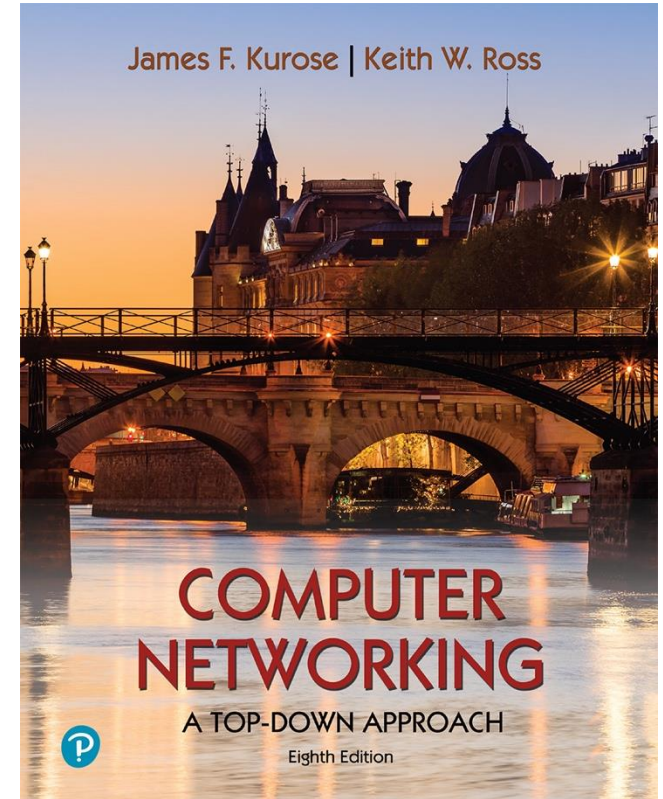# Computer Networks

Amir Mahdi Sadeghzadeh, Ph.D.

Chapter 3
Transport Layer

Computer Networking: A Top-Down Approach

8th edition
Jim Kurose, Keith Ross
Pearson, 2020

Transport services and protocols

- provide *logical communication* between application processes running on different hosts

- transport protocols actions in end systems:
  - sender: breaks application messages into *segments*, passes to network layer
  - receiver: reassembles segments into messages, passes to application layer

- two transport protocols available to Internet applications
  - TCP, UDP
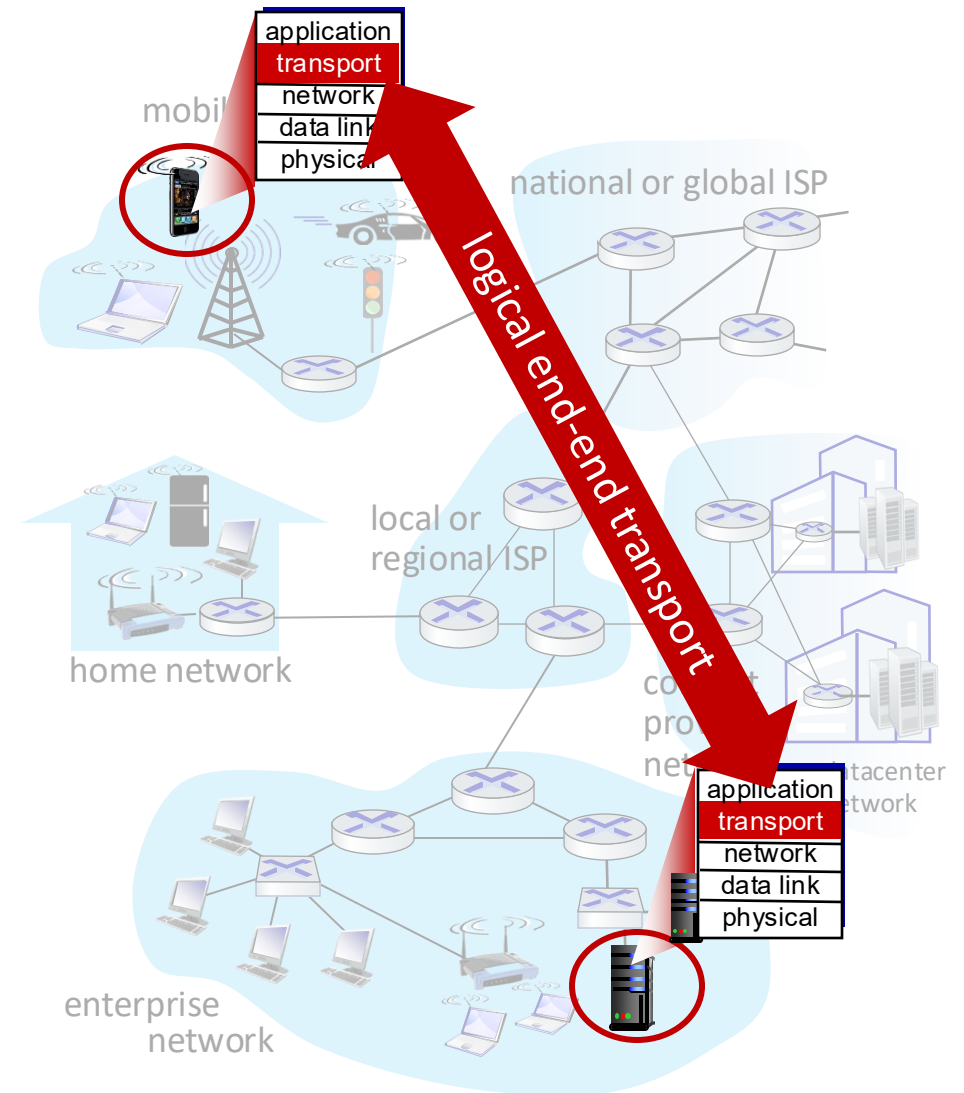
Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- **Principles of reliable data transfer**
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality
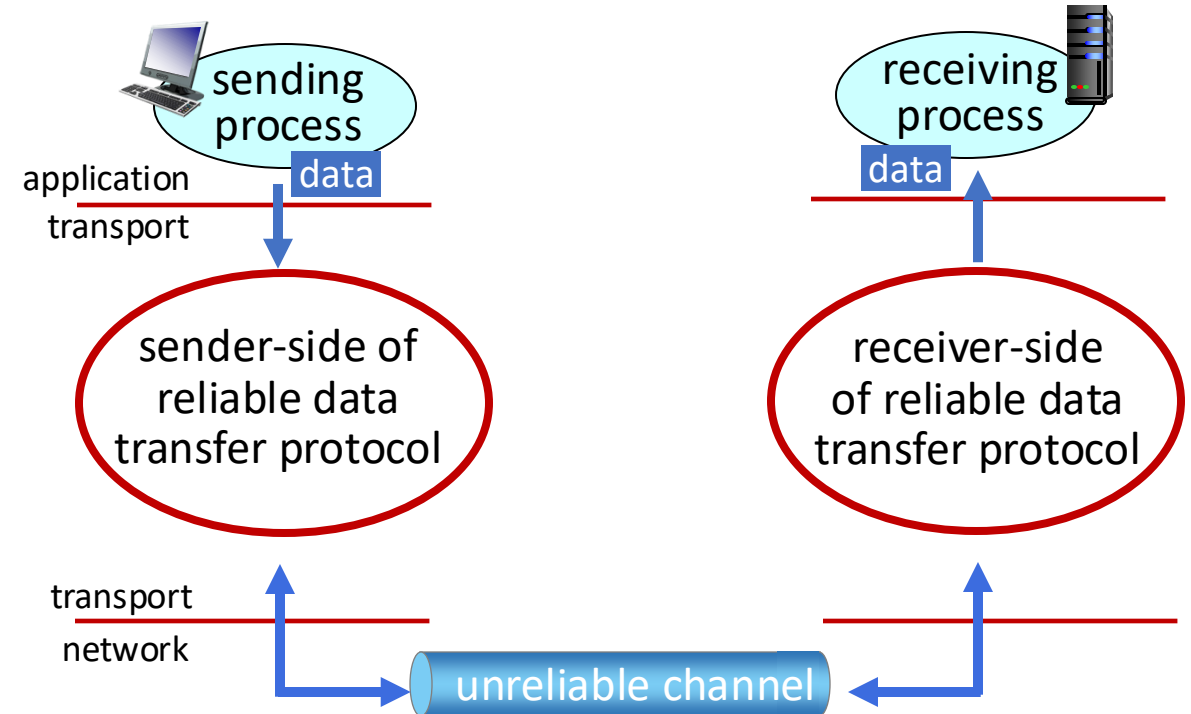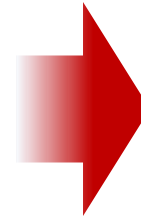
Principles of reliable data transfer

reliable service *abstraction*

Principles of reliable data transfer

reliable service *abstraction*

reliable service *implementation*

Principles of reliable data transfer

Complexity of reliable data transfer protocol will depend (strongly) on characteristics of unreliable channel (lose, corrupt, reorder data?)

sending process

data

application
transport

sender-side of reliable data transfer protocol

receiving process

data

receiver-side of reliable data transfer protocol

transport
network

unreliable channel

reliable service *implementation*
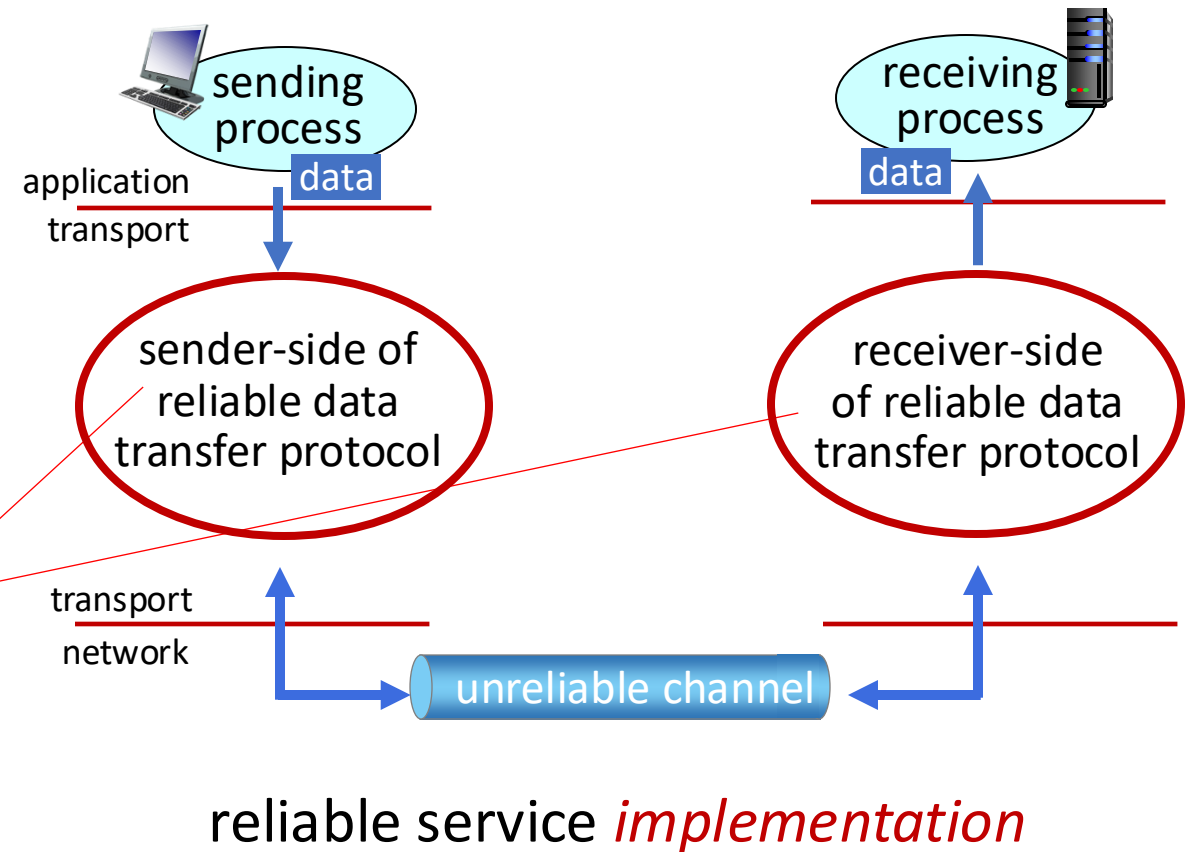
Reliable data transfer protocol (rdt): interfaces

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 layer

`rdt_send()`

sending process

data

receiving process

data

`deliver_data()`

data

sender-side *implementation* of `rdt` reliable data transfer protocol

receiver-side *implementation* of `rdt` reliable data transfer protocol

packet

`udt_send()`

Header data

Header data

`rdt_rcv()`

unreliable channel

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

Bi-directional communication over unreliable channel

rdt_rcv(): called when packet arrives on receiver side of 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

stop and wait
  sender sends one packet, then waits for receiver response

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

stop and wait

sender sends one packet, then waits for receiver response

rdt2.1: discussion

<u>sender:</u>

- 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

<u>receiver:</u>

- 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*

As we will see, TCP uses this approach to be NAK-free

rdt3.0: channels with errors *and* loss

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

- checksum, sequence #s, ACKs, retransmissions will be of help ... but not quite enough

Q: How do *humans* handle lost sender-to-receiver words in conversation?

rdt3.0: channels with errors *and* loss

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 packet being ACKed
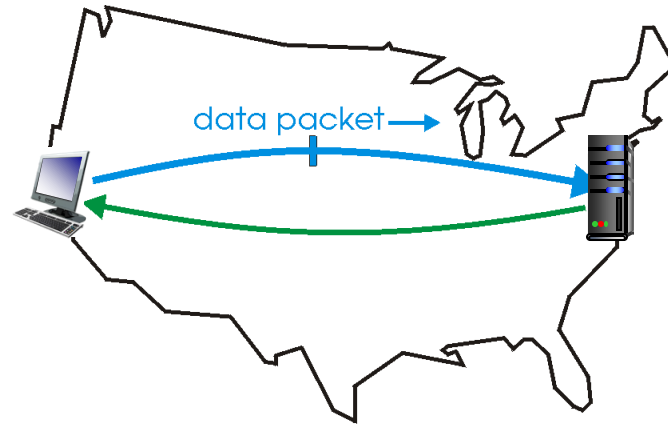- use countdown timer to interrupt after "reasonable" amount of time

timeout

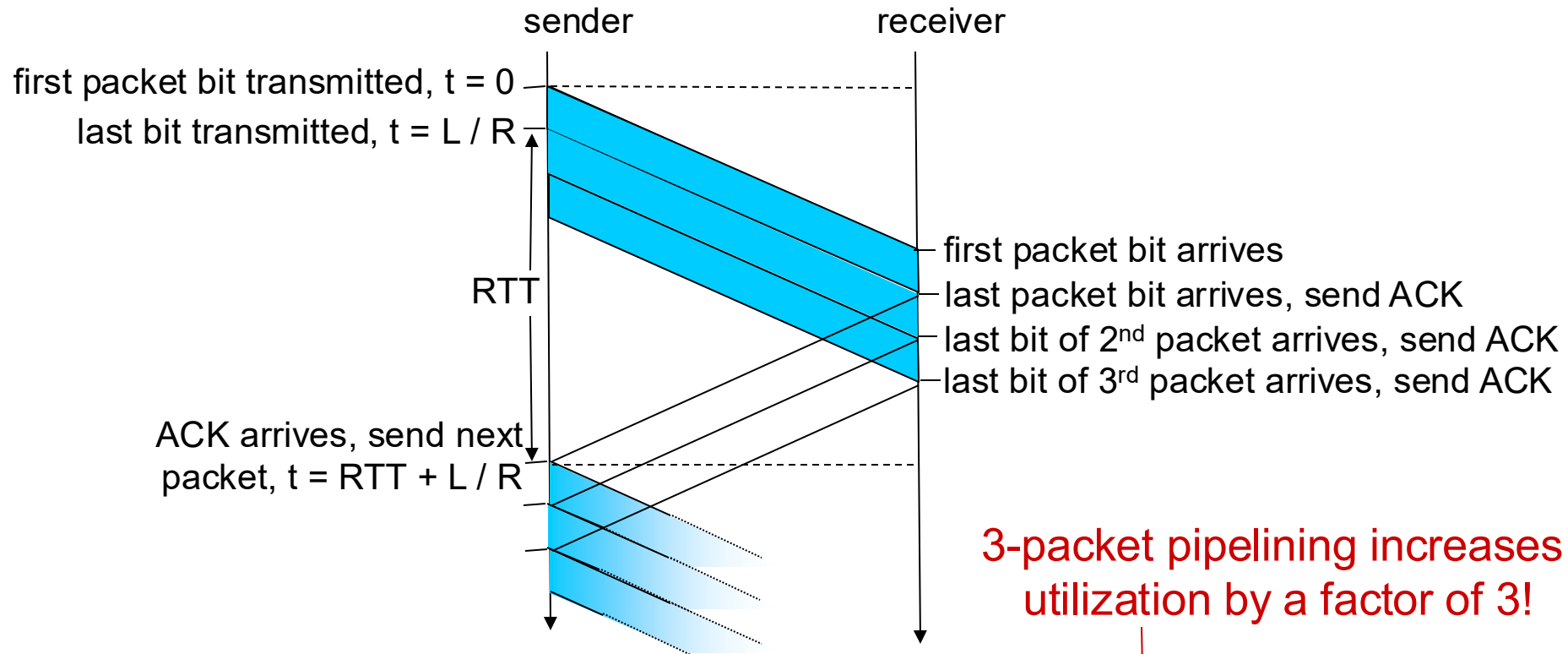# rdt3.0: pipelined protocols operation

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

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

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

Pipelining: increased utilization

sender

receiver

first packet bit transmitted, t = 0

last bit transmitted, t = L / R

RTT

first packet bit arrives

last packet bit arrives, send ACK

last bit of 2nd packet arrives, send ACK

last bit of 3rd packet arrives, send ACK

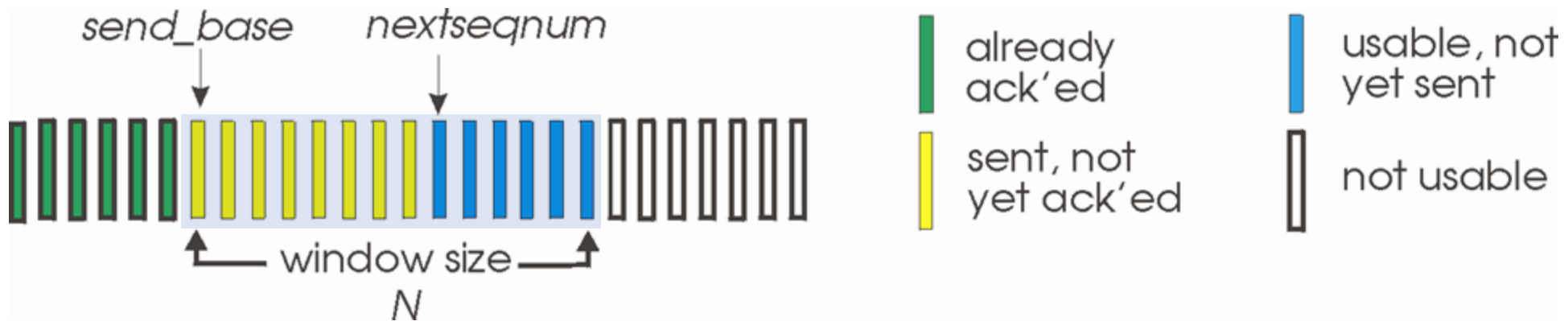ACK arrives, send next packet, t = RTT + L / R

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

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

Go-Back-N: sender

- sender: "window" of up to N, consecutive transmitted but unACKed pkts
 - k-bit seq # in pkt header

- *cumulative ACK:* ACK(n): ACKs all packets up to, including seq # n
  - on receiving ACK(n): move window forward to begin at $n+1$
- timer for oldest in-flight packet
- *timeout(n):* retransmit packet n and all higher seq # packets in window

Go-Back-N: receiver

- ACK-only: always send ACK for correctly-received packet so far, with highest *in-order* seq #
  - may generate duplicate ACKs
  - need only remember `rcv_base`

- on receipt of out-of-order packet:
  - can discard (don't buffer) or buffer: an implementation decision
  - re-ACK pkt with highest in-order seq #

Receiver view of sequence number space:

rcv_base

received and ACKed

Out-of-order: received but not ACKed
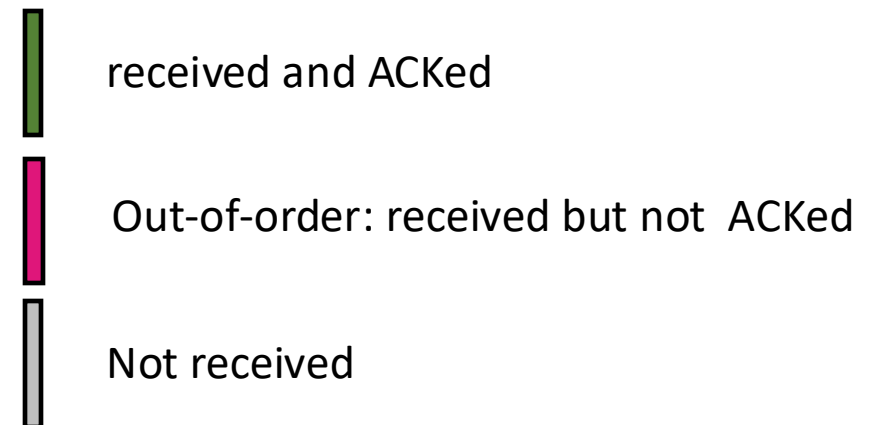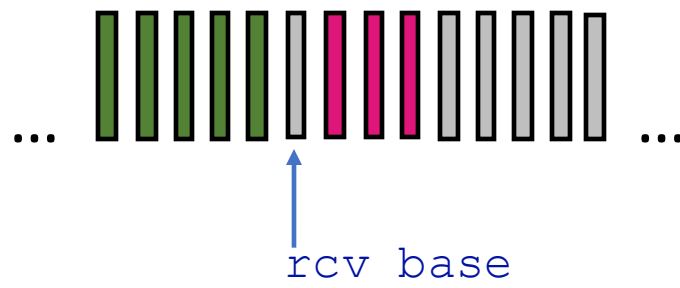
Not received

Selective repeat: the approach

- *pipelining*: *multiple* packets in flight
- *receiver individually ACKs* all correctly received packets
  - buffers packets, as needed, for in-order delivery to upper layer
- sender:
  - maintains (conceptually) a timer for each unACKed pkt
    - timeout: retransmits single unACKed packet associated with timeout
  - maintains (conceptually) "window" over *N* consecutive seq #s
    - limits pipelined, "in flight" packets to be within this window

Selective repeat: sender, receiver windows

(a) sender view of sequence numbers
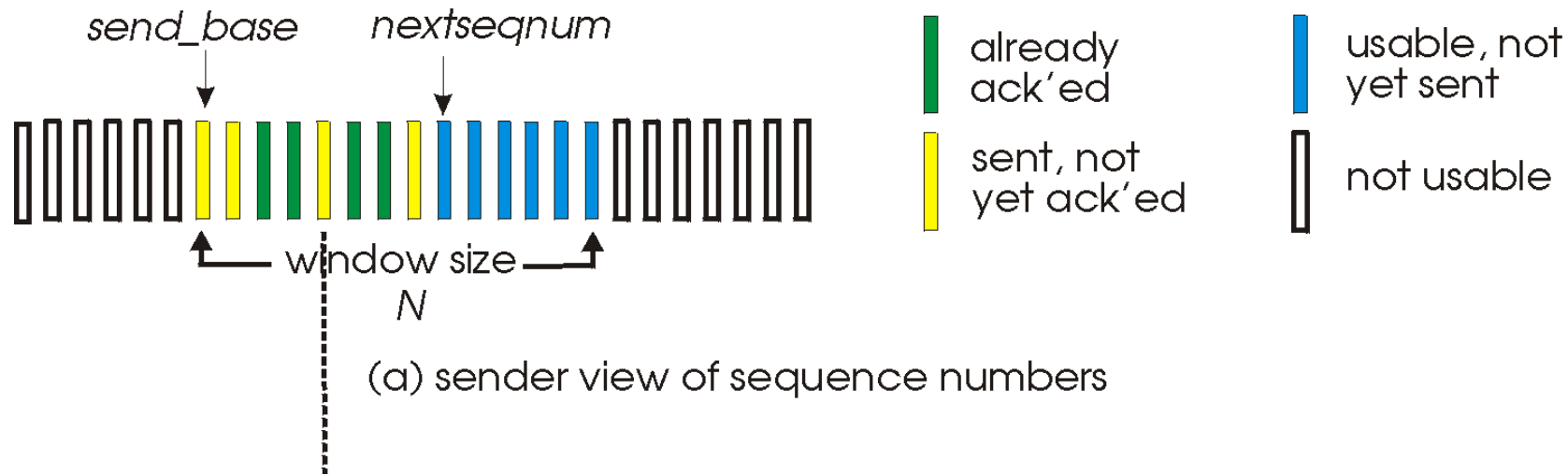
Selective repeat: sender and receiver

sender

data from above:

- if next available seq # in window, send packet

timeout(*n*):

- resend packet *n*, restart timer

ACK(*n*) in [sendbase,sendbase+N-1]:

- mark packet *n* as received

- if n smallest unACKed packet, advance window base to next unACKed seq #

receiver

packet *n* in [rcvbase, rcvbase+N-1]

- send ACK(*n*)

- out-of-order: buffer

- in-order: deliver (also deliver buffered, in-order packets), advance window to next not-yet-received packet

packet *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

`0 1 2 3` 4 5 6 7 8   send pkt1

`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)

0 `1 2 3 4` 5 6 7 8   rcv ack0, send pkt4

0 1 `2 3 4 5` 6 7 8   rcv ack1, send pkt5

record ack3 arrived

pkt 2 timeout

0 1 `2 3 4 5` 6 7 8   send pkt2

0 1 `2 3 4 5` 6 7 8   (but not 3,4,5)

0 1 `2 3 4 5` 6 7 8

0 1 `2 3 4 5` 6 7 8

receive pkt0, send ack0

receive pkt1, send ack1

receive pkt3, buffer,
        send ack3

receive pkt4, buffer,
        send ack4

receive pkt5, buffer,
        send ack5

rcv pkt2; deliver pkt2,
pkt3, pkt4, pkt5; send ack2

Q: what happens when ack2 arrives?