Course on Computer Communication and Networks

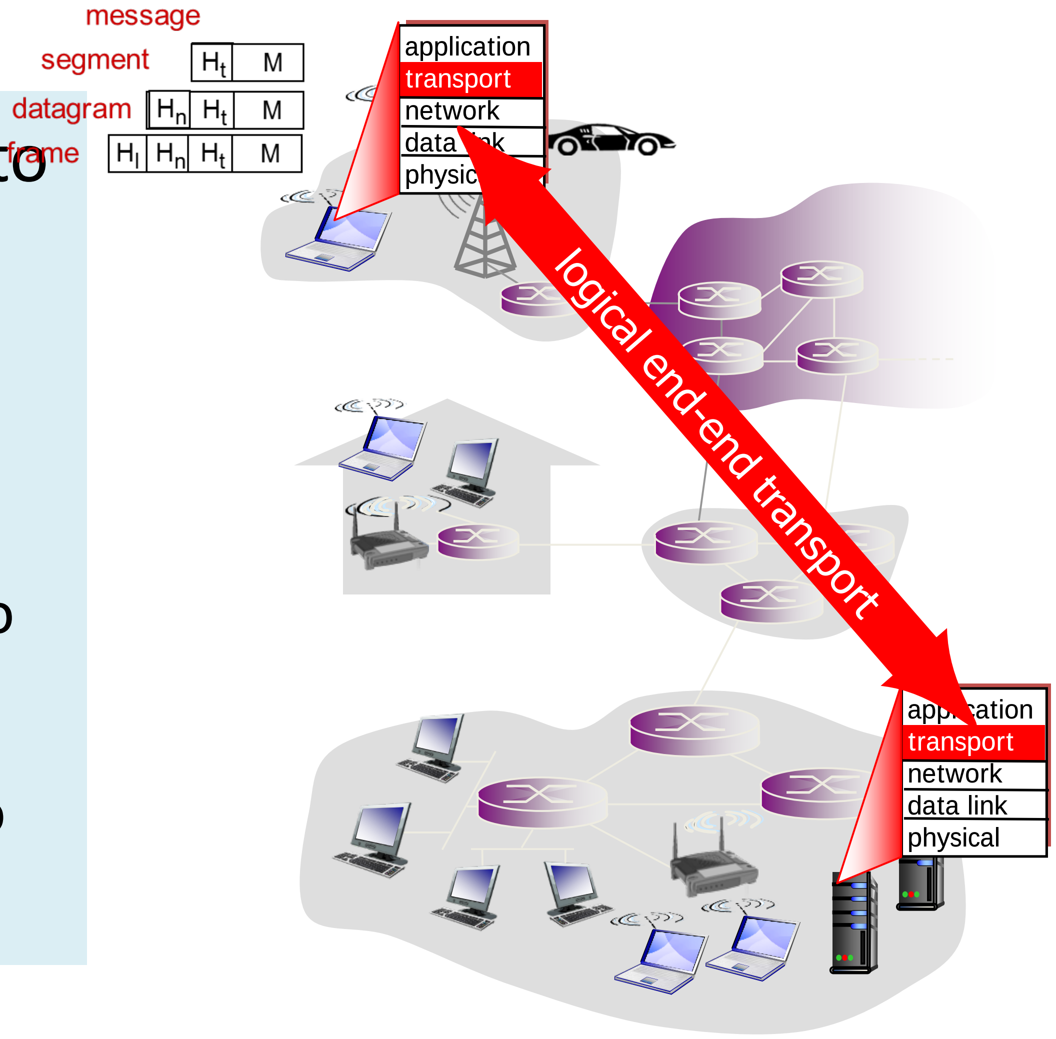
Lecture 4
Chapter 3; Transport Layer, Part A

EDA344/DIT 423, CTH/GU

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

Transport services and protocols

- provide communication services to app-layer protocols
- 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



Parenthesis: On last week's questions

Q: Types of services that a transport layer may need to provide.

- Which of those are provided by in the Internet transport layer protocols?

Services i.e. properties

- No-loss
- In-order delivery
- Timeliness i.e. latency, bandwidth guarantees

Internet transport-layer protocols

Reliable, in-order delivery: **TCP**

- also provides
  - connection setup
  - flow control
  - + care for the health of the network (aka TCP's congestion control)

Best effort (can be unreliable, unordered) delivery: **UDP**

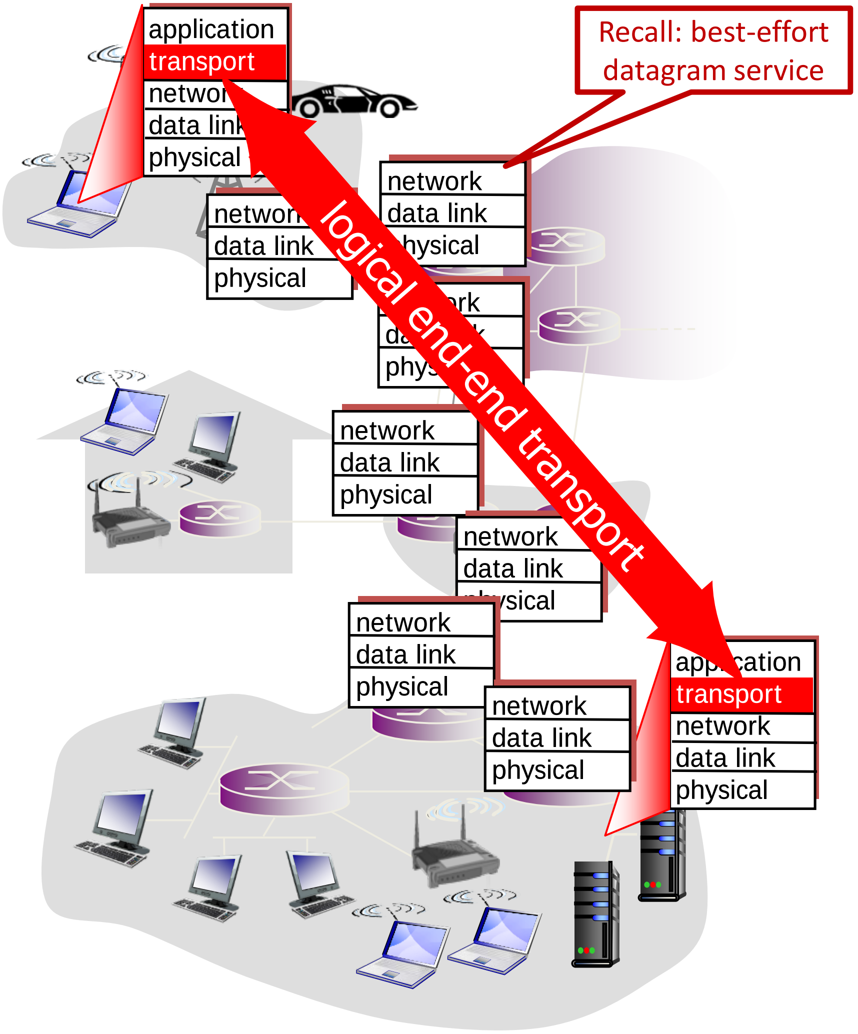
- no-frills extension of "best-effort" IP

Both support addressing (encapsulation), of course!

Transport Layer services **not available** in the Internet:

Delay/bandwidth guarantees. Why?

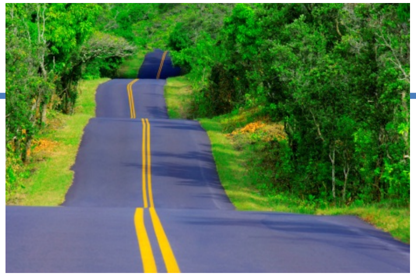
*When the (**successful due to simplicity**) TCP/IP protocol stack was defined, no foreseeable need for such applications in an inter-net.*



Roadmap

Transport Layer: Learning goals:

- understand principles of transport layer services:
  - addressing, multiplexing/demultiplexing
  - reliable data transfer
  - flow control
  - congestion control (not really a Transport-layer issue – *some study in connection to transport layer since it is there in TCP; more in connection with RealTime traffic*)
- instantiation and implementation in the Internet



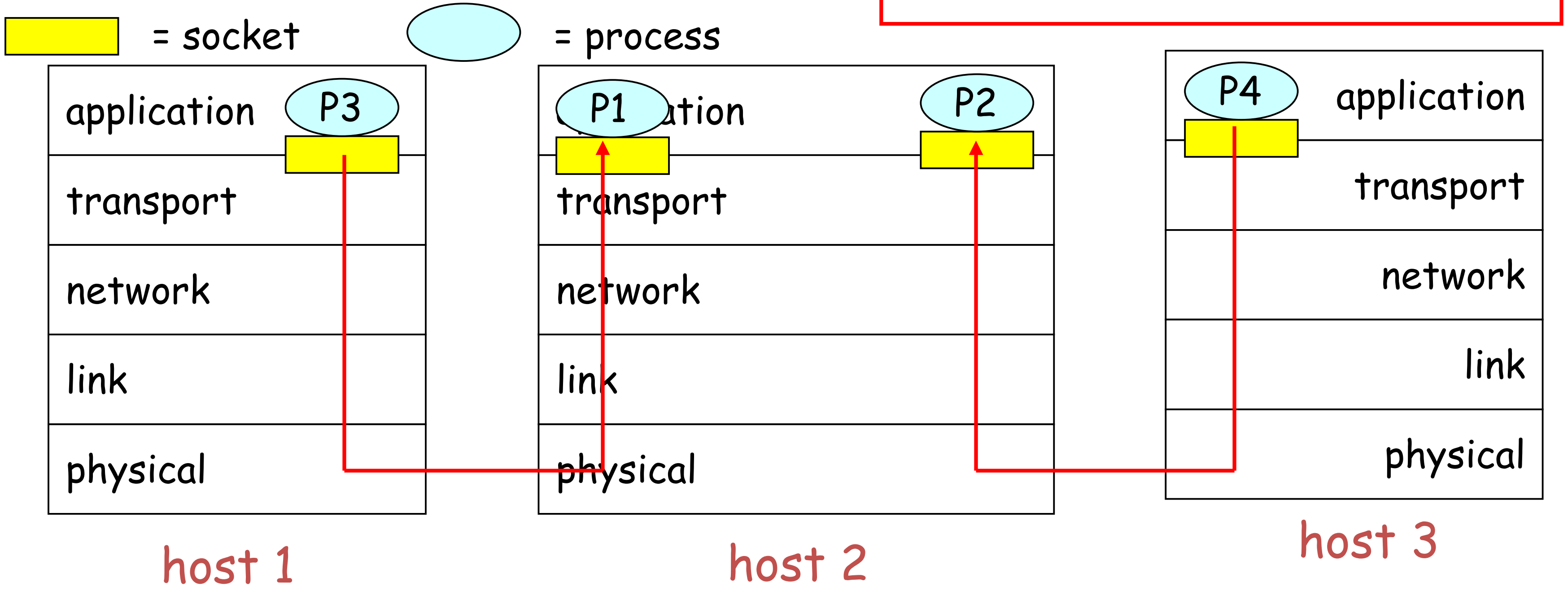
- Transport layer services in Internet
- Addressing, multiplexing/demultiplexing
- Connectionless, unreliable transport: UDP
- Principles of reliable data transfer

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

Addressing: Multiplexing/demultiplexing (+ recall encapsulation)

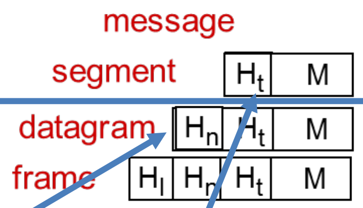
Demultiplexing at rcv host:
delivering received segments to correct socket

Multiplexing at send host:
gathering data, enveloping data with header (later used for demultiplexing)

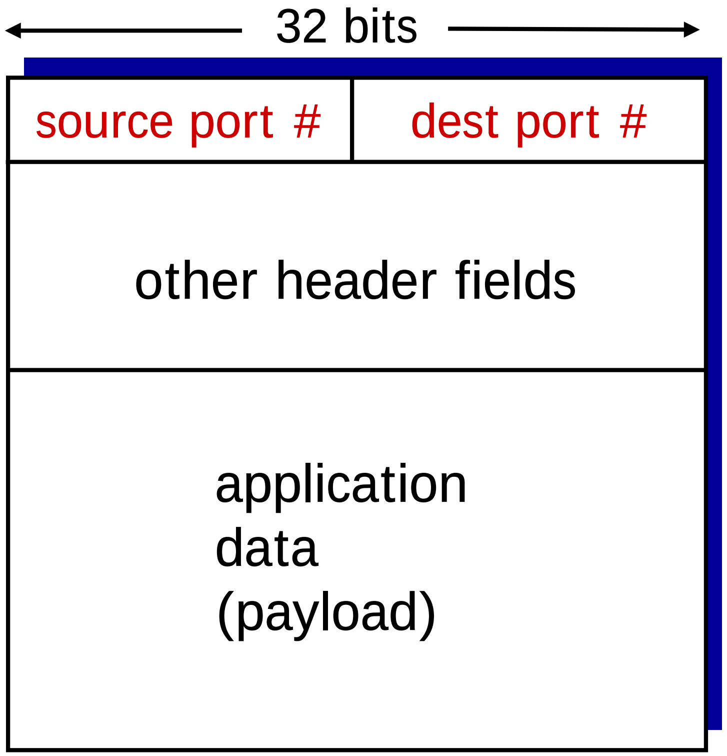


Recall: *segment* - unit of data exchanged between transport layer entities
aka TPDU: transport protocol data unit

Addressing



- Host receives IP datagrams
  - Datagram (i.e. IP packet) 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 + example

when host receives UDP segment:

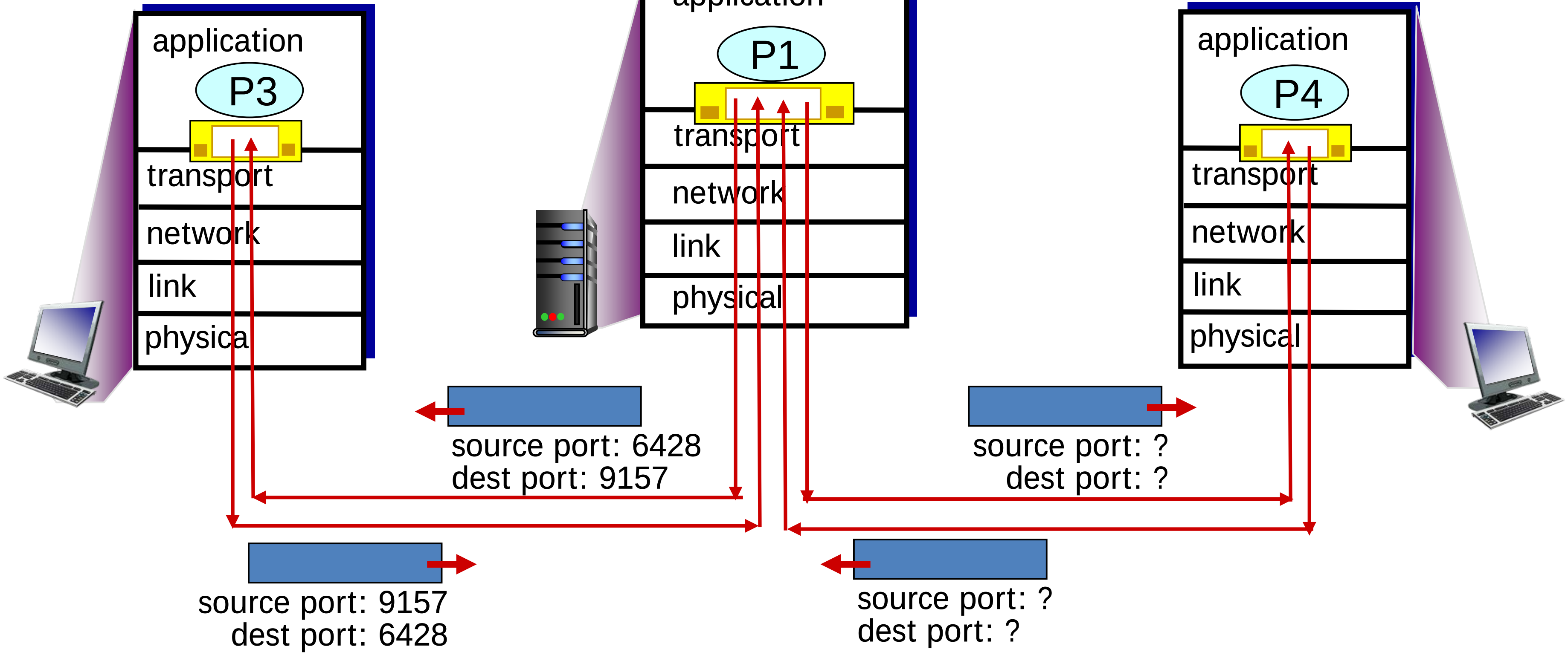
- directs UDP segment to socket with that port #

IP datagrams with *same dest. port #* (but perhaps different source IP addresses or source port numbers will be directed) *to the same socket*

```
DatagramSocket
 mySocket2 = new
 DatagramSocket
 (9157);
```

```
DatagramSocket
  serverSocket = new
  DatagramSocket
 (6428);
```

```
DatagramSocket
 mySocket1 = new
 DatagramSocket
 (5775);
```



TCPConnection-oriented (TCP) addressing/demux + example

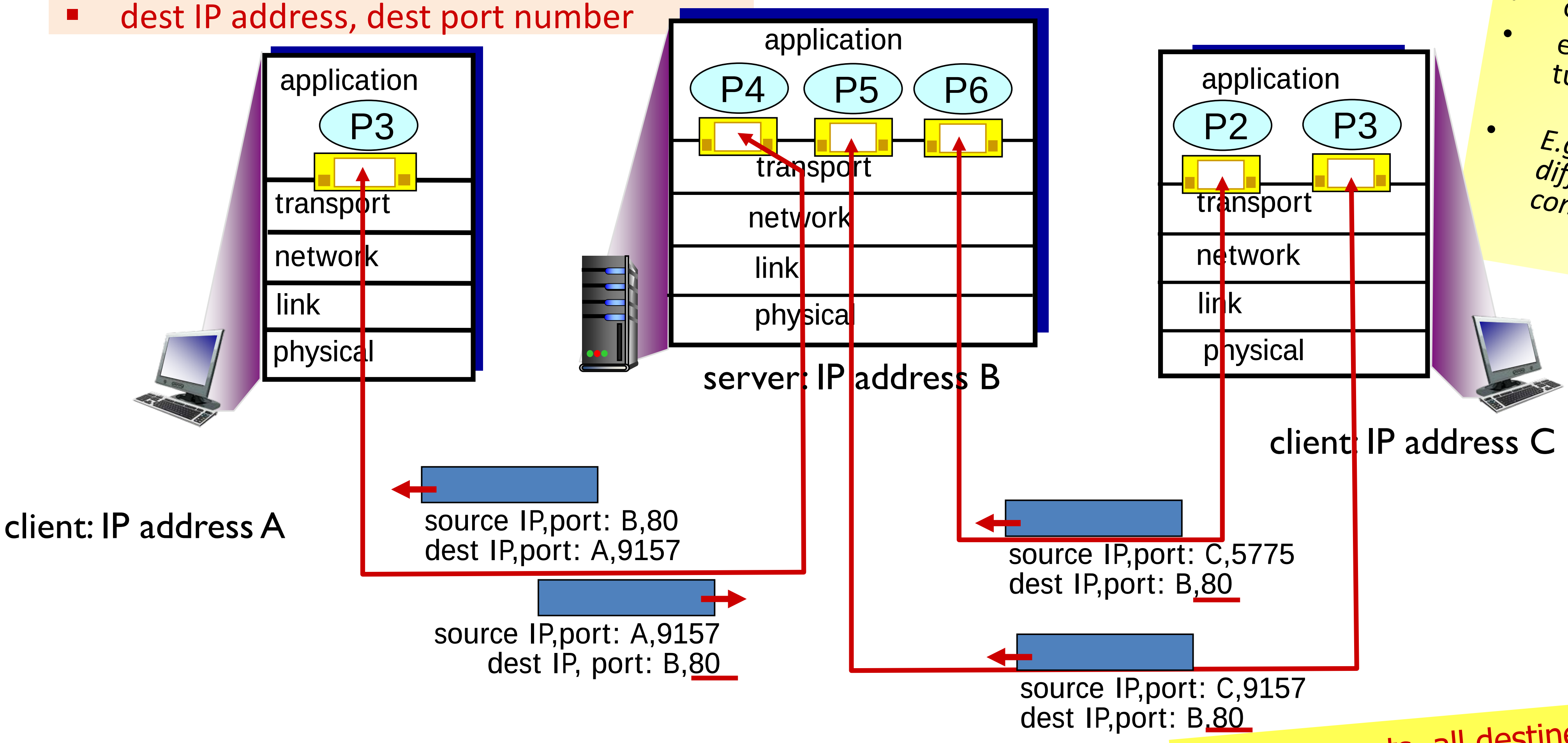
TCP socket identified by 4-tuple:

- source IP address, source port number
- dest IP address, dest port number

(Demux) Receiver uses all 4 values to direct segment to appropriate socket

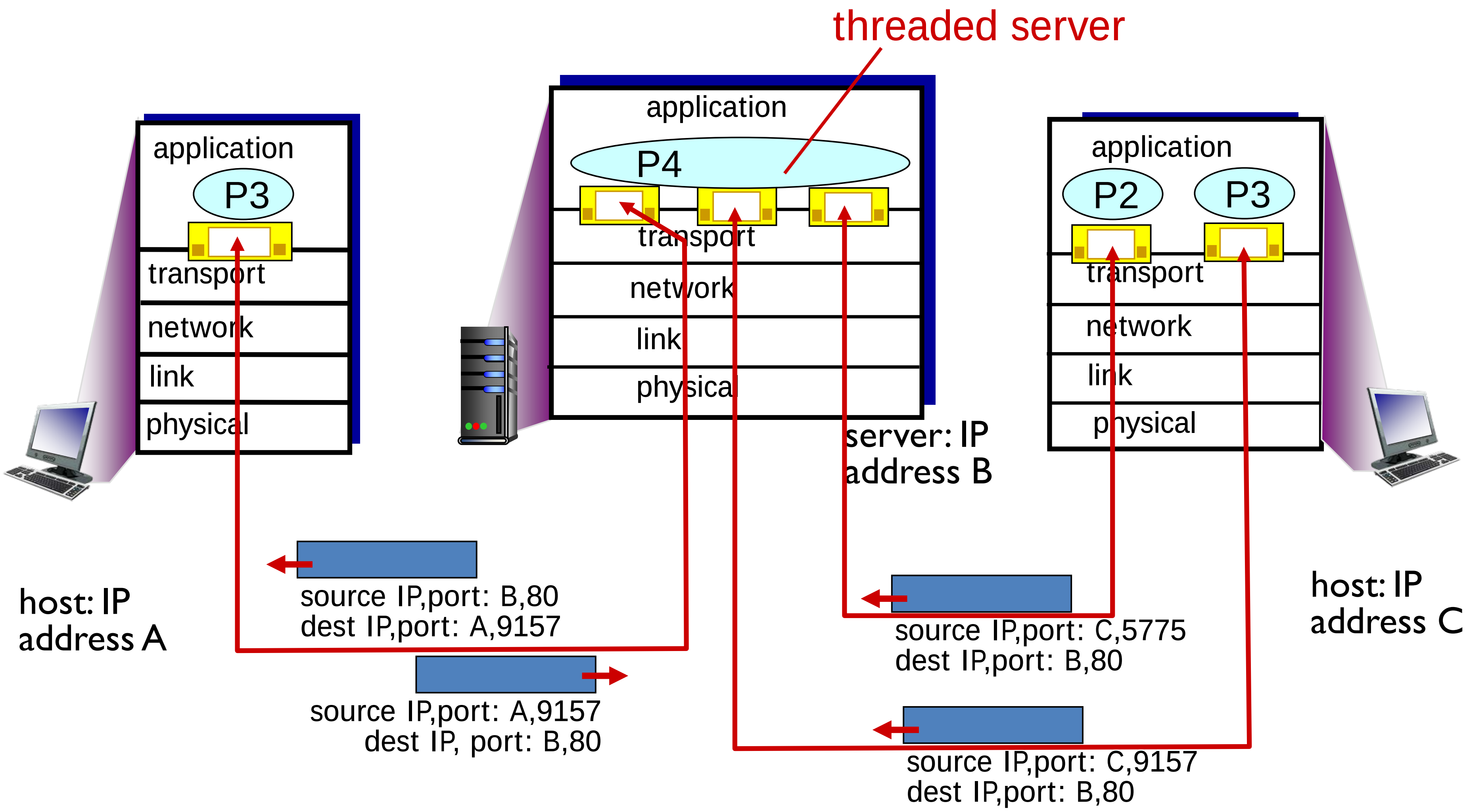
server may support many simultaneous TCP sockets:

- one socket per connection
- each identified by its own 4-tuple
- E.g. web servers have different sockets for each connecting client

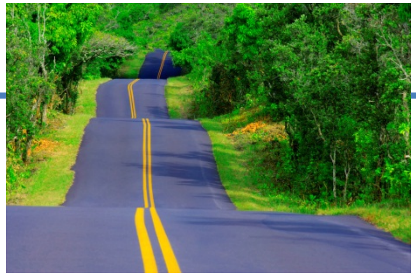


three segments, all destined to IP address: B, dest port: 80 are demultiplexed to *different* sockets

TCP demux: Threaded web server



Roadmap



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

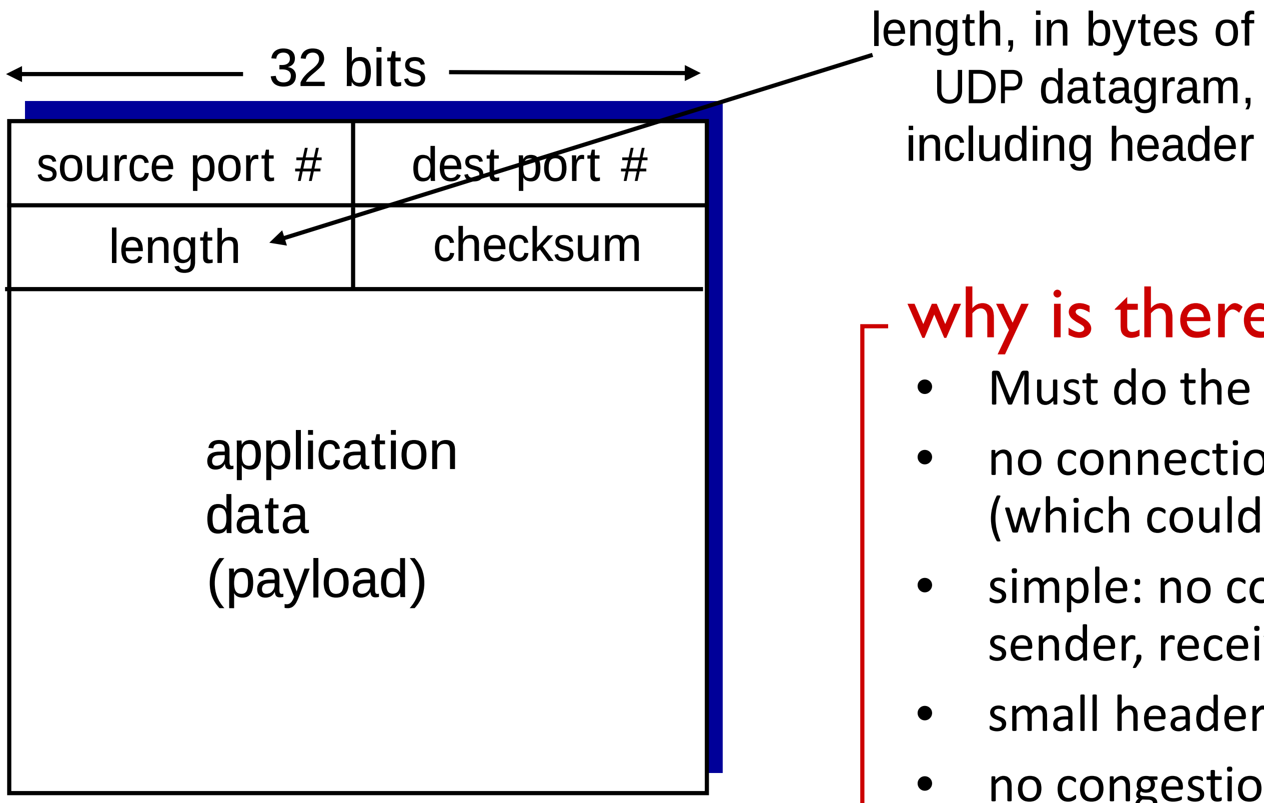
- *Next lecture: connection-oriented transport: TCP*
  - *reliable transfer*
  - *flow control*
  - *connection management*
  - *TCP congestion control*

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 datagram format

UDP used by:

- DNS
- SNMP
- More discussion later…

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 segments faster (than TCP)

UDP Checksum[RFC 1071]: check bit flips

Sender:

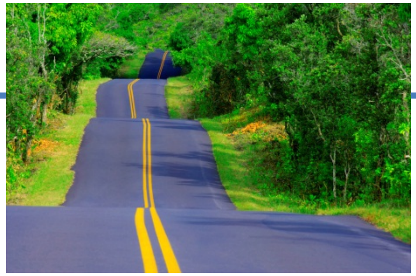
- treat segment contents as sequence of 16-bit integers
- checksum: addition (1’s complement sum) of segment contents
- sender puts checksum value into UDP checksum field

Receiver:

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

		1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0
		1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
Wraparound: Add to final	1	1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
sum		1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0
checksum		0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1

Roadmap

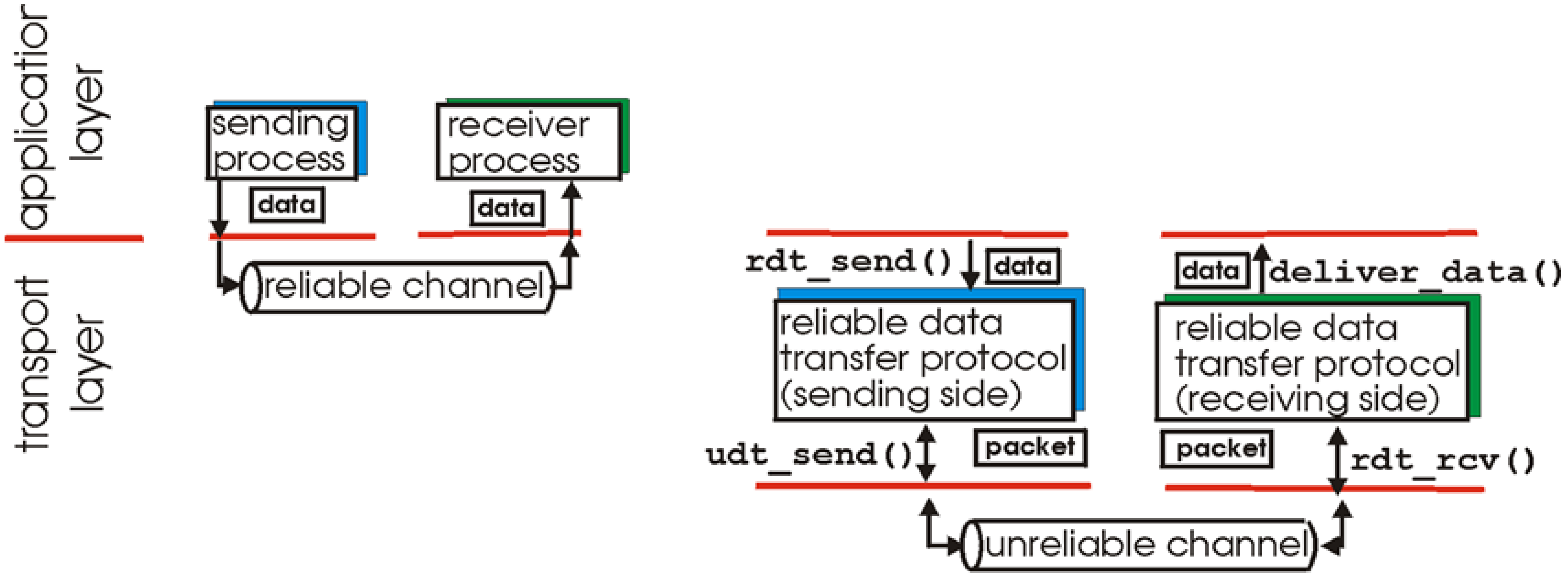


- Transport layer services
- Addressing, multiplexing/demultiplexing
- Connectionless, unreliable transport: UDP
- principles of reliable data transfer

- *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!



(a) provided service

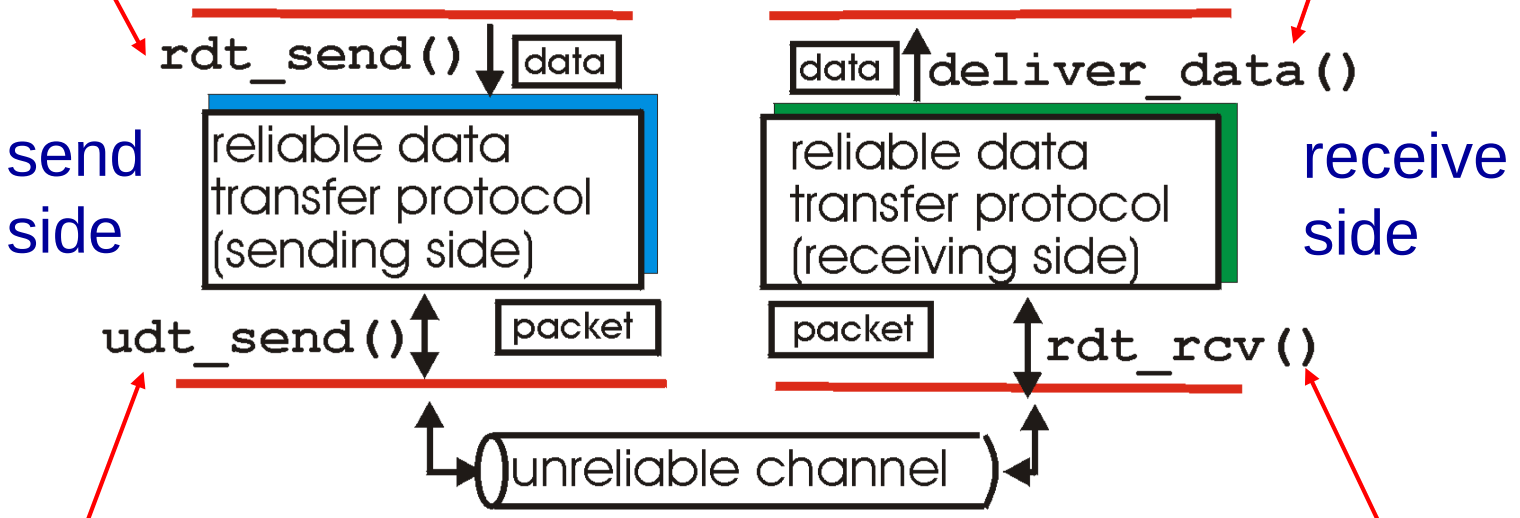
(b) service implementation

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

Reliable data transfer (RDT): 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

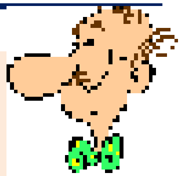
rdt_rcv(): called when packet arrives on rcv-side of channel

RDT:

S application: Author
Writes and Sends pages

R application: publisher
Receives & publishes written pages

S transport: secretary Bob
Receives pages;
Must pass on to publisher in-order



a-time-MMS connection

ne;

ob if MMS connection…

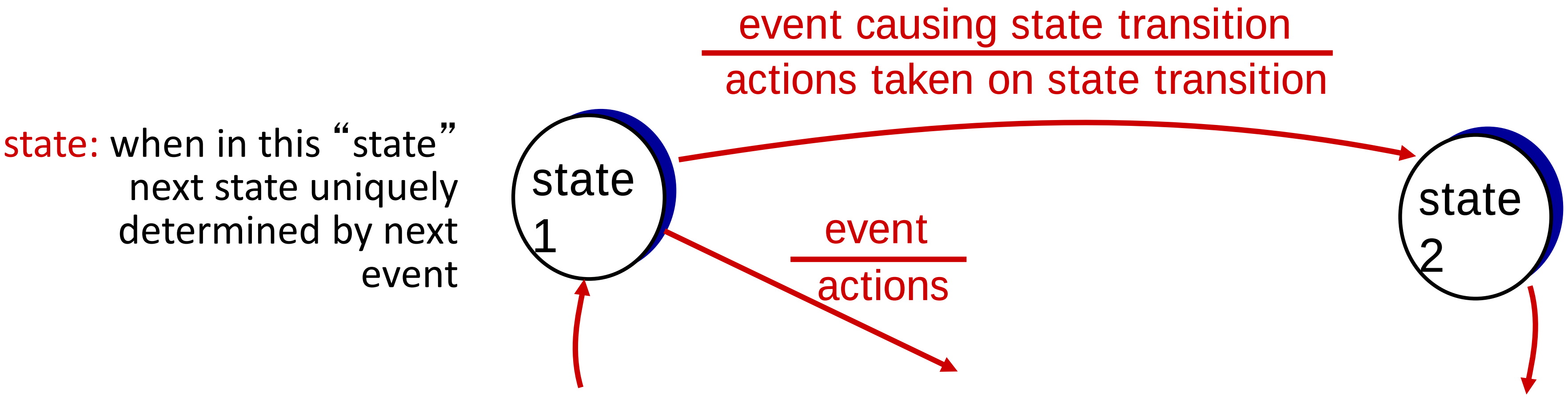
rs?

- …might lose MMSs?

Reliable data transfer: getting started

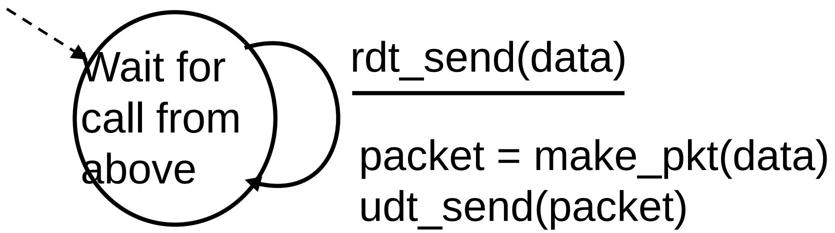
We will:

- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- use **finite state machines (FSM)** to specify sender, receiver behaviour

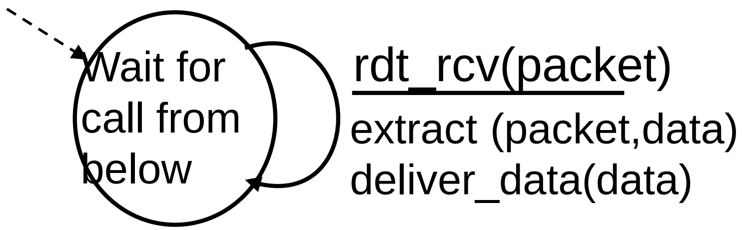


rdt1.0: reliable transfer & reliable channel

- underlying channel perfectly reliable
  - no bit errors, no loss of packets
- separate FSMs for sender, receiver:



sender



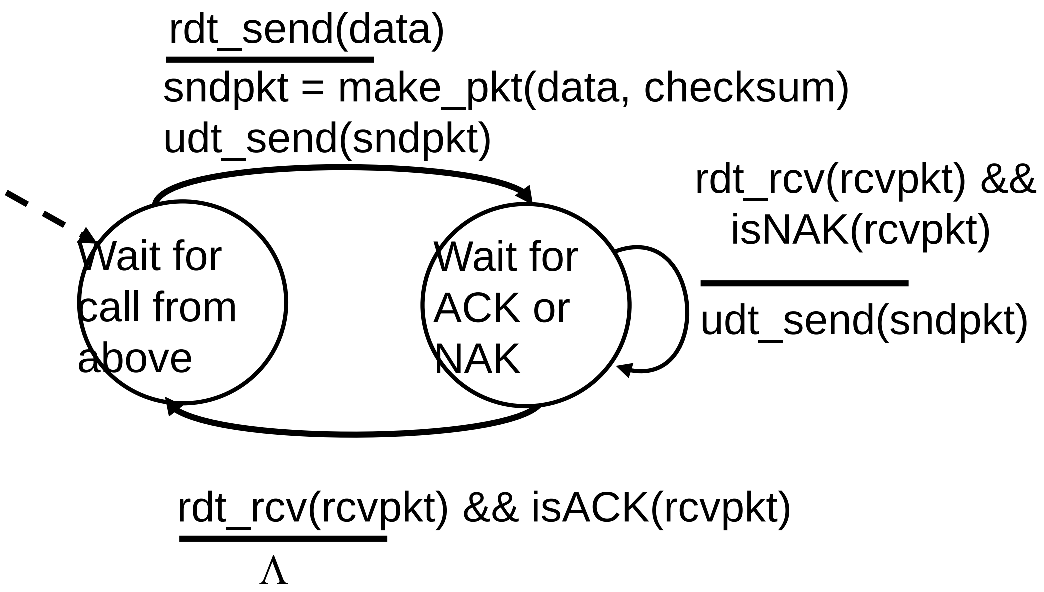
receiver

rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - checksum to detect bit errors
- 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

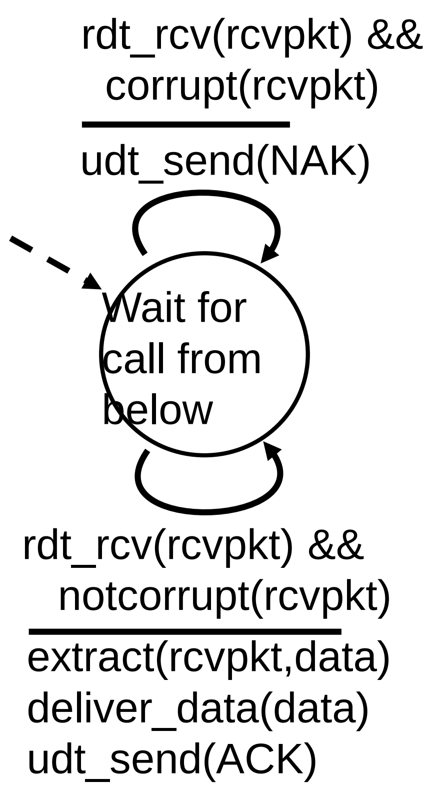
New mechanisms in `rdt2.0` (beyond `rdt1.0`):
- error detection
- feedback: control msgs (ACK,NAK) from receiver to sender

rdt2.0: FSM specification

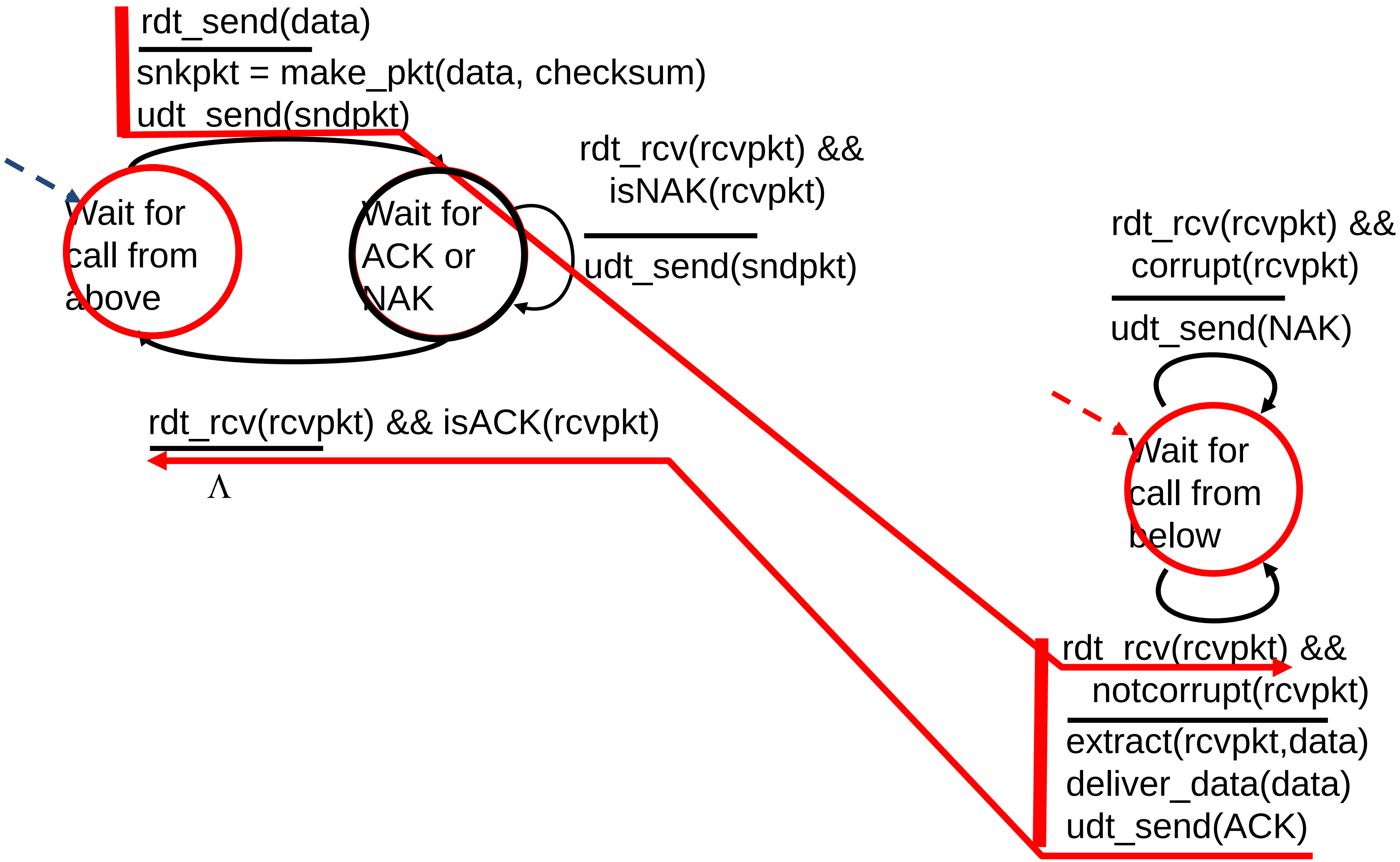


sender

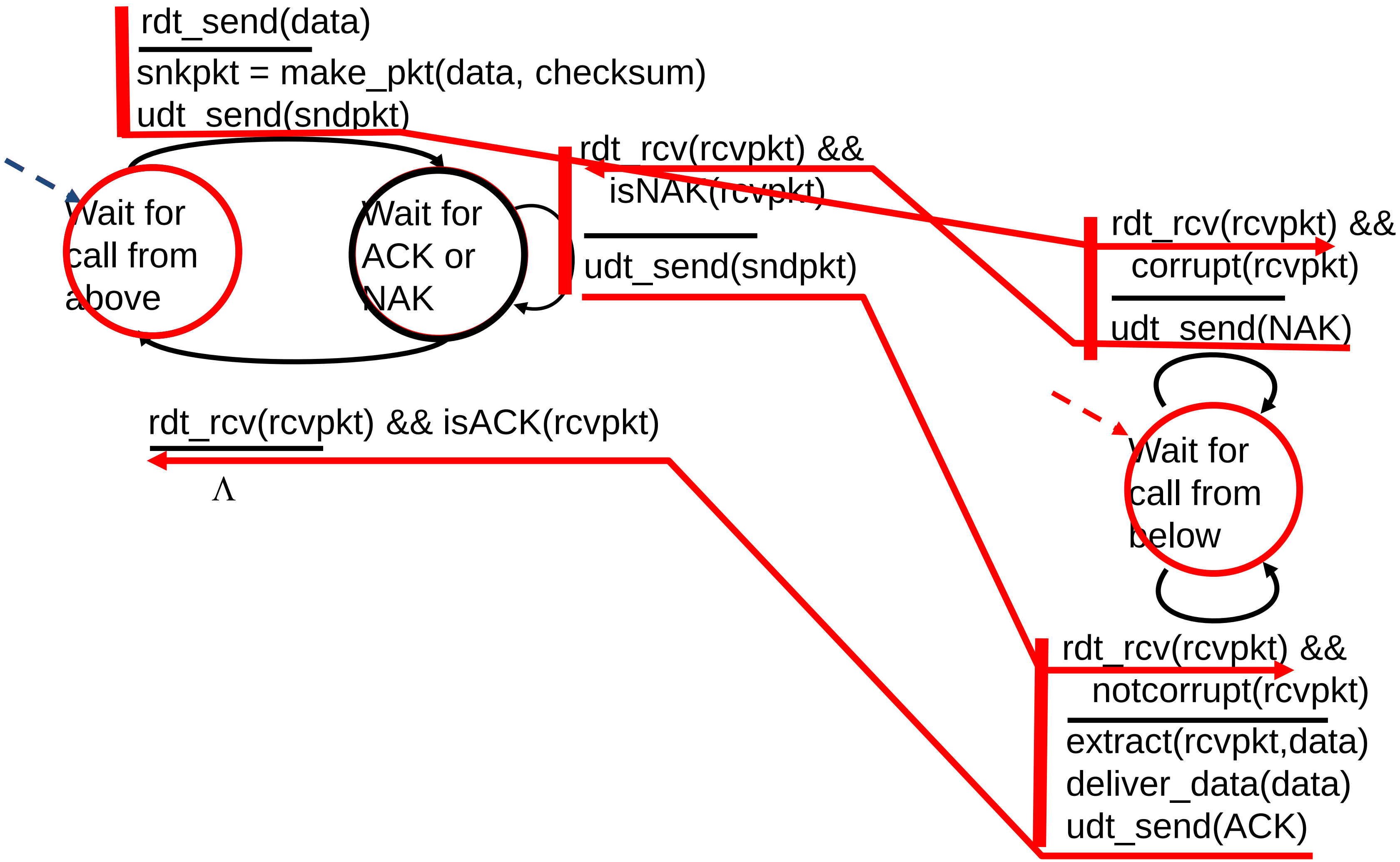
receiver



rdt2.0: operation with no errors



rdt2.0: error scenario

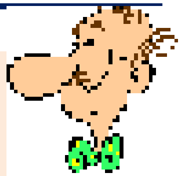


Recall: RDT
(Reliable Data Transfer, aka error control)

S application: Author
Writes and Sends pages

R application: publisher
Receives & publishes written pages

S transport: secretary Bob
Receives pages;
Must pass on to publisher in-order



a-time-MMS connection

MMS connection…

rdt3.0: channels with errors *and* loss

We saw: how ack+retransmit can solve problems with errors

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

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

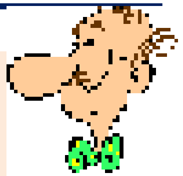
- retransmits if no ACK received in this time
  - requires countdown timer

Recall: RDT
(Reliable Data Transfer, aka error control)

S application: Author
Writes and Sends pages

R application: publisher
Receives & publishes written pages

S transport: secretary Bob
Receives pages;
Must pass on to publisher in-order



a-time-MMS connection

MMS connection…

rdt3.0 (cont) : channels with errors *and* loss

We saw: how ack+retransmit can solve problems with errors

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

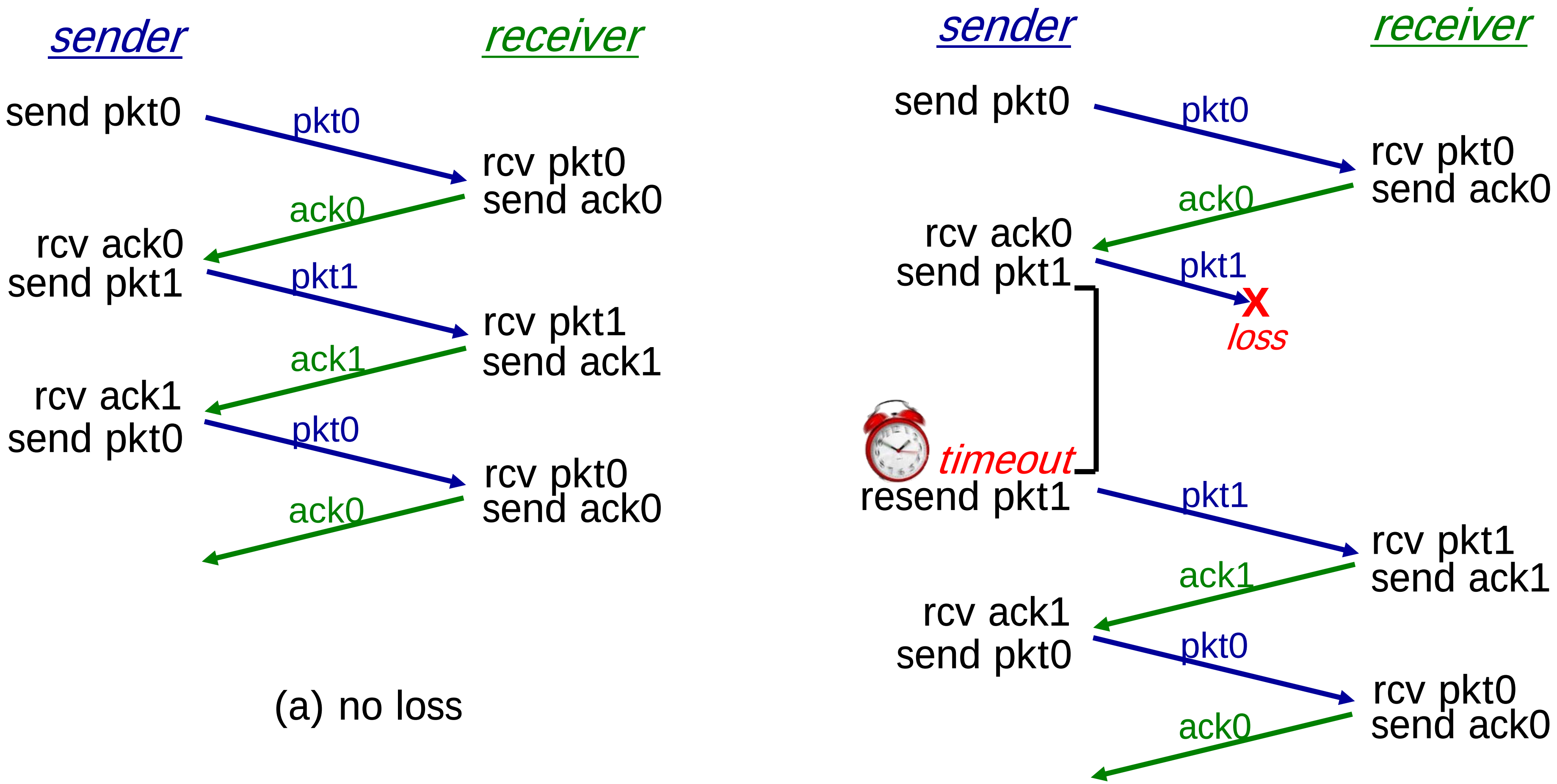
approach: sender waits "reasonabl[...]"
amount of time for ACK

- retransmits if no ACK received in this [...]
  - 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 upwards) duplicate pkt
- For stop&wait 0-1 (ie 1 bit) enough for sequence nr.

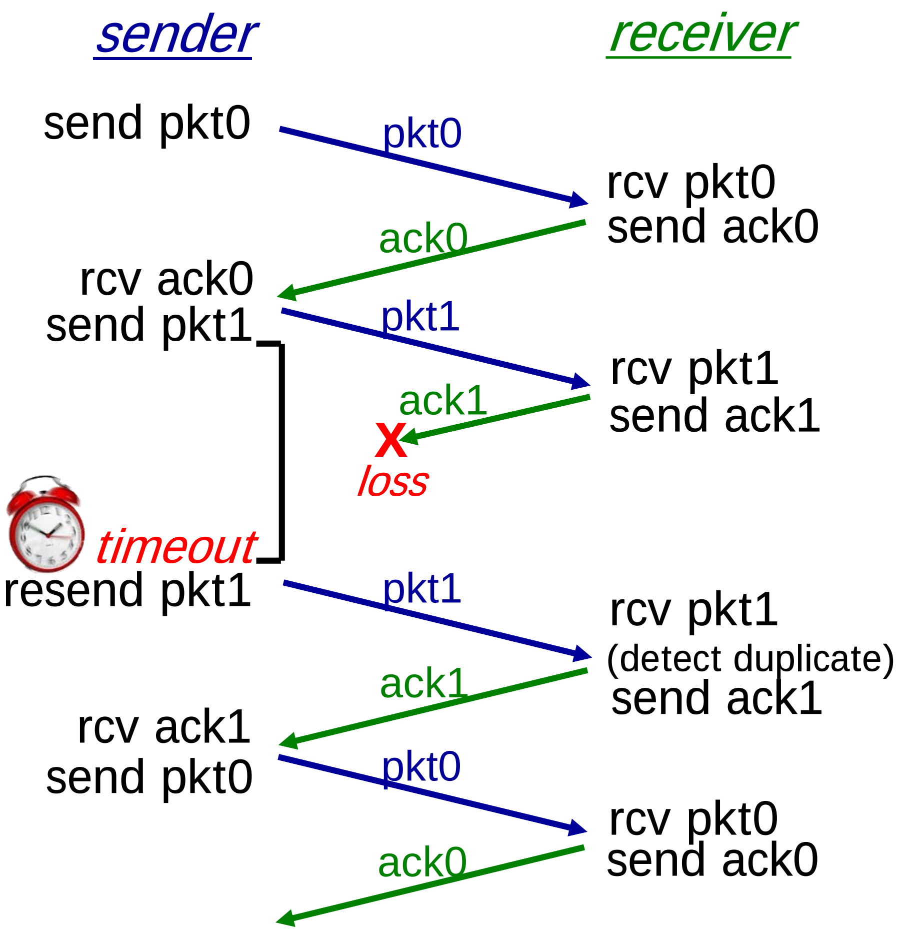
rdt3.0 in action



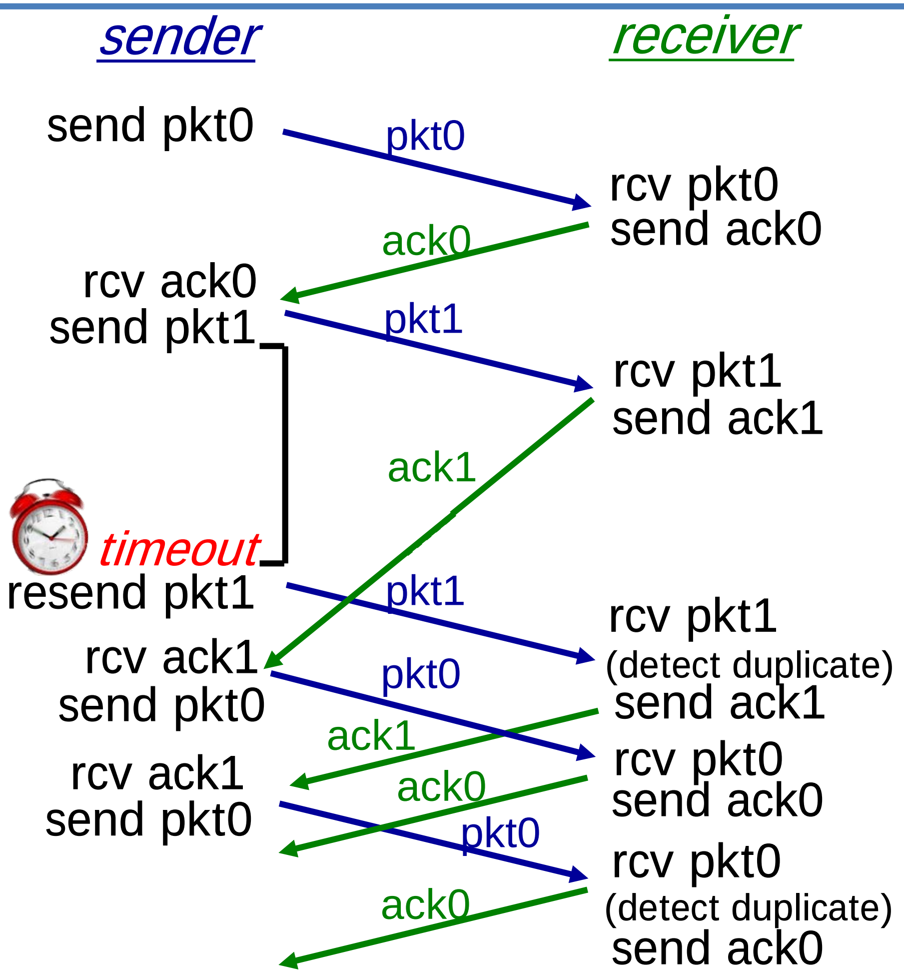
(a) no loss

(b) packet loss

rdt3.0 in action

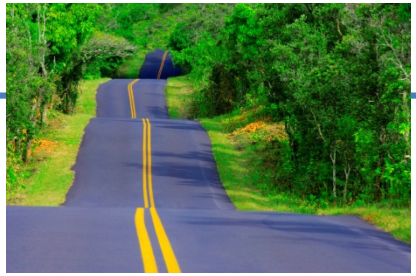


(c) ACK loss



(d) premature timeout/ delayed ACK

Roadmap



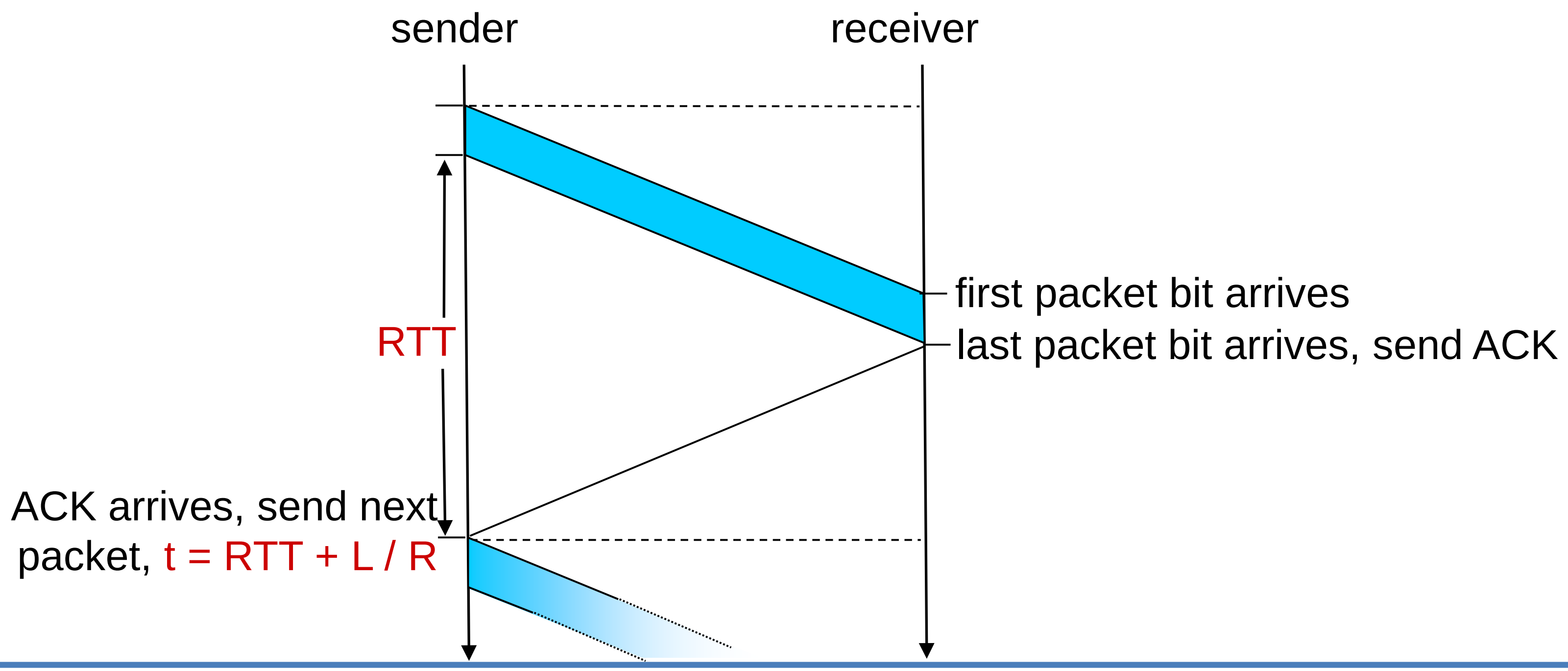
- Transport layer services in Internet
- Addressing, multiplexing/demultiplexing
- Connectionless, unreliable transport: UDP
- principles of reliable data transfer
  - Efficiency perspective

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

Performance of rdt3.0 (stop&wait)

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

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

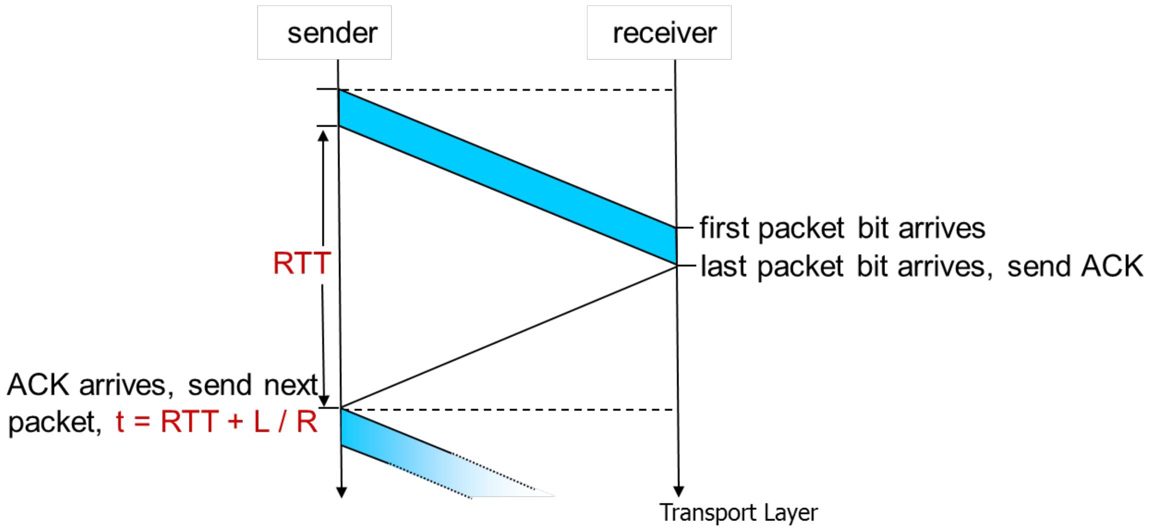


Performance of rdt3.0 (cont)

Utilization (fraction of time sender busy sending, or fraction of utilized bandwidth):

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

- Ie → approx. 300 kbps **effective throughput** over a 1 Gbps channel
- network protocol limits use of physical resources!

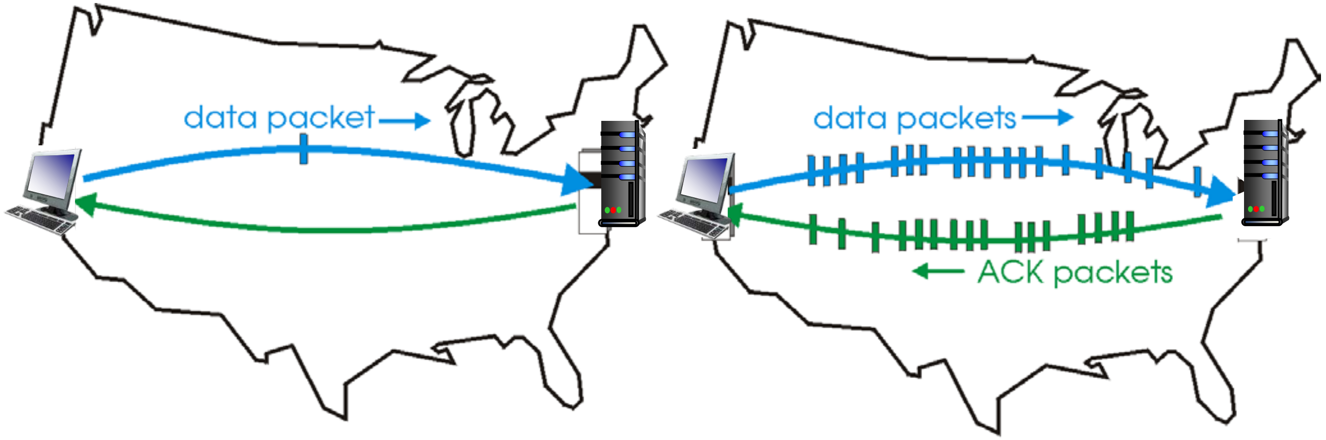


Is RDT necessarily that slow/inefficient?

Pipelined protocols

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

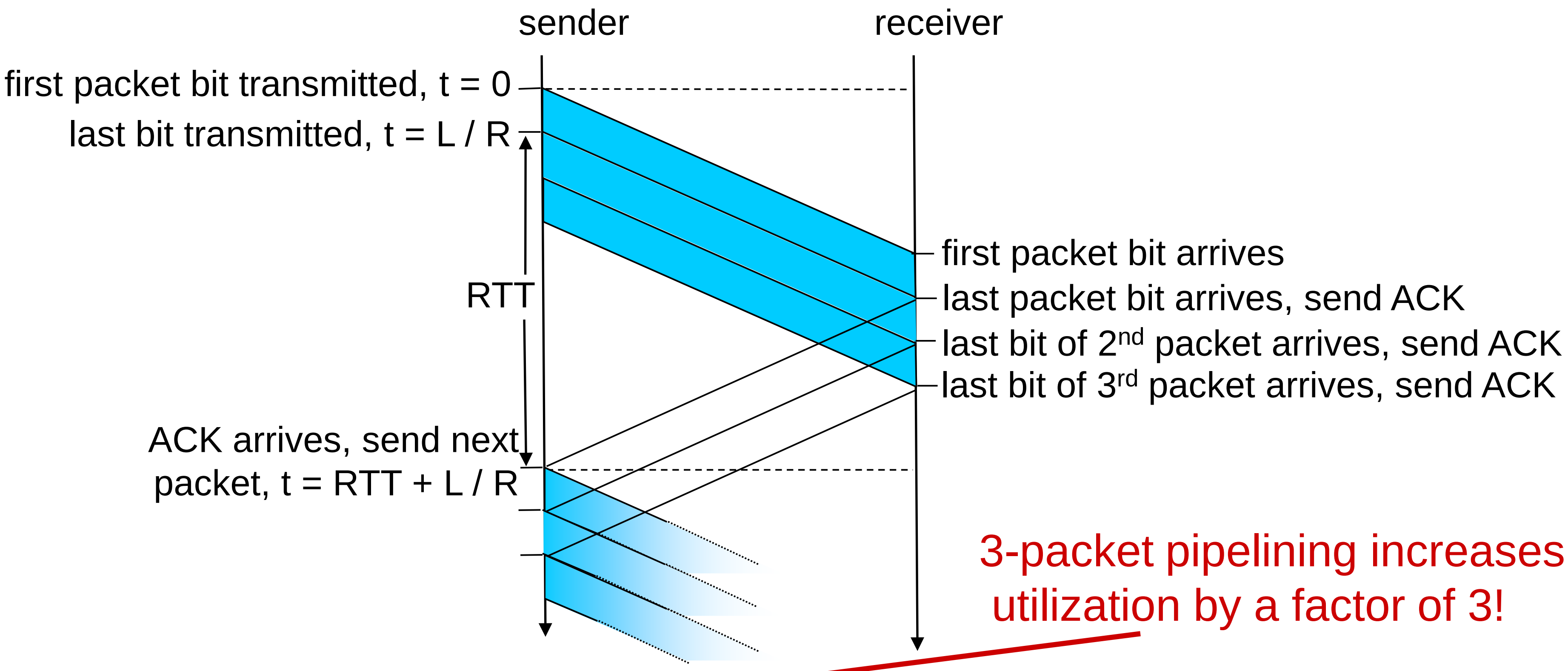
- range of sequence numbers must be increased
- buffering at sender and/or receiver



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

(b) a pipelined protocol in operation

Pipelining: increased utilization



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

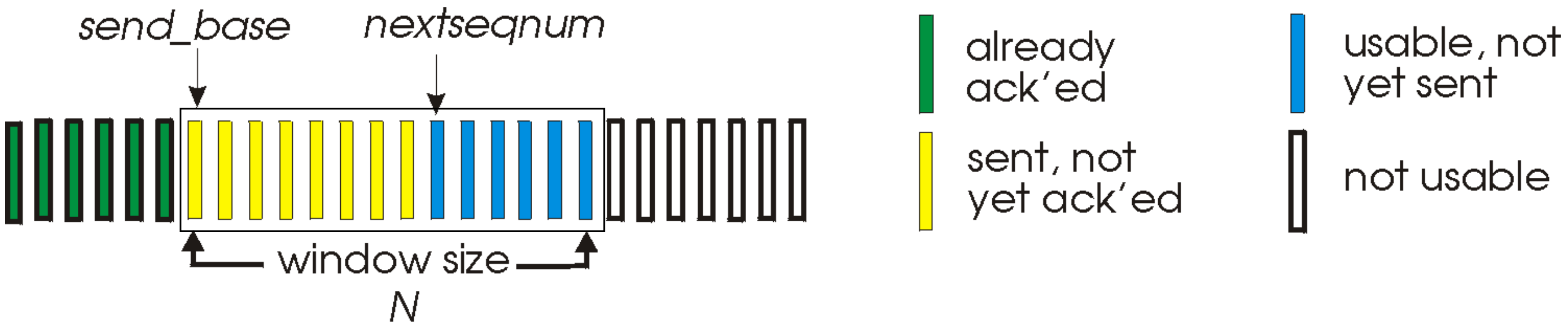
$$U_{sender} = \frac{3L / R}{RTT + L / R} = \frac{.0024}{30.008} = 0.00081$$

Pipelined protocols: ack-based error control

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

Go-Back-n: sender

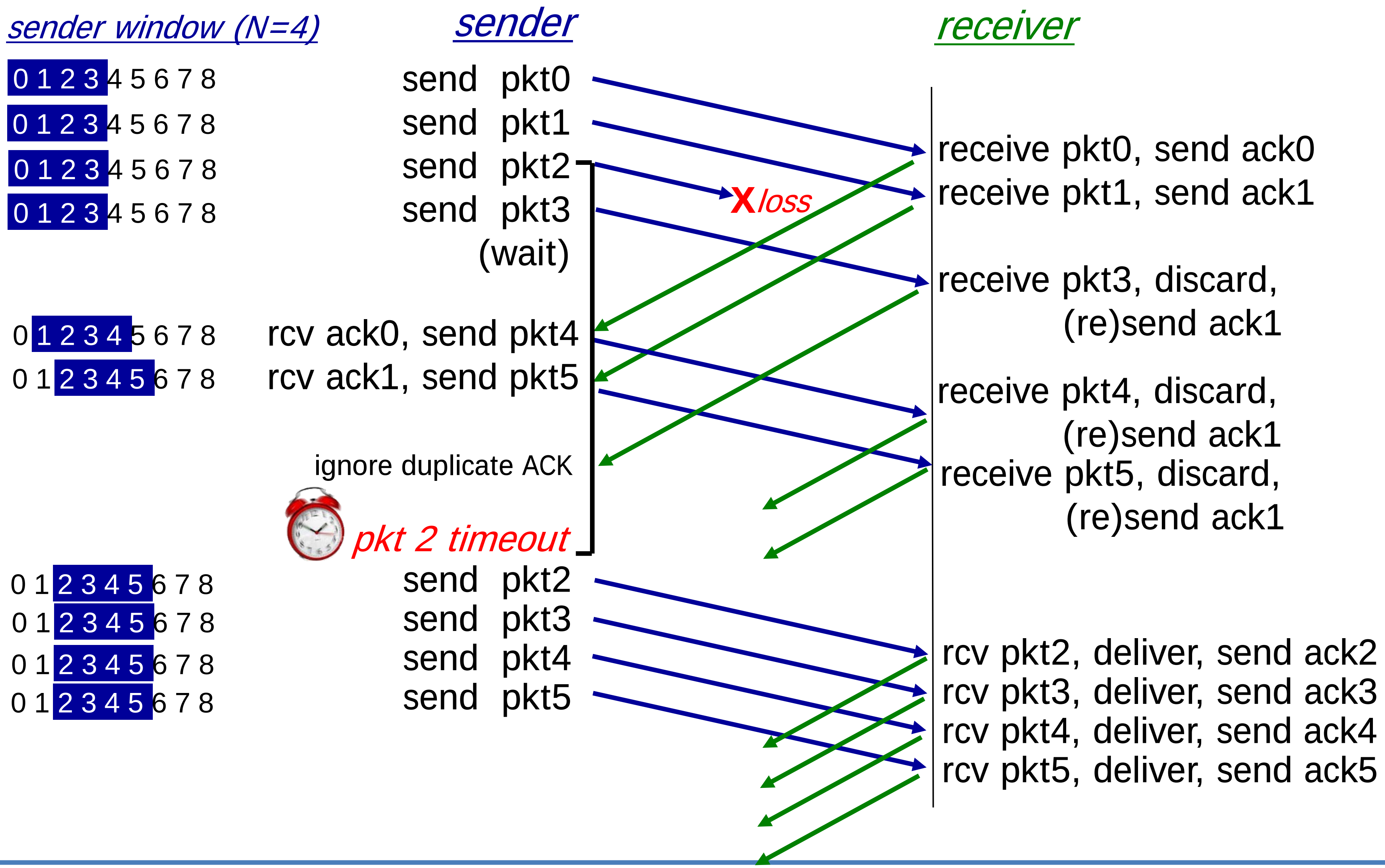
- “window” of up to N, consecutive unack’ed pkts allowed



- ACK(n): ACKs all pkts up to, including seq # n - *“cumulative ACK”*
  - may receive duplicate ACKs
- timer for oldest in-flight pkt
- *timeout(n)*: retransmit packet n and all higher seq # pkts in window

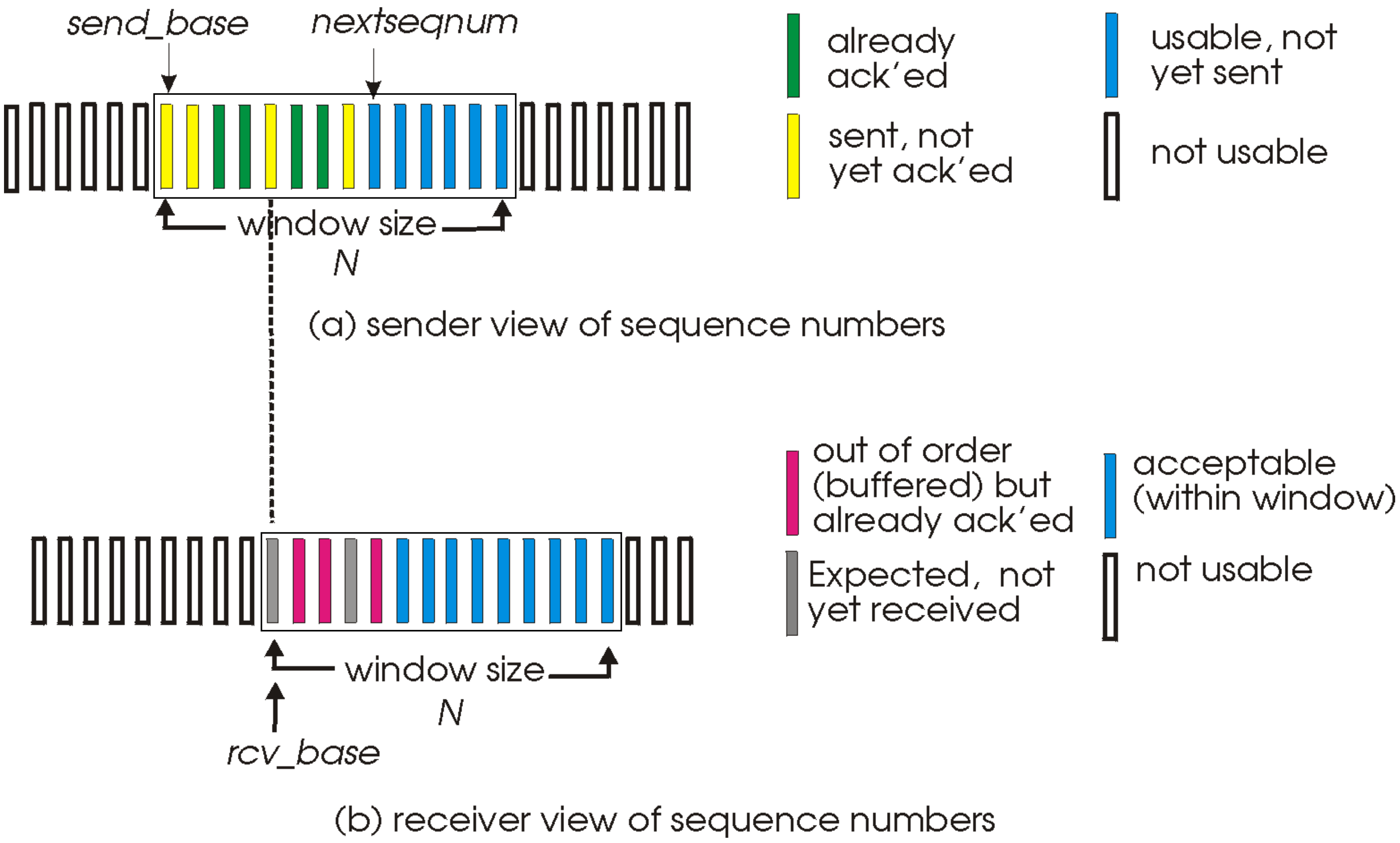
GBn in action

https://media.pearsoncmg.com/aw/ecs_kurose_compnetwork_7/cw/content/interactiveanimations/go-back-n-protocol/index.html



Selective repeat: sender, receiver windows

- receiver *individually* acknowledges received pkts
  - buffers pkts for eventual in-order delivery to upper layer
- sender only resends pkts for which ACK not received
  - Requires timer for each unACKed pkt



Selective repeat

Sender: upon…

...data from above:

- if next_pkt_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: upon receiving…

… pkt n in [rcvbase, rcvbase+N-1]

- send ACK(n)
- If out-of-order: buffer
- If in-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt

…pkt n in [rcvbase-N,rcvbase-1]

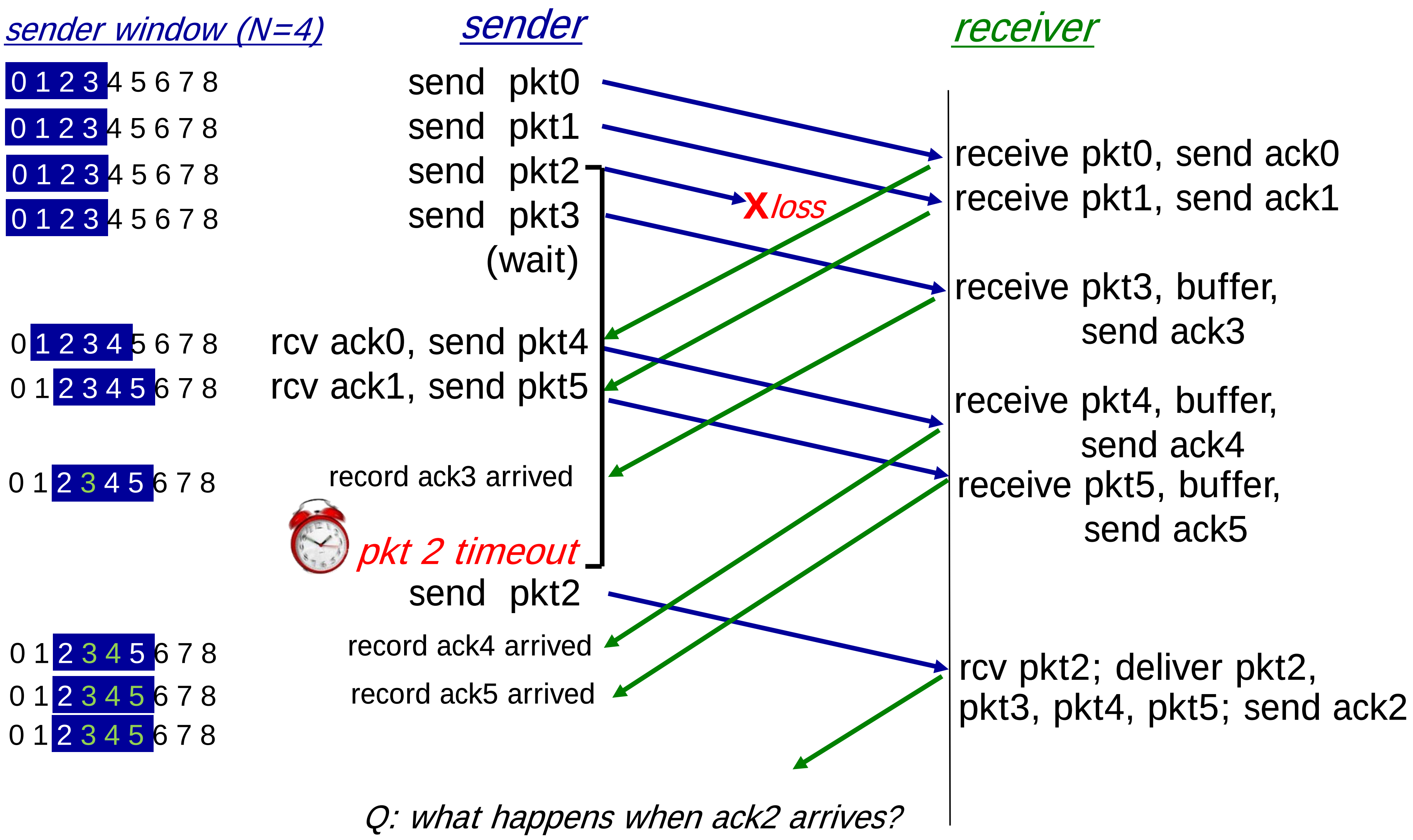
- ACK(n)

otherwise:

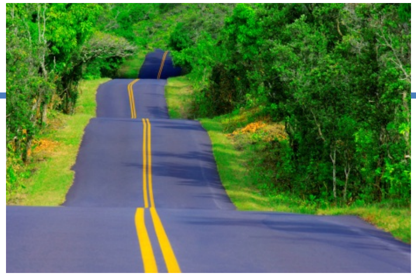
- ignore

Selective repeat in action

https://media.pearsoncmg.com/aw/ecs_kurose_compnetwork_7/cw/content/interactiveanimations/selective-repeat-protocol/index.html



Roadmap



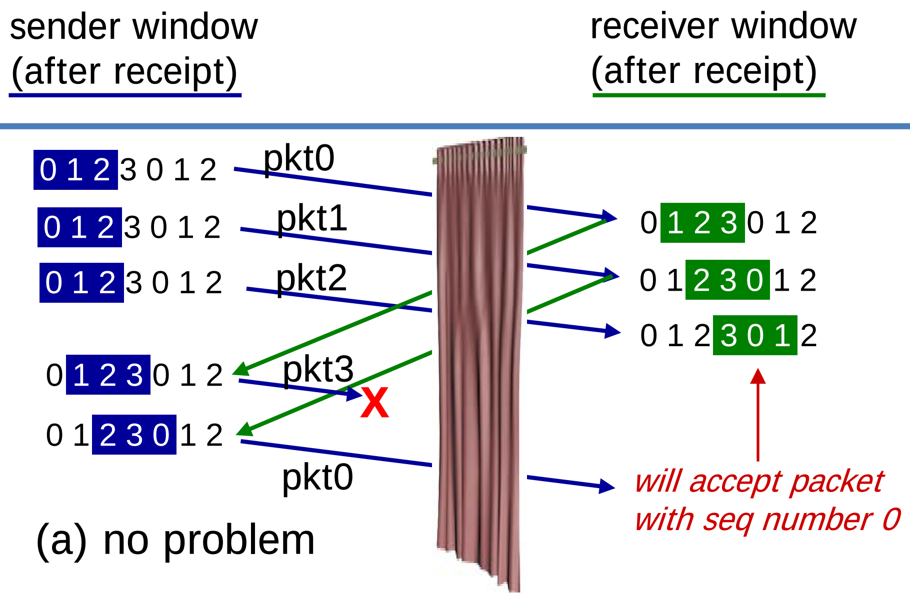
- Transport layer services in Internet
- Addressing, multiplexing/demultiplexing
- Connectionless, unreliable transport: UDP
- principles of reliable data transfer
  - Efficiency perspective: pipelined protocols & error control through go-back-n, selective-repeat
    - Sequence numbers
- *Next: connection-oriented transport: TCP*
  - *reliable transfer*
  - *flow control*
  - *connection management*
  - *TCP congestion control*

Selective repeat: Sequence numbers

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

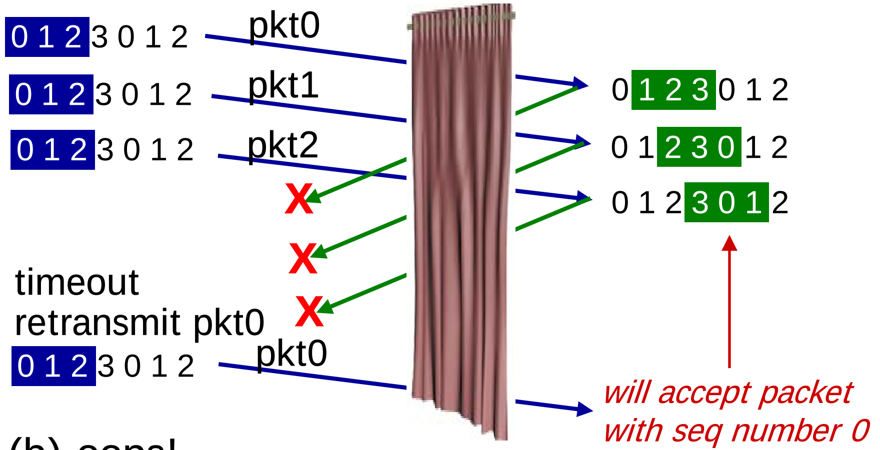
❖ duplicate data accepted as new in (b)

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



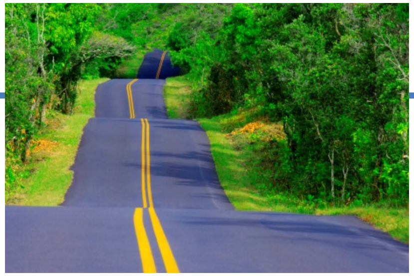
(a) no problem

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



(b) oops!

Roadmap



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

- *Next: 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?
- th errors, for the following, for a pair of sender-receive

  ion delay and transmission delay > propagation delay

  vindow's transmission delay < propagation delay and

  elay; illustrate both go-back-n and selective repeat

  ut between S-R in the above cases, when there are no
-
- ce numbers when the sender may retransmit due to
- data transfer session if the sequence-numbers range is
- 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:*

Proof sketch: 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