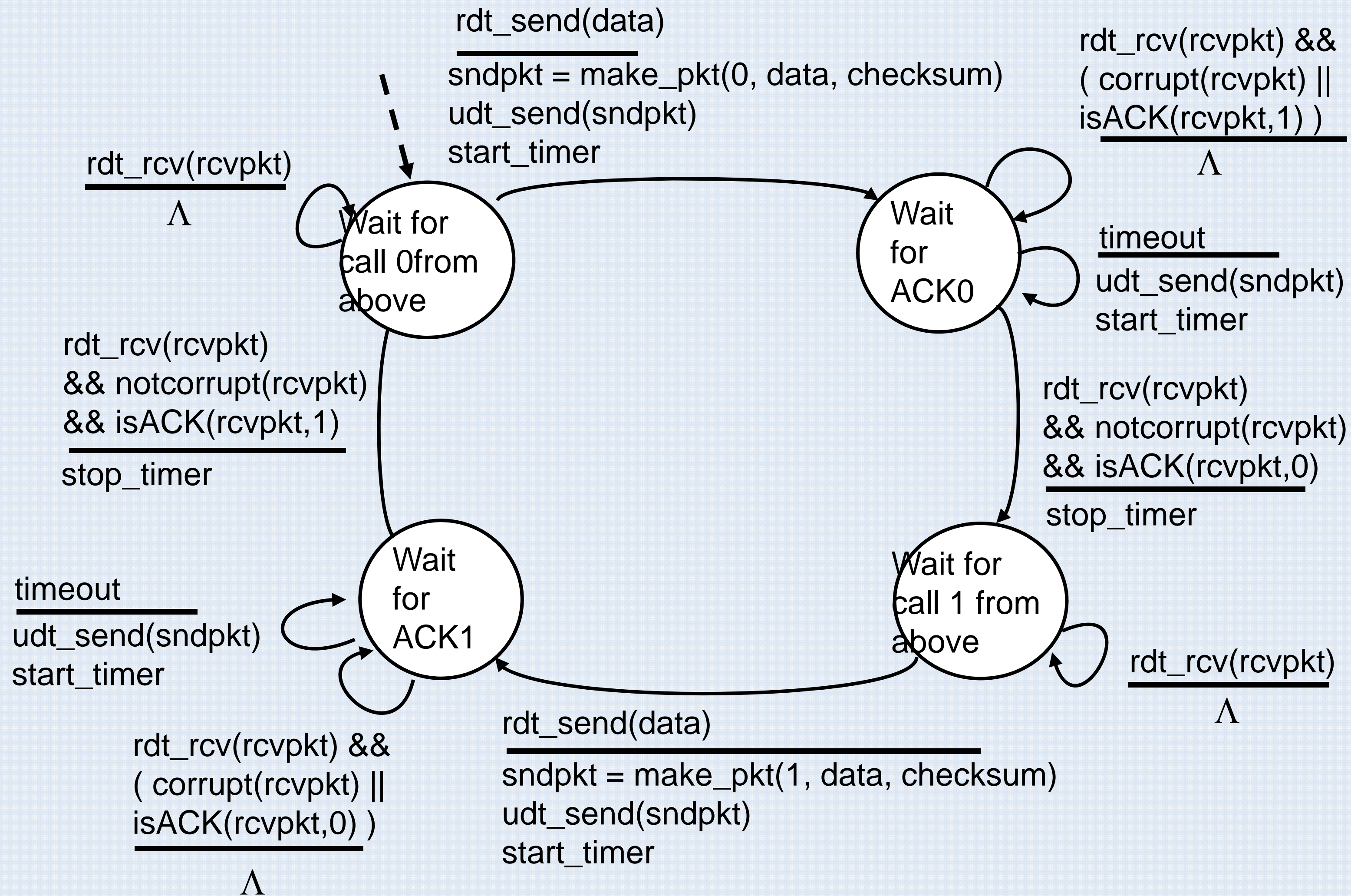
approach: sender waits "reasonable" amount of time for ACK

- retransmits if no ACK received in this time
- requires countdown timer
- if pkt (or ACK) just delayed (not lost):
  - Must handle duplicates ->

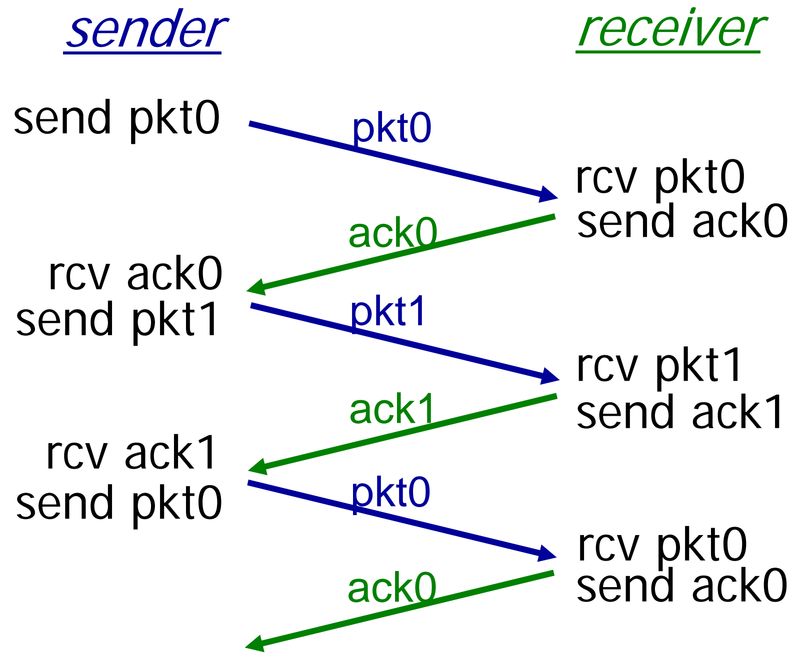
handling duplicates:

- sender adds *sequence number* to each pkt
- receiver discards (doesn't deliver up) duplicate pkt

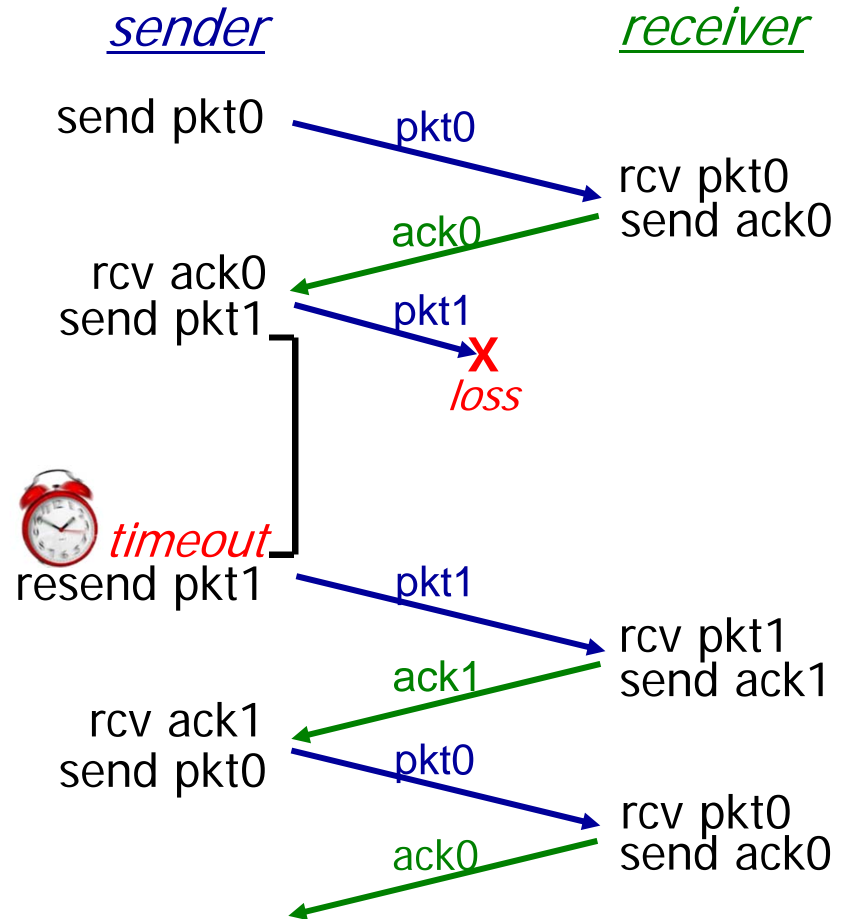
rdt3.0 sender



rdt3.0 in action

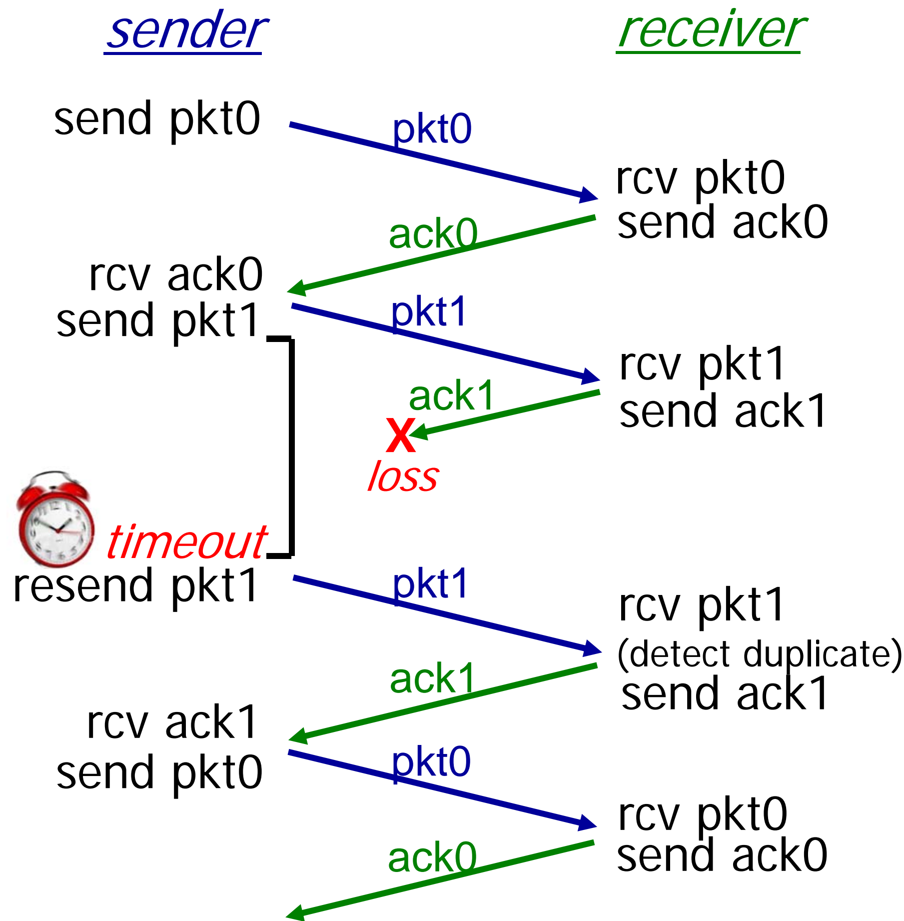


(a) no loss

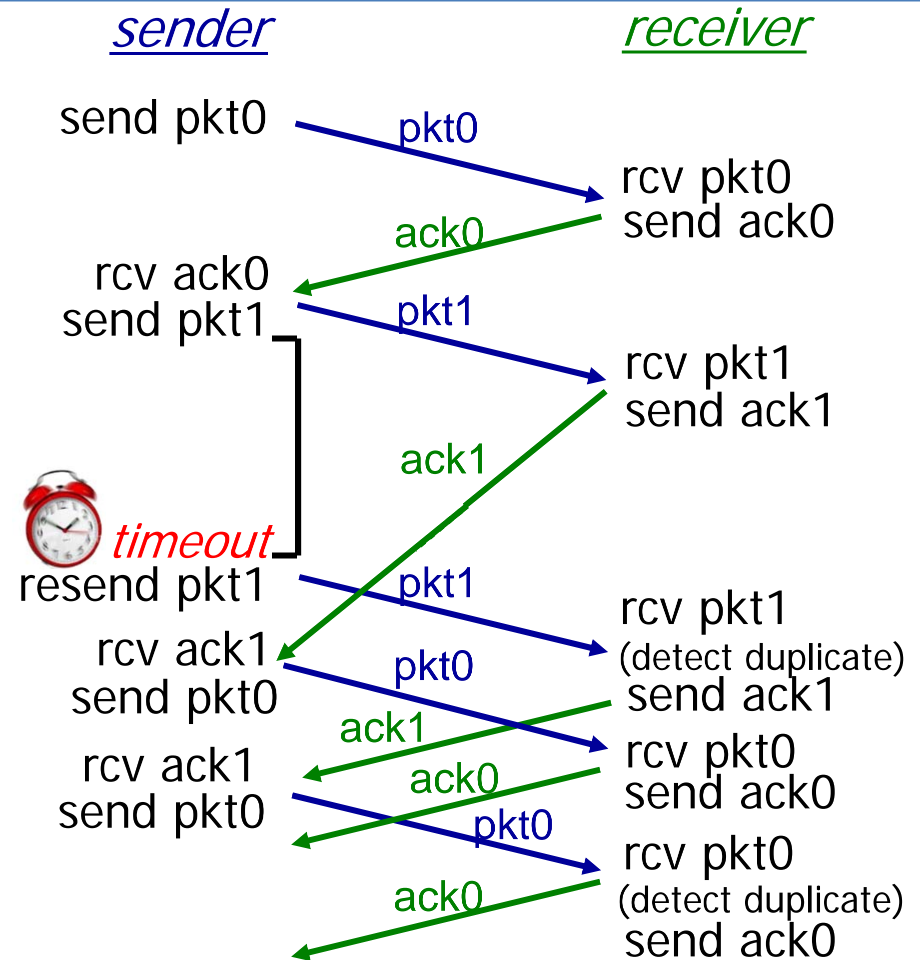


(b) packet loss

rdt3.0 in action



(c) ACK loss



(d) premature timeout/ delayed ACK

Performance of rdt3.0

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

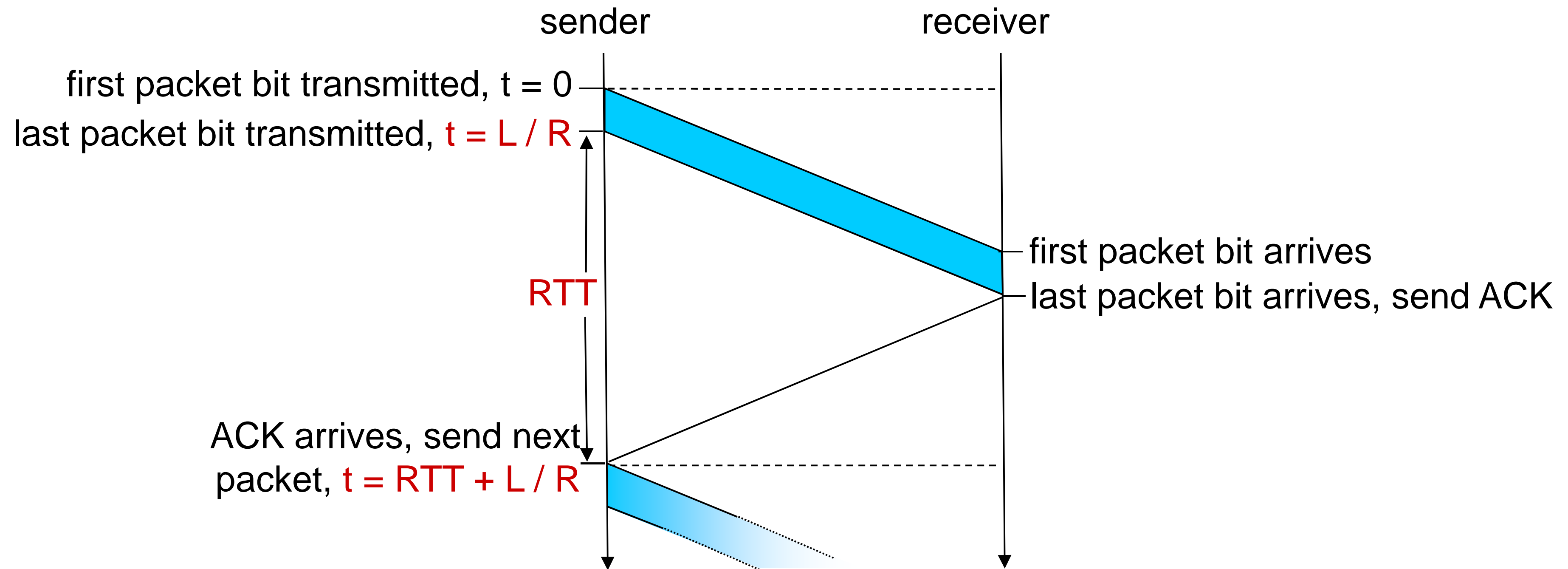
$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

- *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:
  → we get 330 kbps throughput over a 1 Gbps link
- network protocol limits use of physical resources!

rdt3.0: stop-and-wait operation



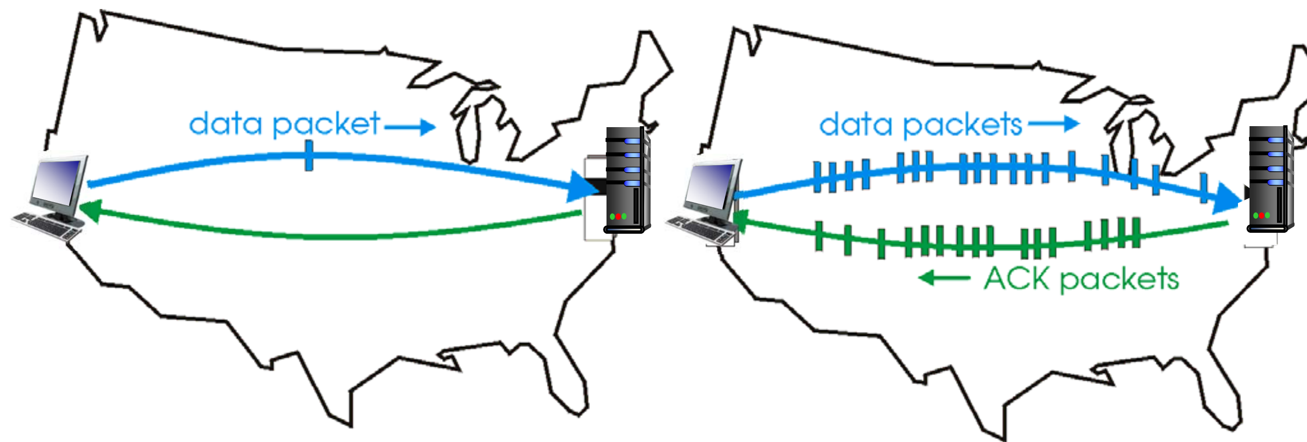
$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

Is RDT necessarily that slow/inefficient?

Pipelined protocols

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

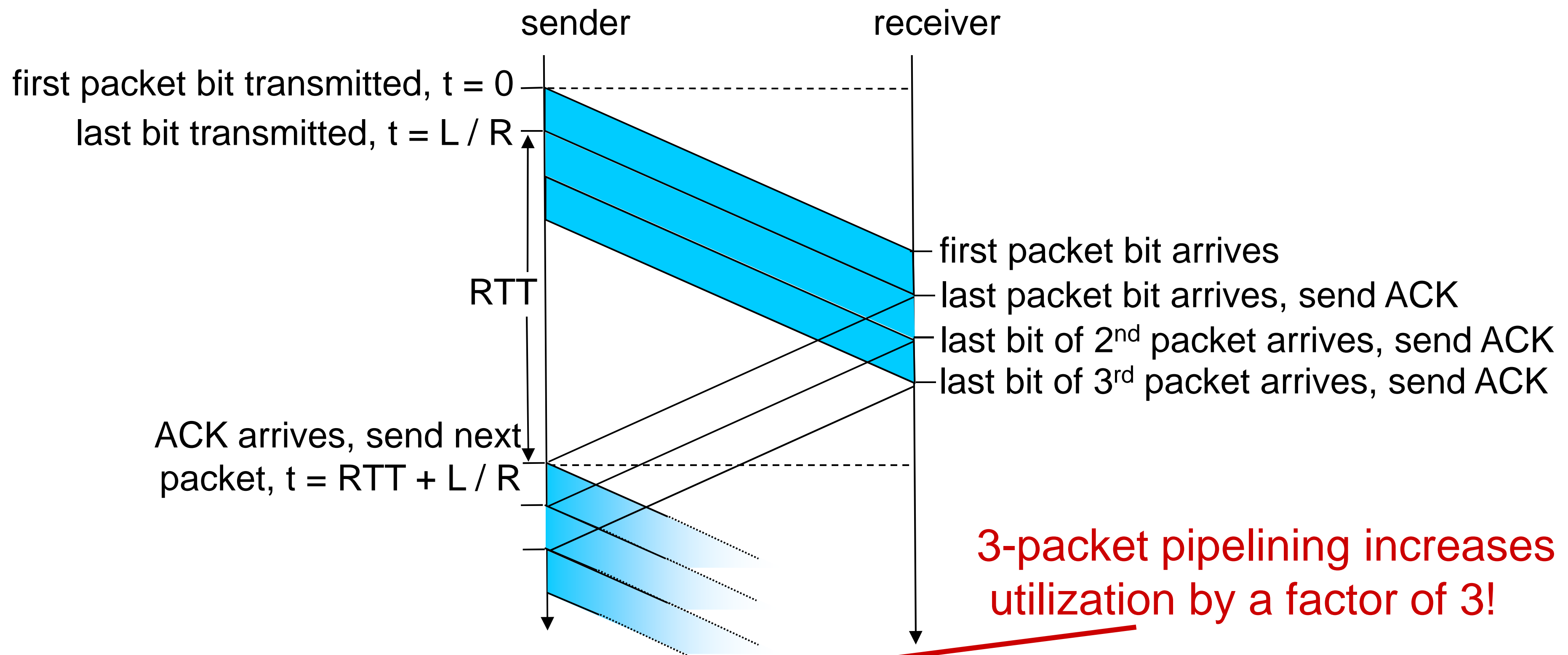
- range of sequence numbers must be increased
- buffering at sender and/or receiver



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

(b) a pipelined protocol in operation

Pipelining: increased utilization



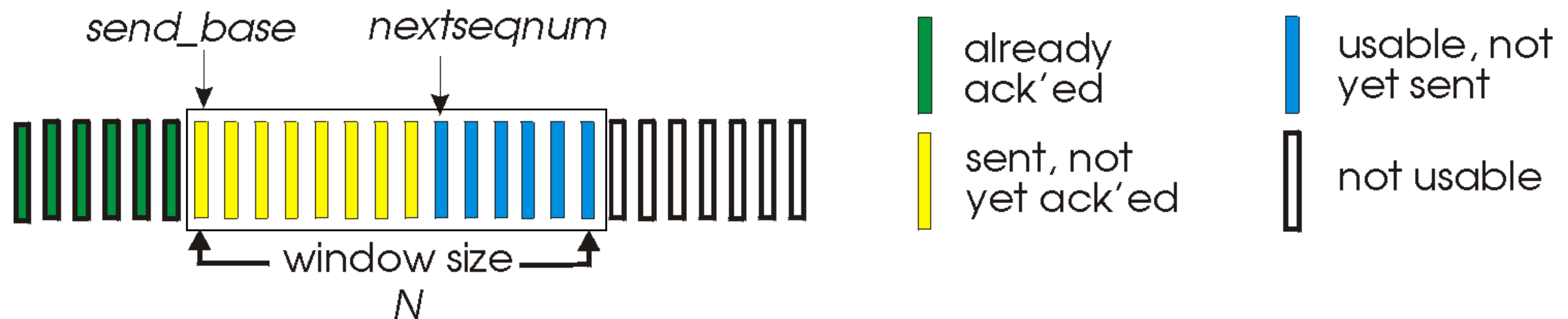
$$U_{sender} = \frac{3L / R}{RTT + L / R} = \frac{.0024}{30.008} = 0.00081$$

Pipelined protocols: ack-based error control

if data is lost, two generic forms of pipelined protocols: *go-Back-N, selective repeat*

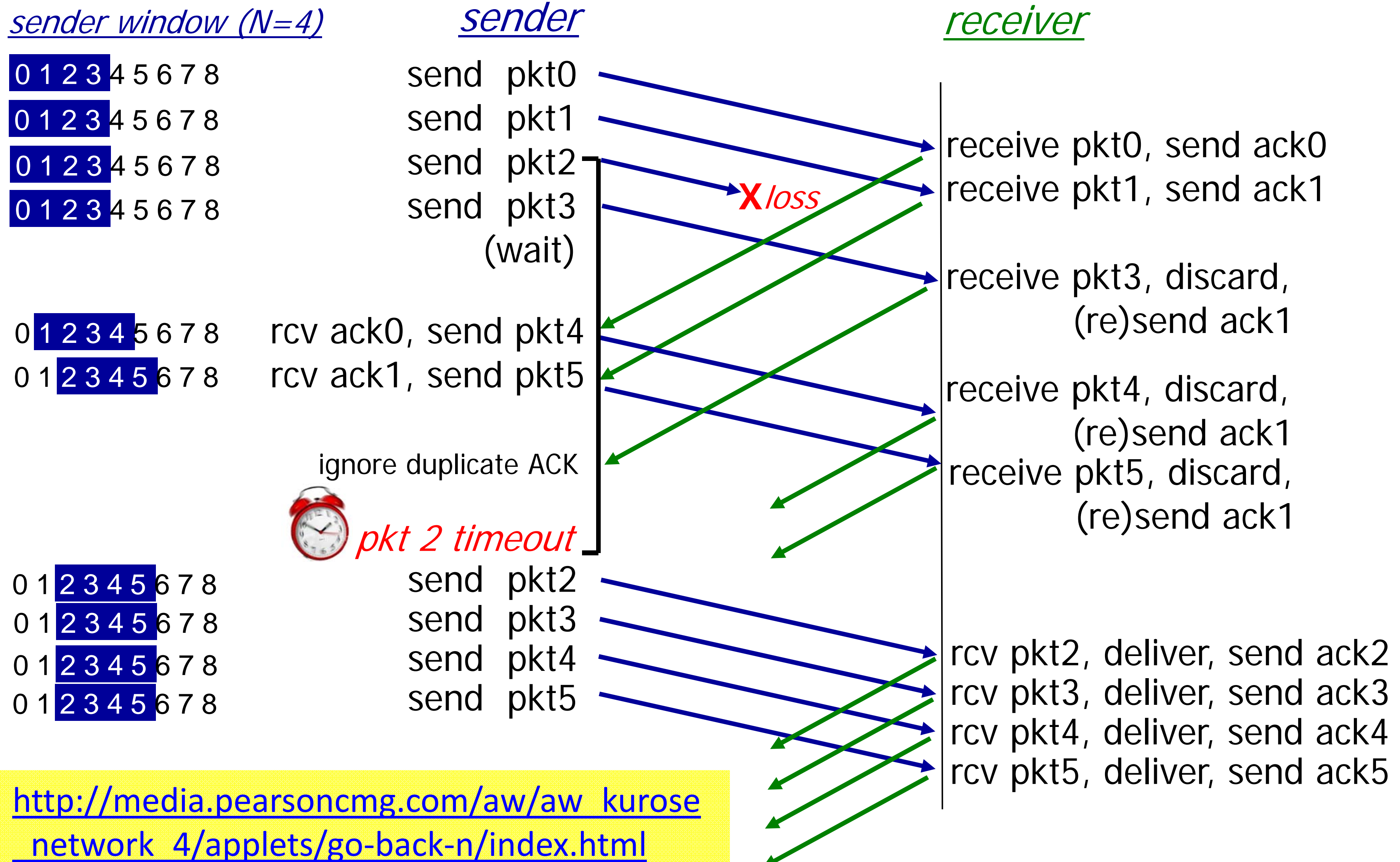
Go-Back-n: sender

- "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 packet n and all higher seq # pkts in window

GBn in action

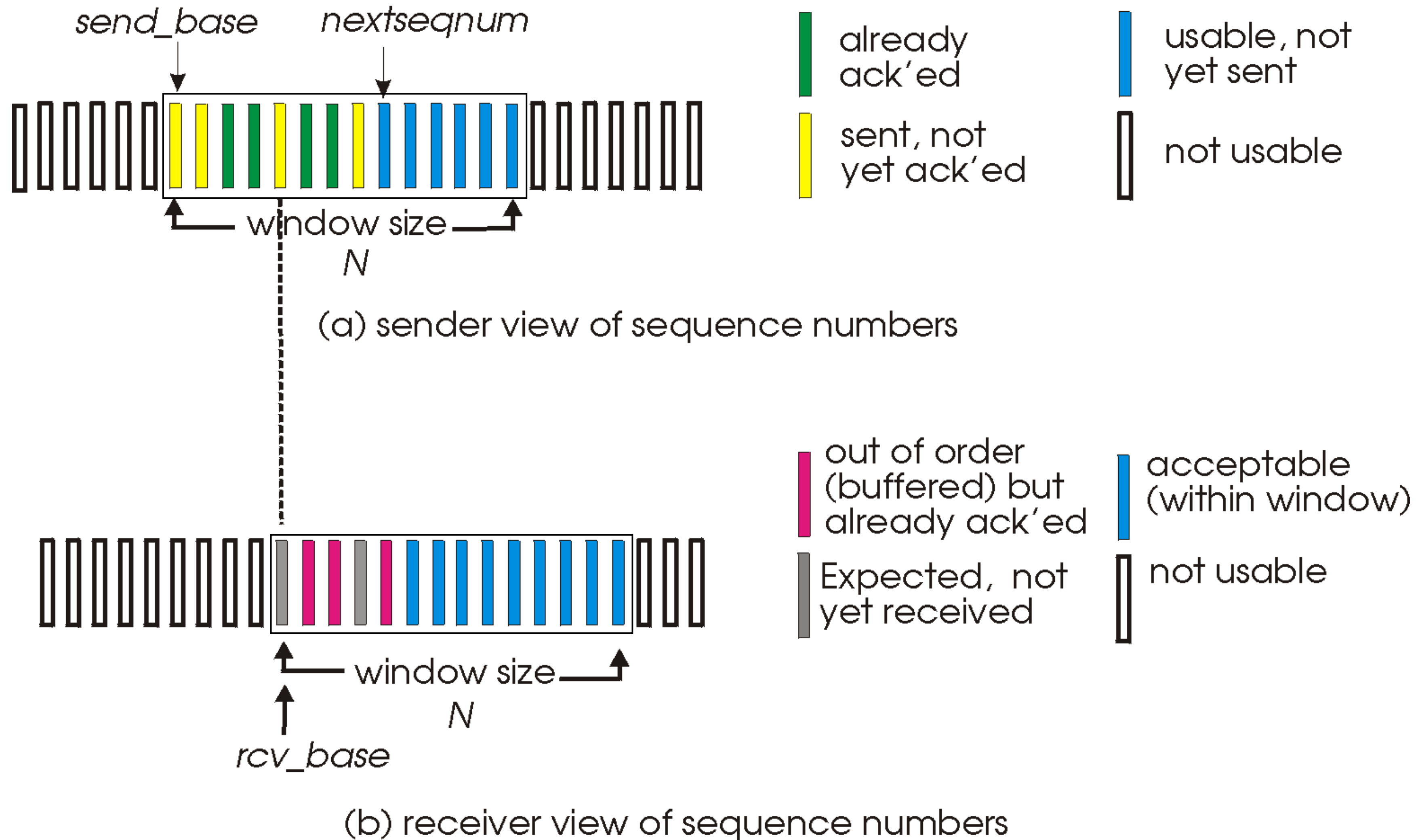


http://media.pearsoncmg.com/aw/aw_kurose_network_4/applets/go-back-n/index.html

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
  - limits seq #s of sent, unACKed 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

sender window (N=4)	sender	receiver
0 1 2 3 4 5 6 7 8	send pkt0	receive pkt0, send ack0
0 1 2 3 4 5 6 7 8	send pkt1	receive pkt1, send ack1
0 1 2 3 4 5 6 7 8	send pkt2 (X loss)	
0 1 2 3 4 5 6 7 8	send pkt3 (wait)	receive pkt3, buffer, send ack3
0 1 2 3 4 5 6 7 8	rcv ack0, send pkt4	receive pkt4, buffer, send ack4
0 1 2 3 4 5 6 7 8	rcv ack1, send pkt5	receive pkt5, buffer, send ack5
0 1 2 3 4 5 6 7 8	record ack3 arrived	
	pkt 2 timeout	
	send pkt2	rcv pkt2; deliver pkt2, pkt3, pkt4, pkt5; send ack2
0 1 2 3 4 5 6 7 8	record ack4 arrived	
0 1 2 3 4 5 6 7 8	record ack5 arrived	
0 1 2 3 4 5 6 7 8		

Q: what happens when ack2 arrives?

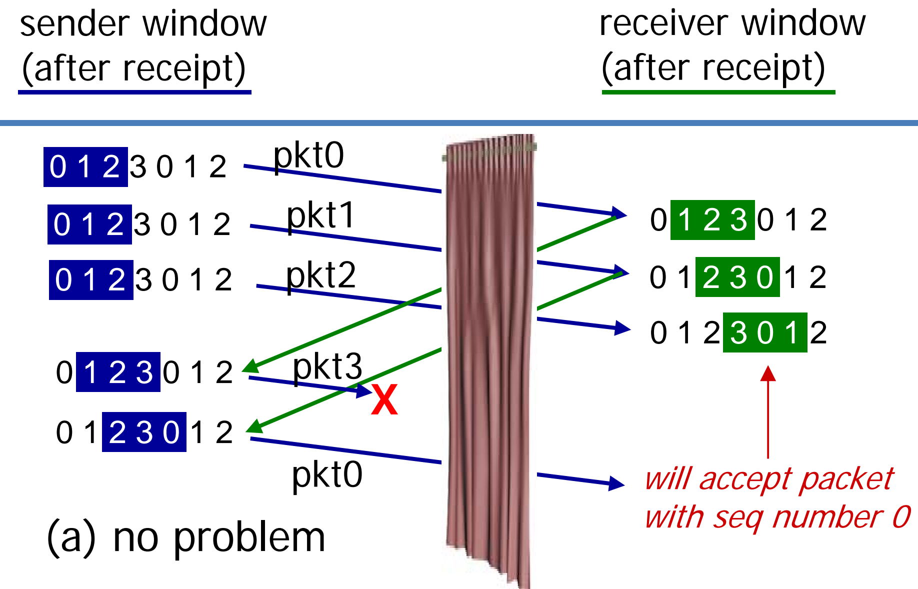
Sequence numbers

example:

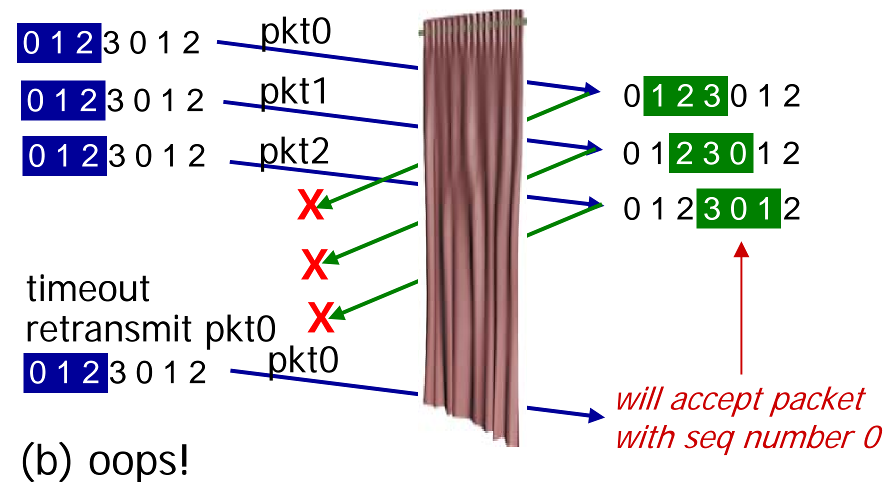
- seq #'s: 0, 1, 2, 3
- window size=3

- receiver sees no difference in two scenarios!
- duplicate data accepted as new in (b)

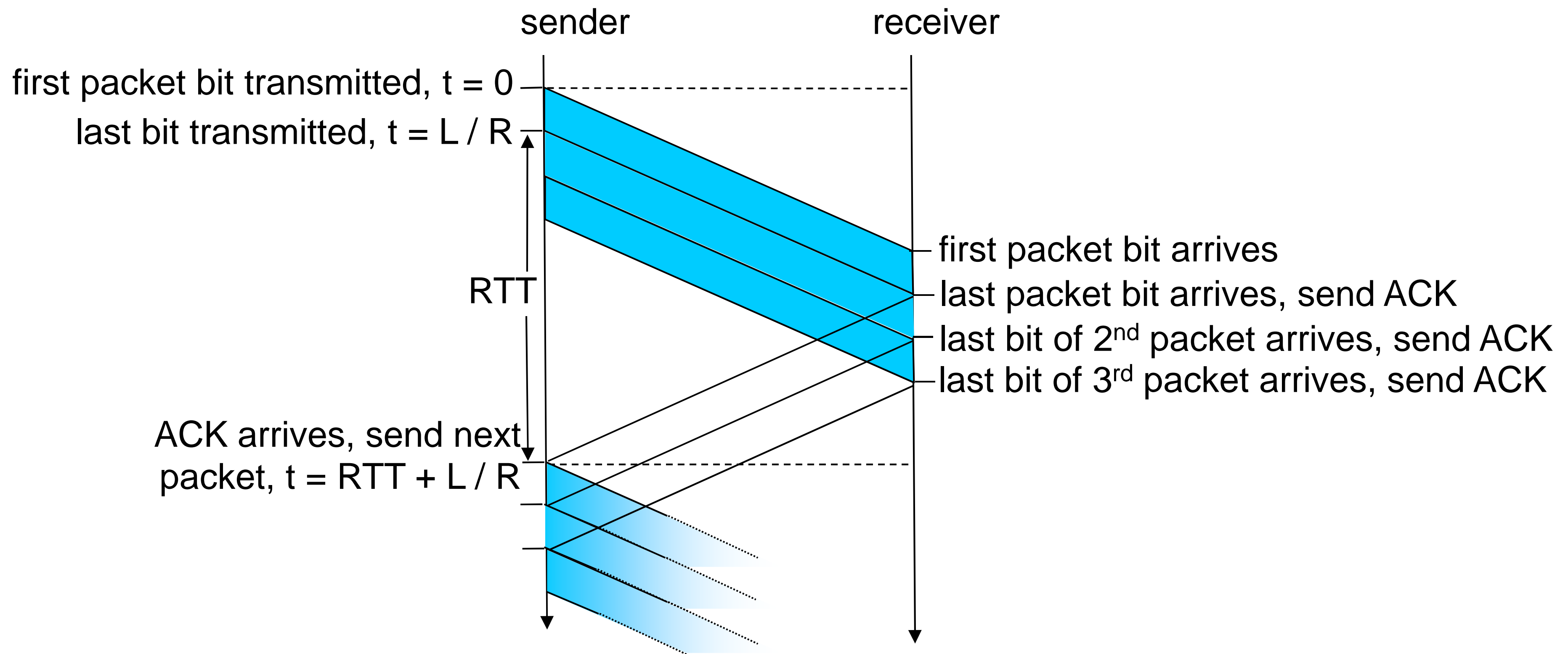
Q: what relationship between seq # size and window size to avoid problem in (b)?



receiver can't see sender side.
receiver behavior identical in both cases!
something's (very) wrong!



Pipelining: increased utilization
Ack-based => flow control at the same time!!!



Flow control: Sender/receiver problem; S cares to not overwhelm R

Roadmap

- Transport layer services
- Addressing, multiplexing/demultiplexing
- Connectionless, unreliable transport: UDP
- principles of reliable data transfer

- *Next lecture: connection-oriented transport: TCP*
  - *reliable transfer*
  - *flow control*
  - *connection management*
  - *TCP congestion control*

Reading instructions chapter 3

- **KuroseRoss book**

Careful	Quick
3.1, 3.2, 3.4-3.7	3.3

- **Other resources (further, optional study)**

  - Lakshman, T. V., Upamanyu Madhow, and Bernhard Suter. "Window-based error recovery and flow control with a slow acknowledgement channel: a study of TCP/IP performance." INFOCOM'97. Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE. Vol. 3. IEEE, 1997.
  - Rizzo, Luigi. "Effective erasure codes for reliable computer communication protocols." ACM SIGCOMM Computer Communication Review 27.2 (1997): 24-36.
  - A. Agarwal and M. Charikar, “On the advantage of network coding for improving network throughput,” in Proceedings of the IEEE Information Theory Workshop, Oct. 2004
  - Harvey, N. J., Kleinberg, R., & Lehman, A. R. (2006). On the capacity of information networks. IEEE/ACM Transactions on Networking (TON), 14(SI), 2345-2364.

Some review questions on this part

- Why do we need an extra protocol, i.e. UDP, to deliver the datagram service of Internets IP to the applications?
- Draw space-time diagrams without errors and with errors, for the following, for a pair of sender-receive S-R: (assume only 1 link between them)
  - Stop-and-wait: transmission delay < propagation delay and transmission delay > propagation delay
  - Sliding window aka pipeleined protocol, with window's transmission delay < propagation delay and window's transmission delay > propagation delay; illustrate both go-back-n and selective repeat when there are errors
  - Show how to compute the effective throughput between S-R in the above cases, whene there are no errors

Review questions cont.

- What are the goals of reliable data transfer?
- Reliable data transfer: show why we need sequence numbers when the sender may retransmit due to timeouts.
- Show how there can be wraparound in a reliable data transfer session if the sequence-numbers range is not large enough.
- Describe the go-back-N and selective repeat methods for reliable data transfer

Extra slides, for further study

Bounding sequence numbers for stop-and-wait…

… s.t. no wraparound, i.e. we do not run out of numbers: *binary value suffices for stop-and-wait:*

Prf: assume towards a contradiction that there is wraparound when we use binary seq. nums.

- R expects segment #f, receives segment #(f+2):

  R rec. f+2 => S sent f+2 => S rec. ack for f+1

  => R ack f+1=> R ack f => contradiction

- R expects f+2, receives f:

  R exp. f+2 => R ack f+1 => S sent f+1

  => S rec. ack for f => contradiction