Connection-oriented transport: TCP

CE 352, Computer Networks

Salem Al-Agtash

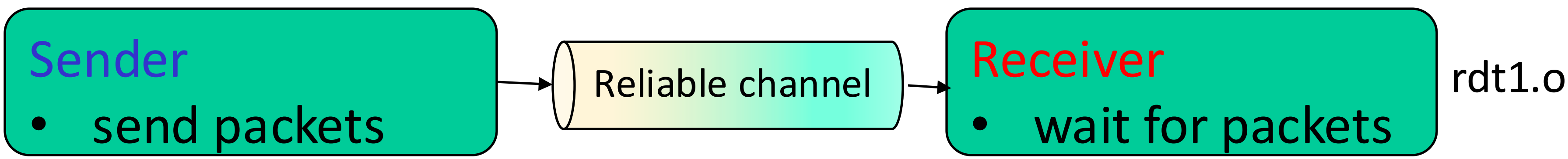
Lecture 10

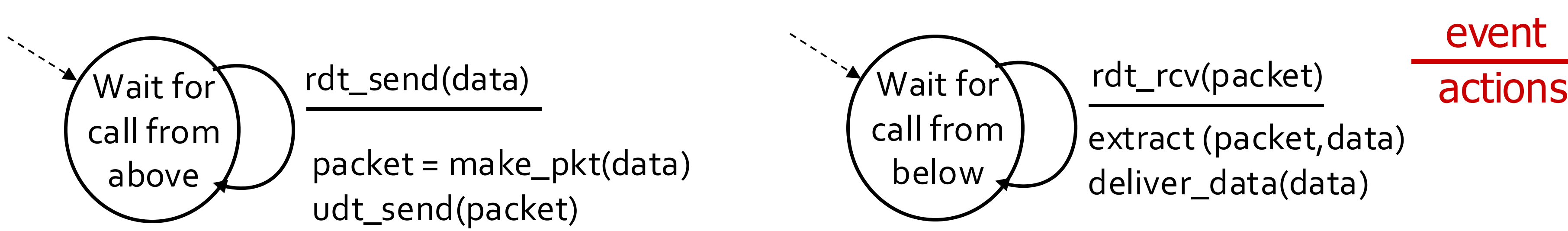
Recap (transport layer)

- Transport layer provides communication between application processes (mux/demux using ports) so that reliable and efficient data delivery is achieved.
- Network layer can result in packets that are corrupted, delayed, dropped, reordered, or duplicated.
- Network layer gives no guidance on traffic volume to send and when
- TCP and UDP are the common transport protocols
- UDP is a minimalist lightweight communication between processes
- TCP offers a reliable, in-order, and byte stream communication with congestion control

Recap (reliable transport channel)

- Reliable transport channel is easy in a perfect world
  - rdt1.0: Simple protocol
    - provides neither flow nor error control





sender receiver

Recap (unreliable transport channel)

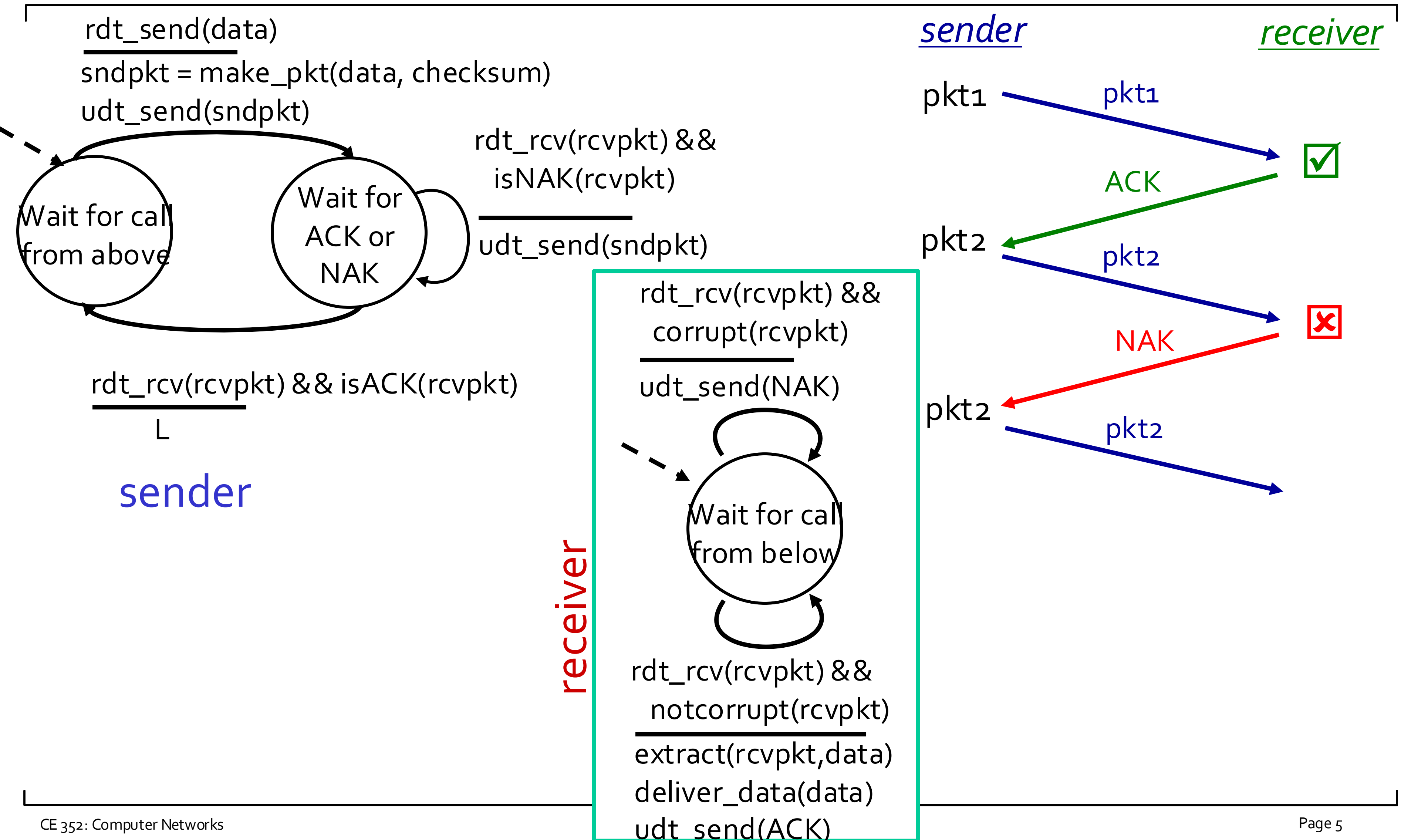
- Components of solution to send packets over unreliable channel:
  - checksums (to detect bit errors)
  - timers (to detect loss)
  - acknowledgements (Ack, NAK)
  - sequence numbers (to deal with duplicates)
- Stop-and-wait
  - rdt2.0: deals with packet corruption
  - rdt2.1: deals with garbled Ack/NAKs
  - rdt2.2: NAK-free by the use of sequence numbers
  - rdt3.0: deals with packet loss – timer driven loss detection
- Pipelined protocol (sliding window)
  - Go-Back-N
  - Selective repeat

Sender: send packet i
- set timer, wait for ACK
- if (ACK): i++, repeat
- if (NAK || timeout): repeat

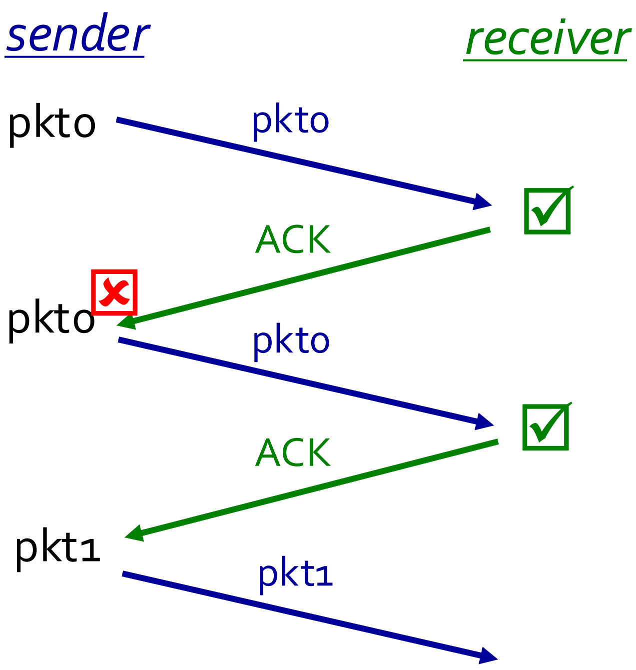
→ Unreliable channel →

Receiver: wait for packet
- if (packet is OK): send ACK
- else send NAK
- repeat

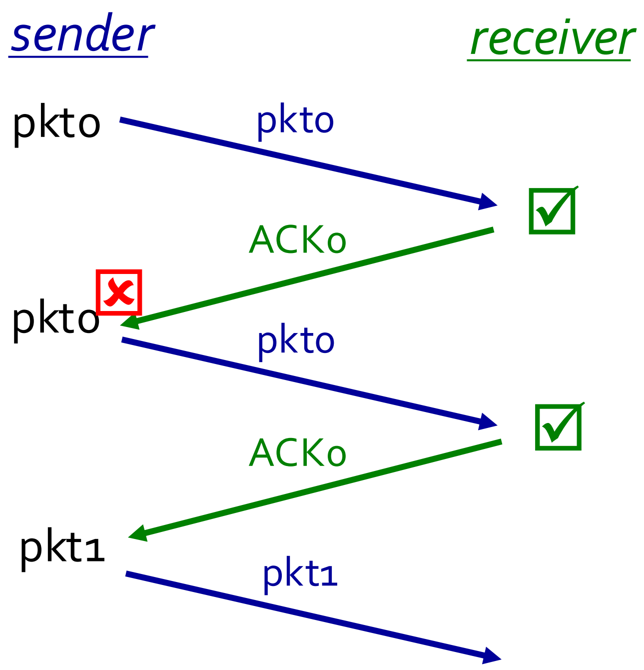
Recap (rdt2.0: deals with packet corruption)



Recap (rdt2.1/2.2: deals with garbled ACK-NAK/ data and ack packets carry sequence numbers)

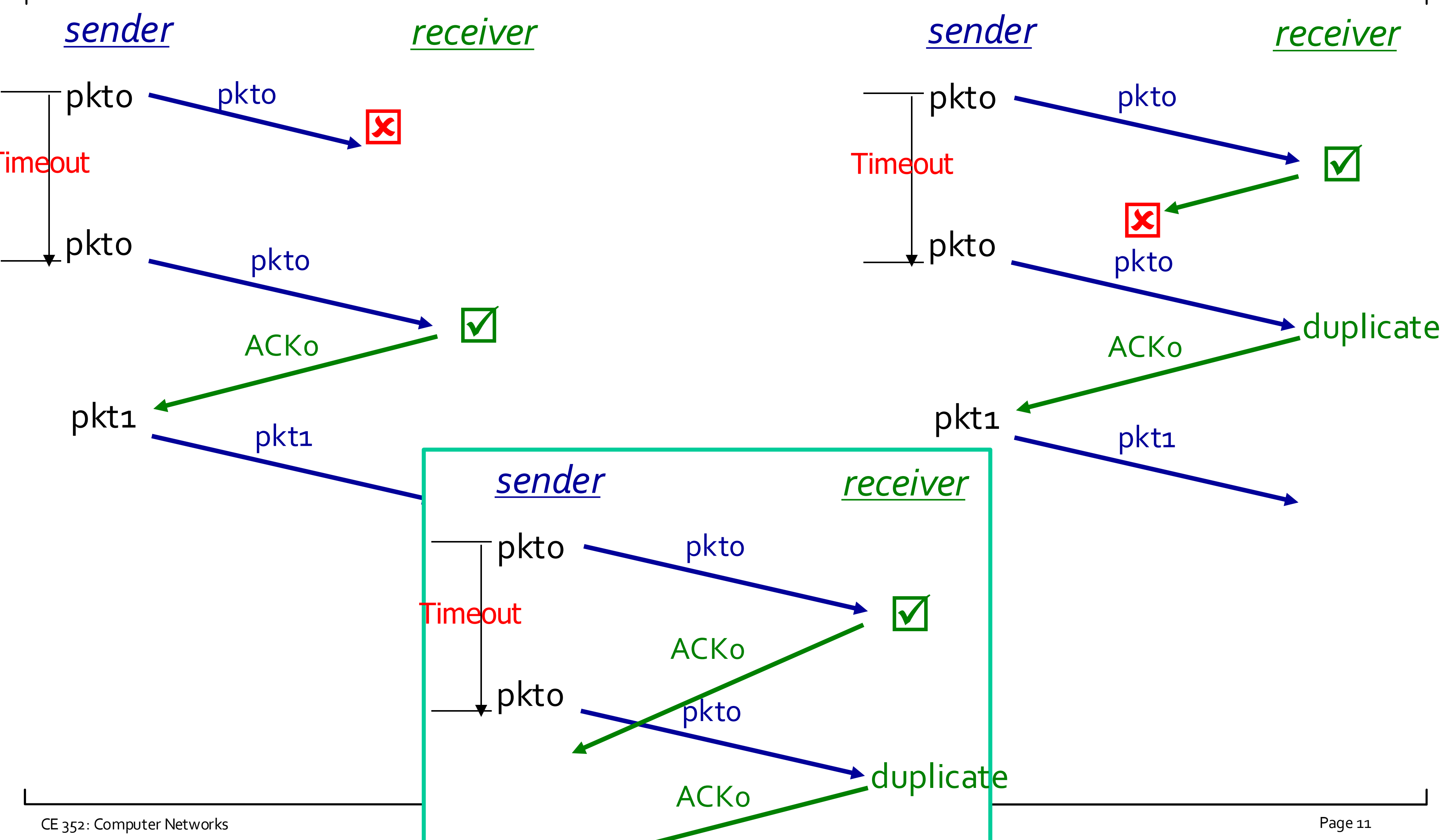


rdt2.1

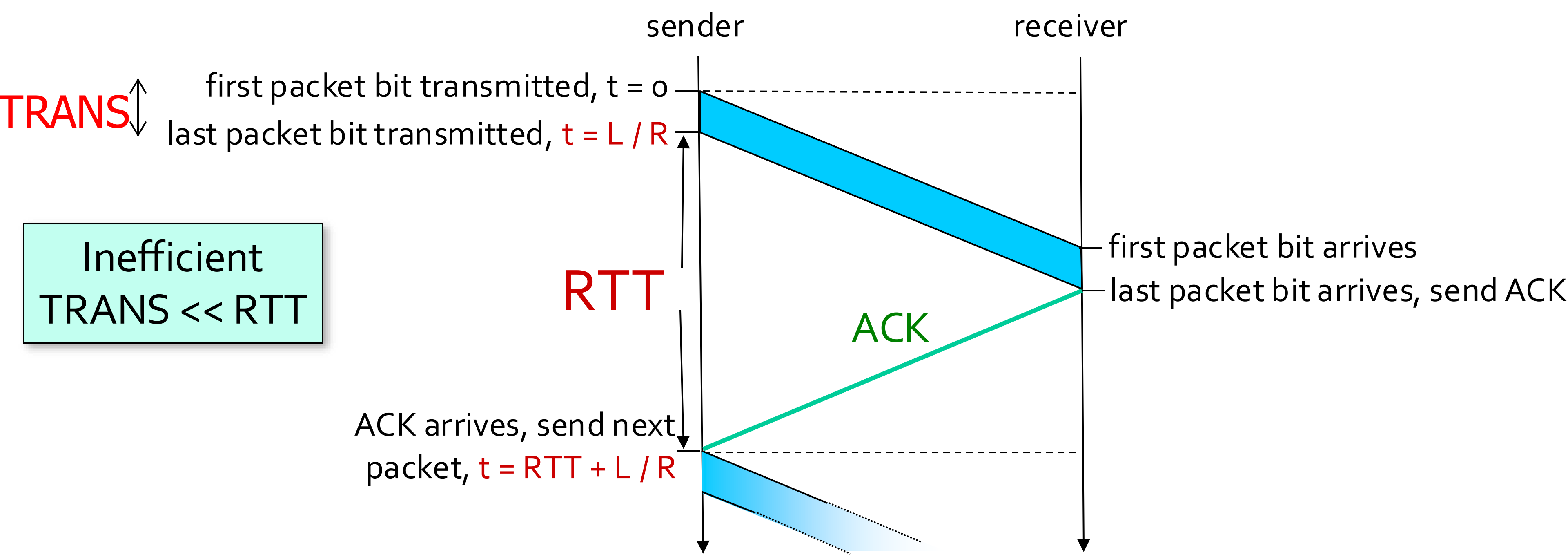


rdt2.2

Recap (rdt3.0: deals with packet loss Timer-driven loss detection)



Recap (stop-and-wait is inefficient)

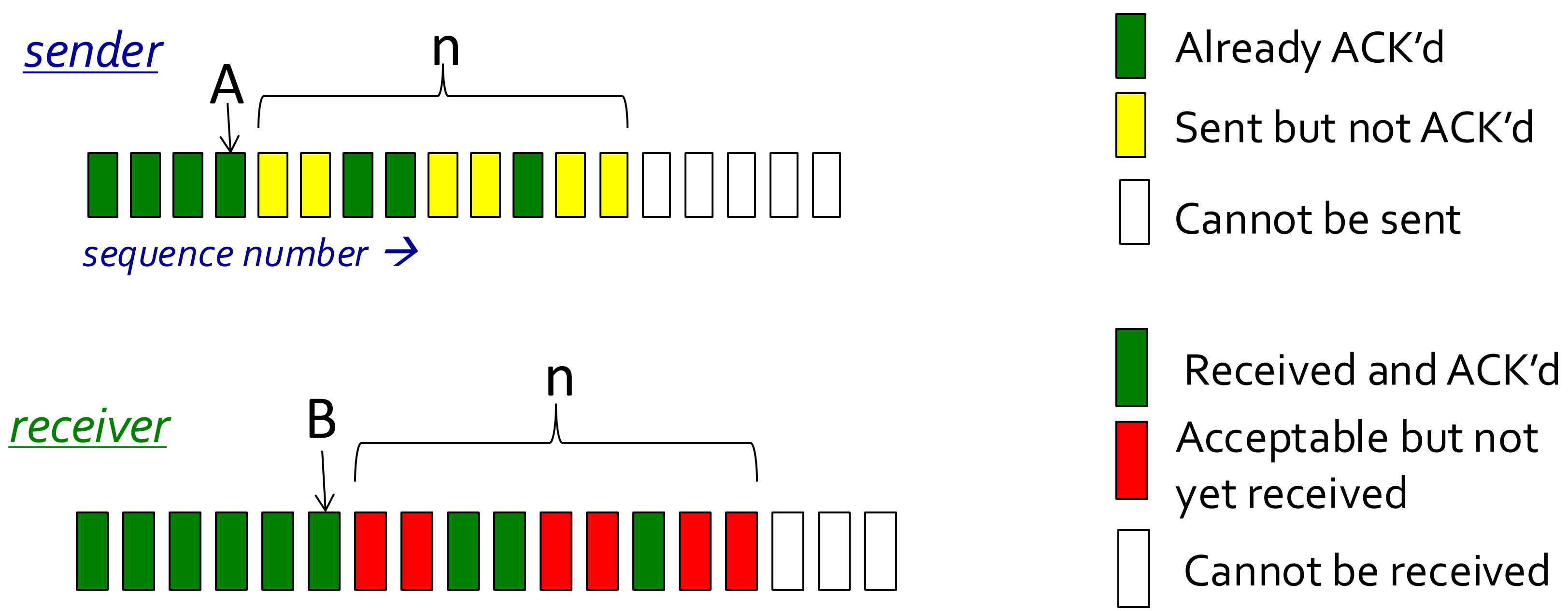


- R= 1 Gbps link, 15 ms prop. delay, L= 8000 bit packet → total t = 30.008 msec

→ 1,000 bytes in 30.008 milliseconds, gives throughput of only 267 kbps

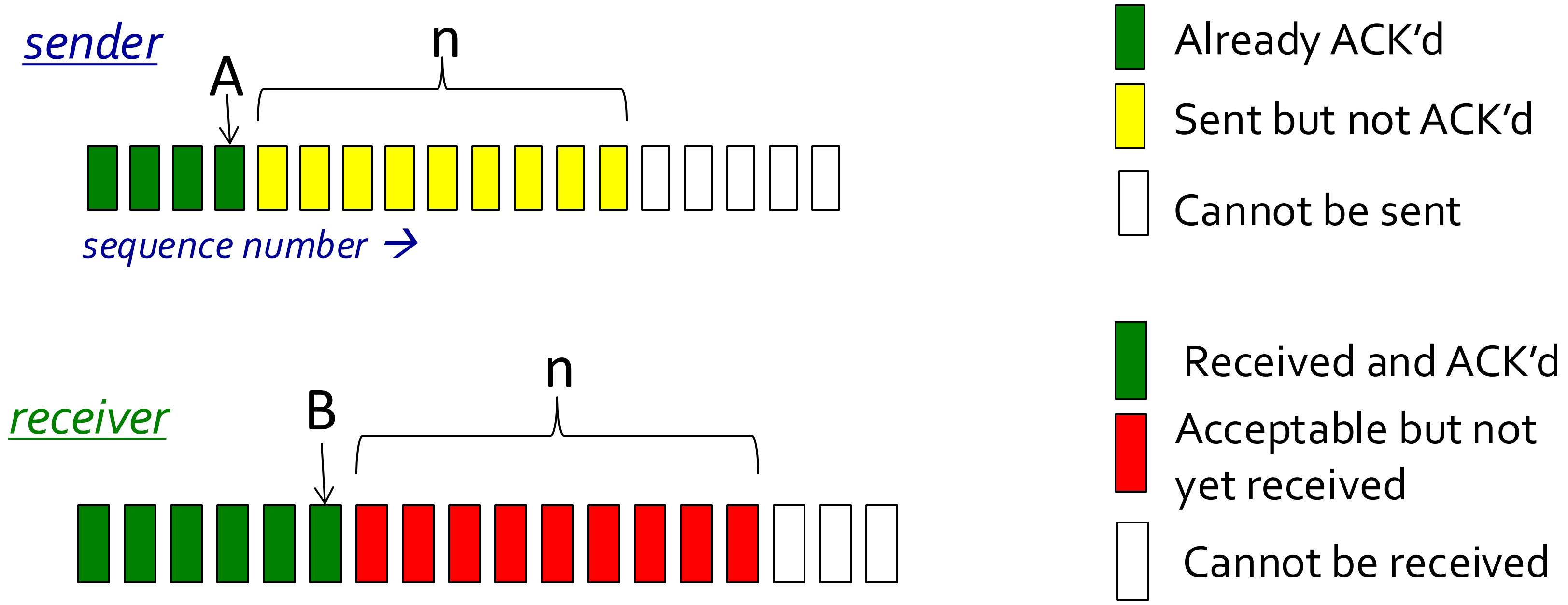
Recap (Pipelined protocol: Sliding window)

- Window: set of adjacent sequence numbers (window size *n*)
- Send up to *n* packets at a time
  - Sender can send packets in its window
  - Receiver can accept packets in its window
  - Window slides on successful reception/ acknowledgement

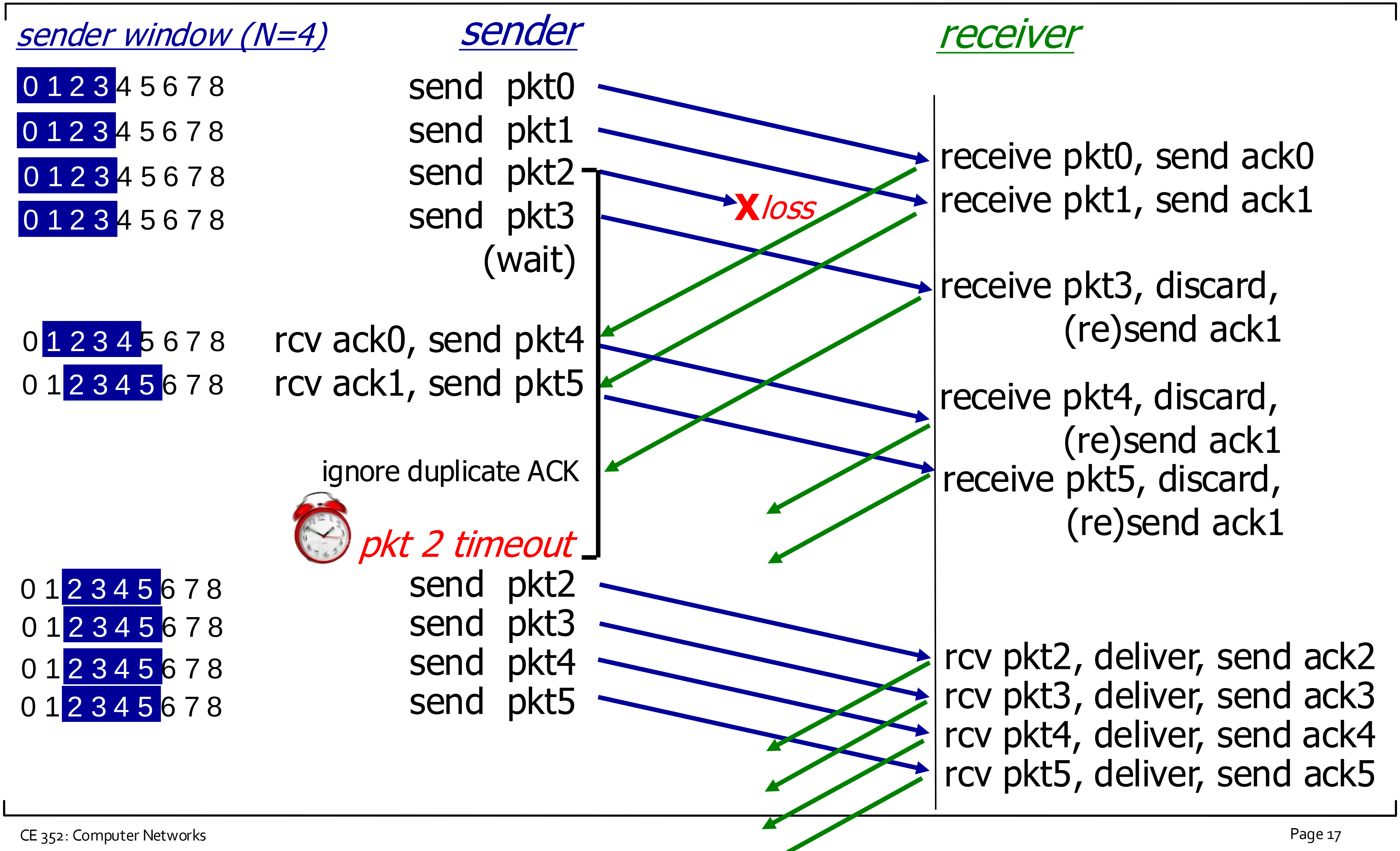


Recap (Go-Back-N)

- Sender transmits up to *n* unacknowledged packets
- Receiver only accepts packets in order
  - Receiver discards out-of-order packets
  - Receiver uses cumulative acknowledgements
  - Sender sets timer for 1st outstanding ack (A+1), if timeout, retransmit A+1, … A+n

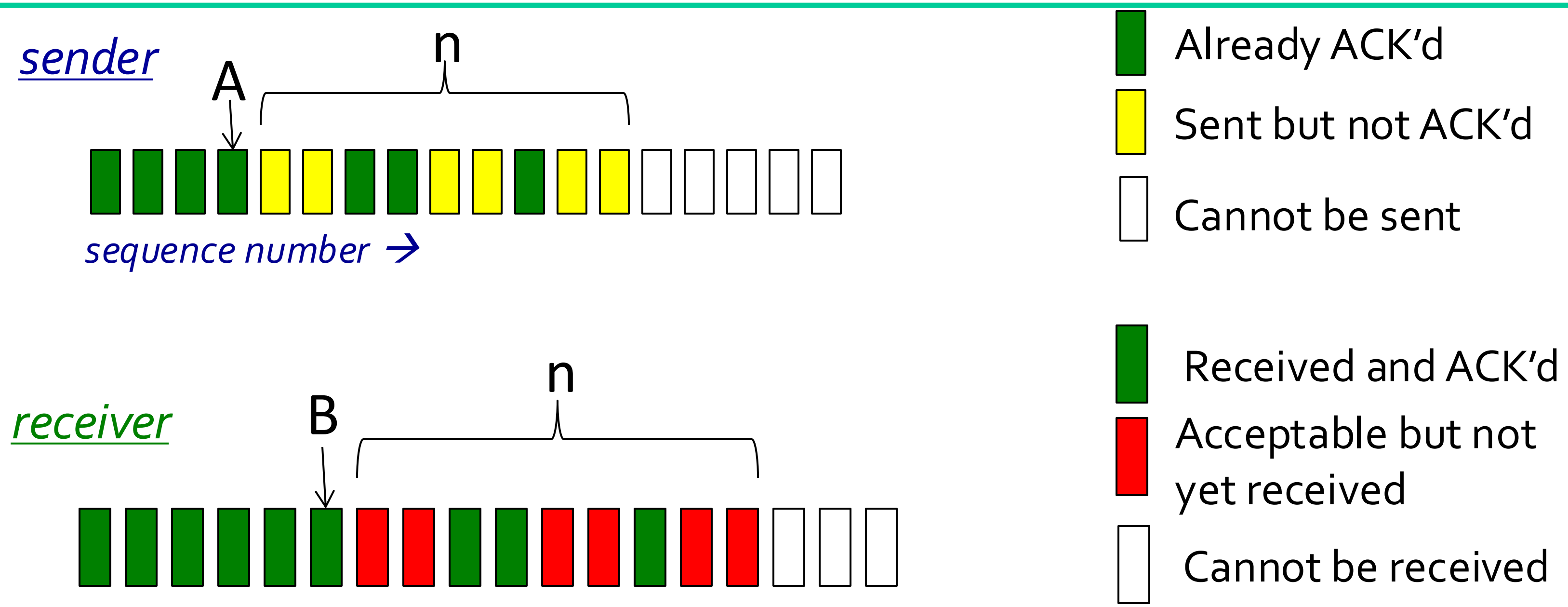


Recap (GBN example)

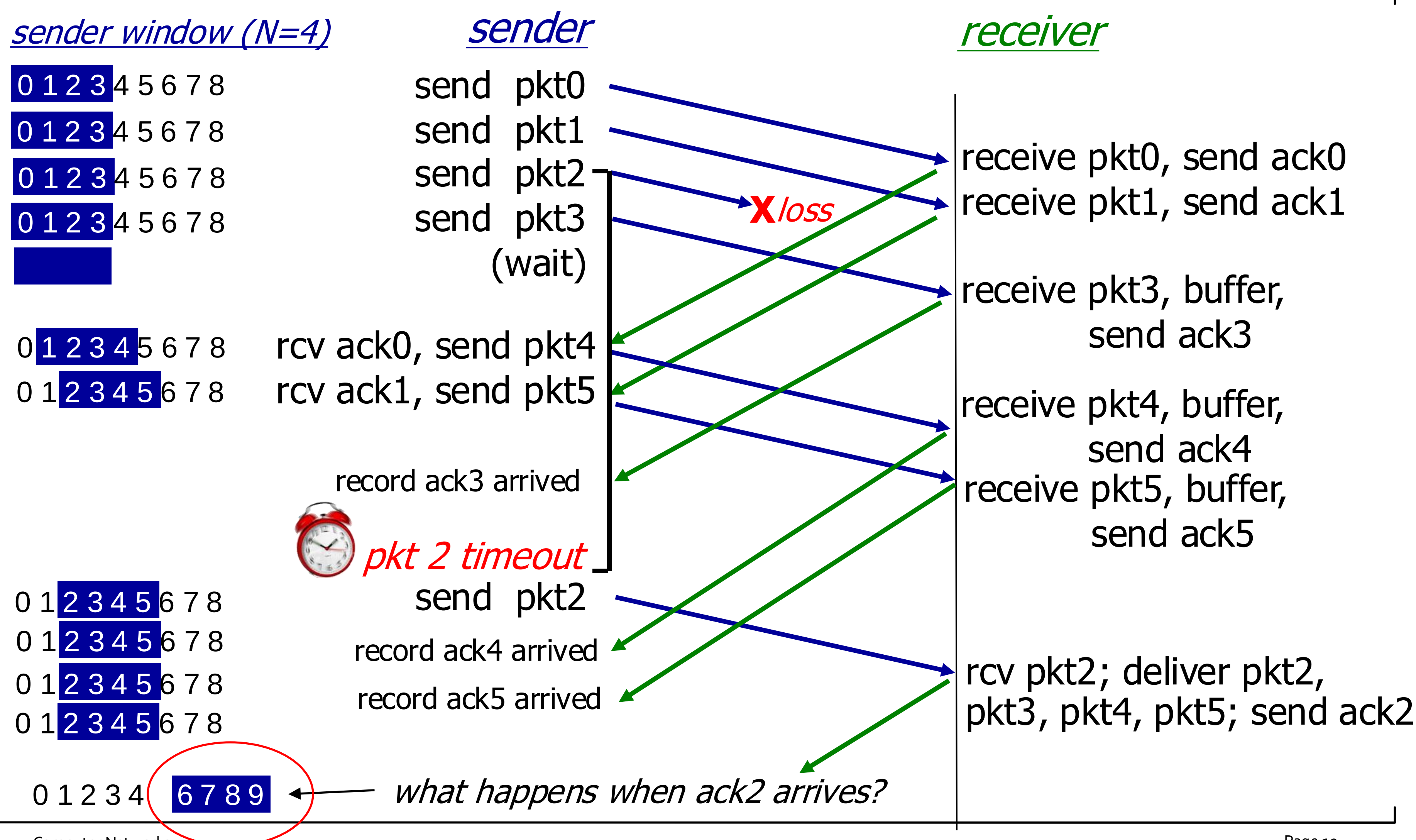


Recap (Selective repeat)

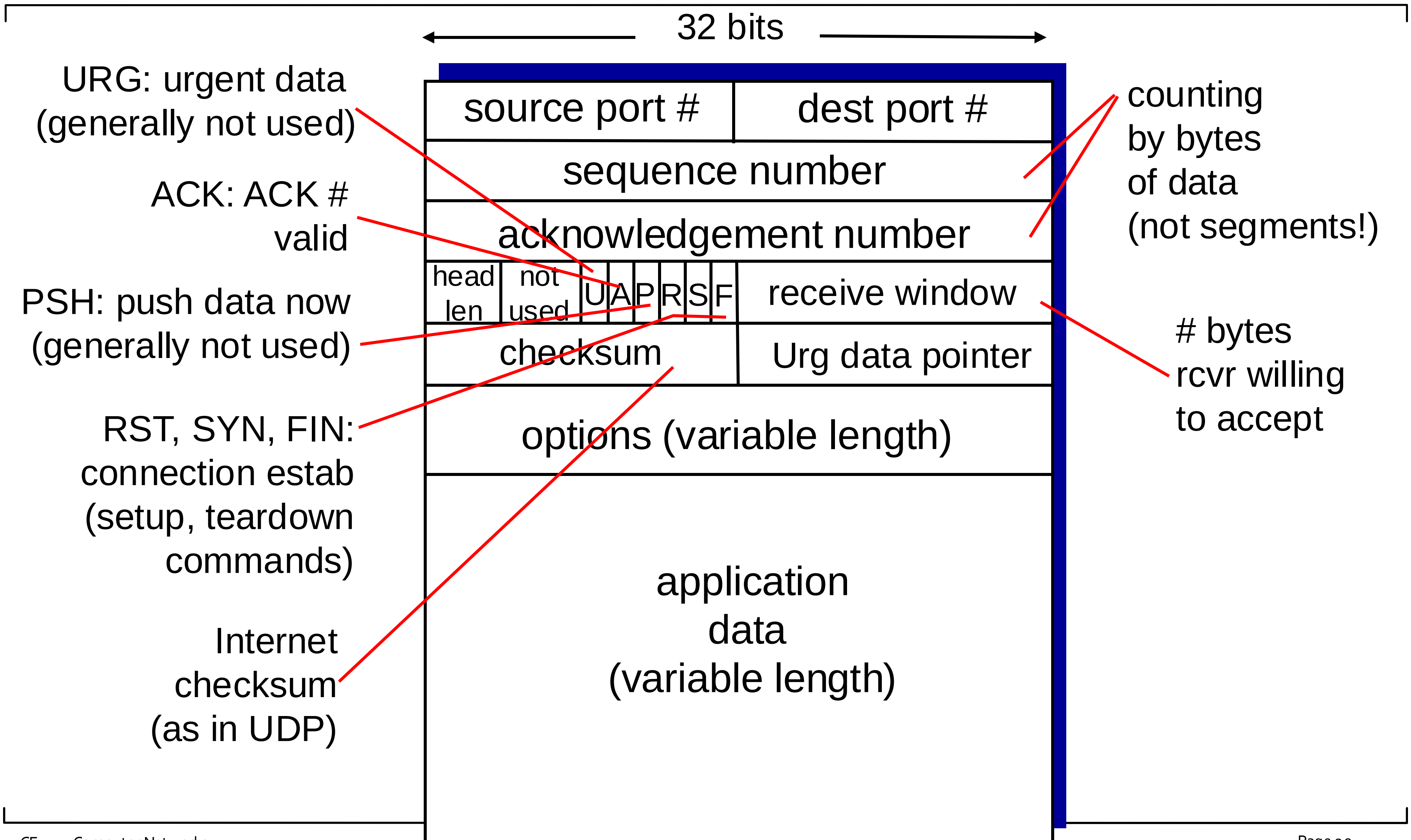
- Sender transmits up to *n* unacknowledged packets
- Assume packet m is lost, m+1 is not
- Receiver indicates packet m+1 is correctly received
- Sender retransmits only packet on m timeout
- Efficient in retransmission, but requires book-keeping



Recap (Selective repeat example)



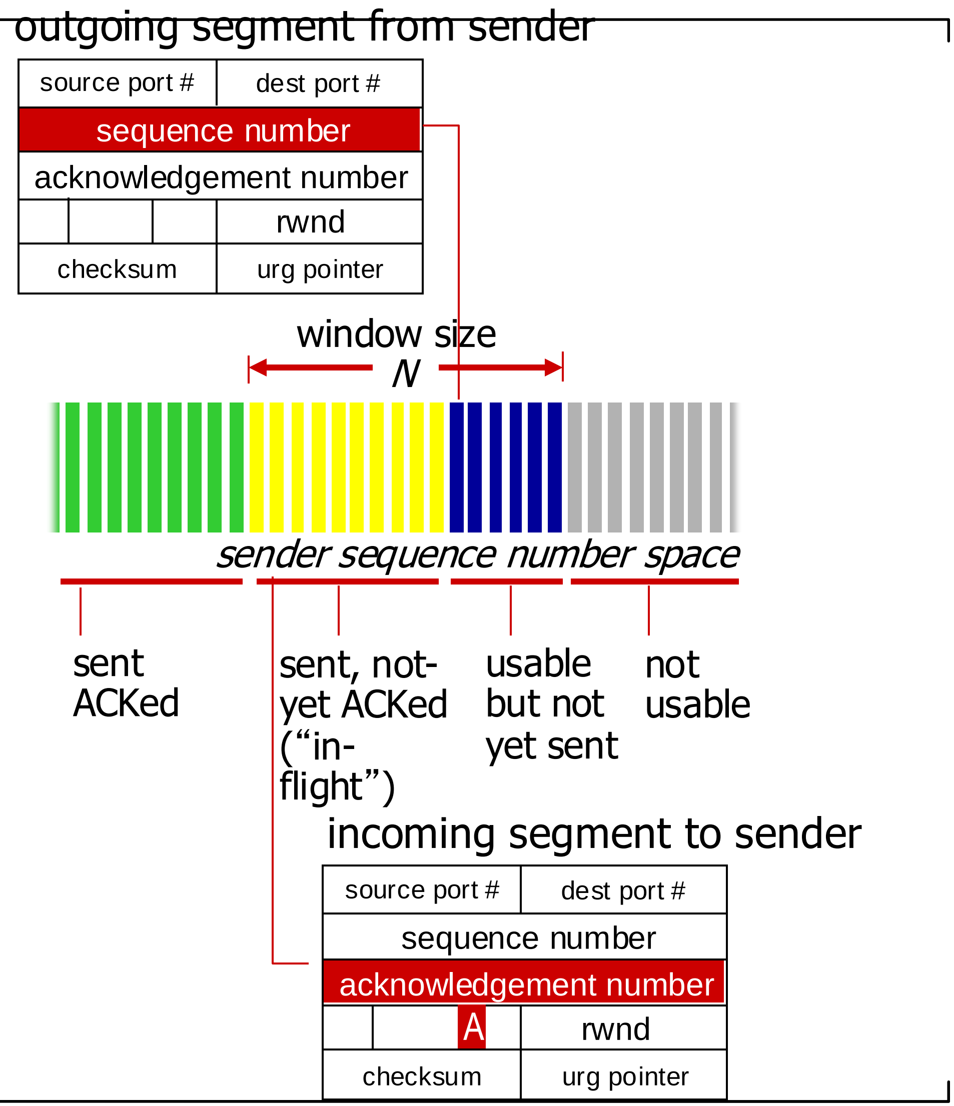
TCP header



32 bits
URG: urgent data (generally not used)
ACK: ACK # valid
PSH: push data now (generally not used)
RST, SYN, FIN: connection estab (setup, teardown commands)
Internet checksum (as in UDP)
source port #
dest port #
sequence number
acknowledgement number
head len
not used
U
A
P
R
S
F
receive window
checksum
Urg data pointer
options (variable length)
application data (variable length)
counting by bytes of data (not segments!)
bytes rcvr willing to accept

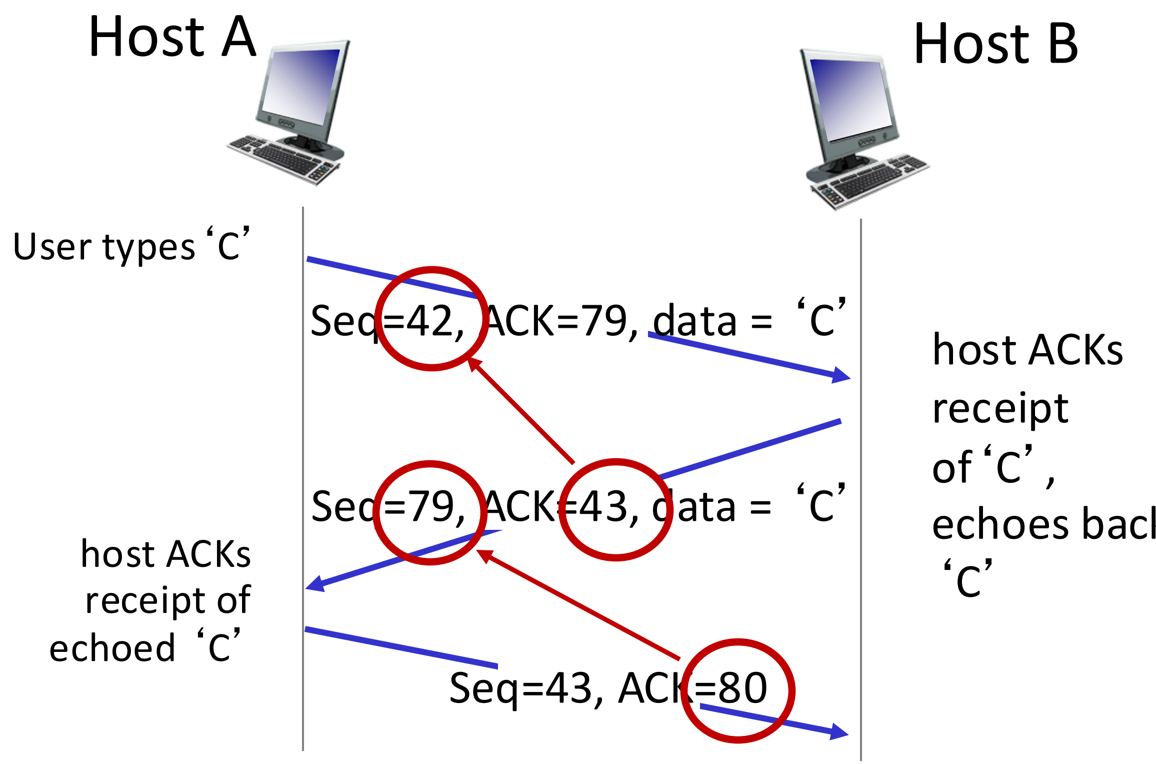
TCP seq. numbers, ACKs

- Sequence numbers:
  - byte stream "number" of first byte in segment's data
- Acknowledgements:
  - seq # of next byte expected from other side
  - cumulative ACK



TCP seq. numbers, ACKs

- Starting seq. no. 42 for client and 79 for server
- After TCP established, before data sent, client waits for byte 79 and server waits for byte 42
- Server replies with ack 43, seg 79 and echo back 'C'
- Acknowledgment is said to be piggybacked



simple telnet scenario

TCP reliable data transfer

TCP creates rdt service on top of IP's unreliable service

- sender/receiver agree to establish connection "handshake'
- pipelined segments for efficiency
- cumulative acks for acknowledgements
- single retransmission timer (for loss detection)
- checksums (for error detection)

retransmissions triggered by:

- timeout events
- duplicate acks

simplified TCP sender:

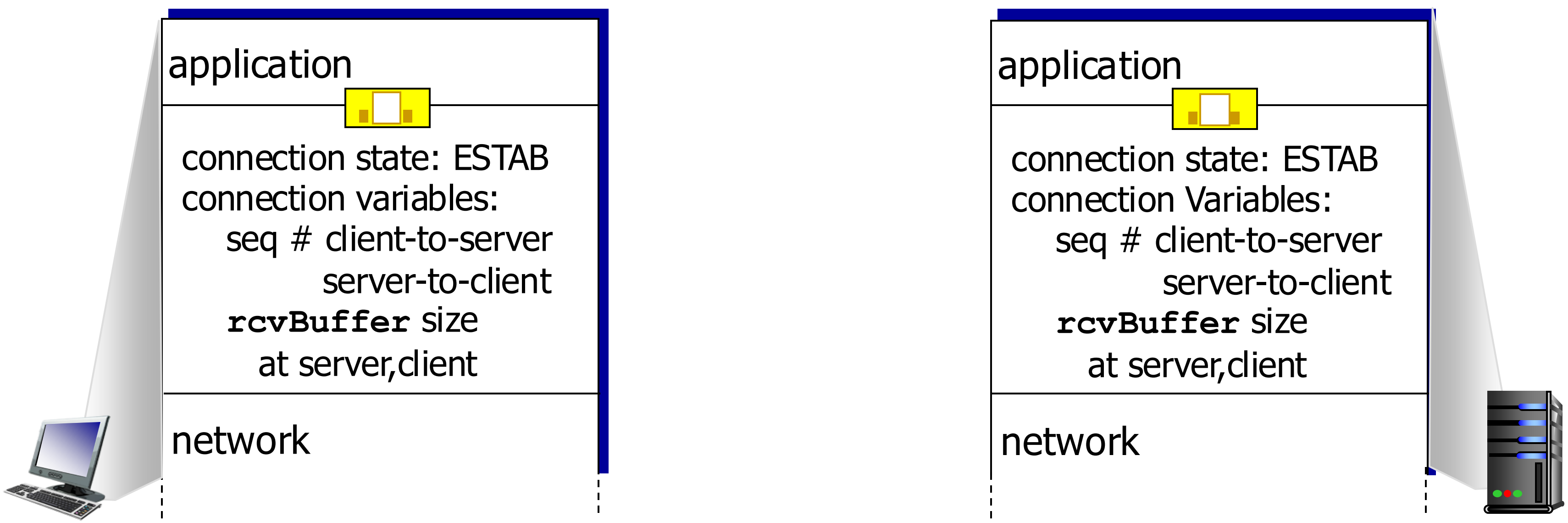
- ignore duplicate acks
- ignore flow control, congestion control

Connection Management

before exchanging data, sender/receiver "handshake":

agree to establish connection (each knowing the other willing to establish connection)

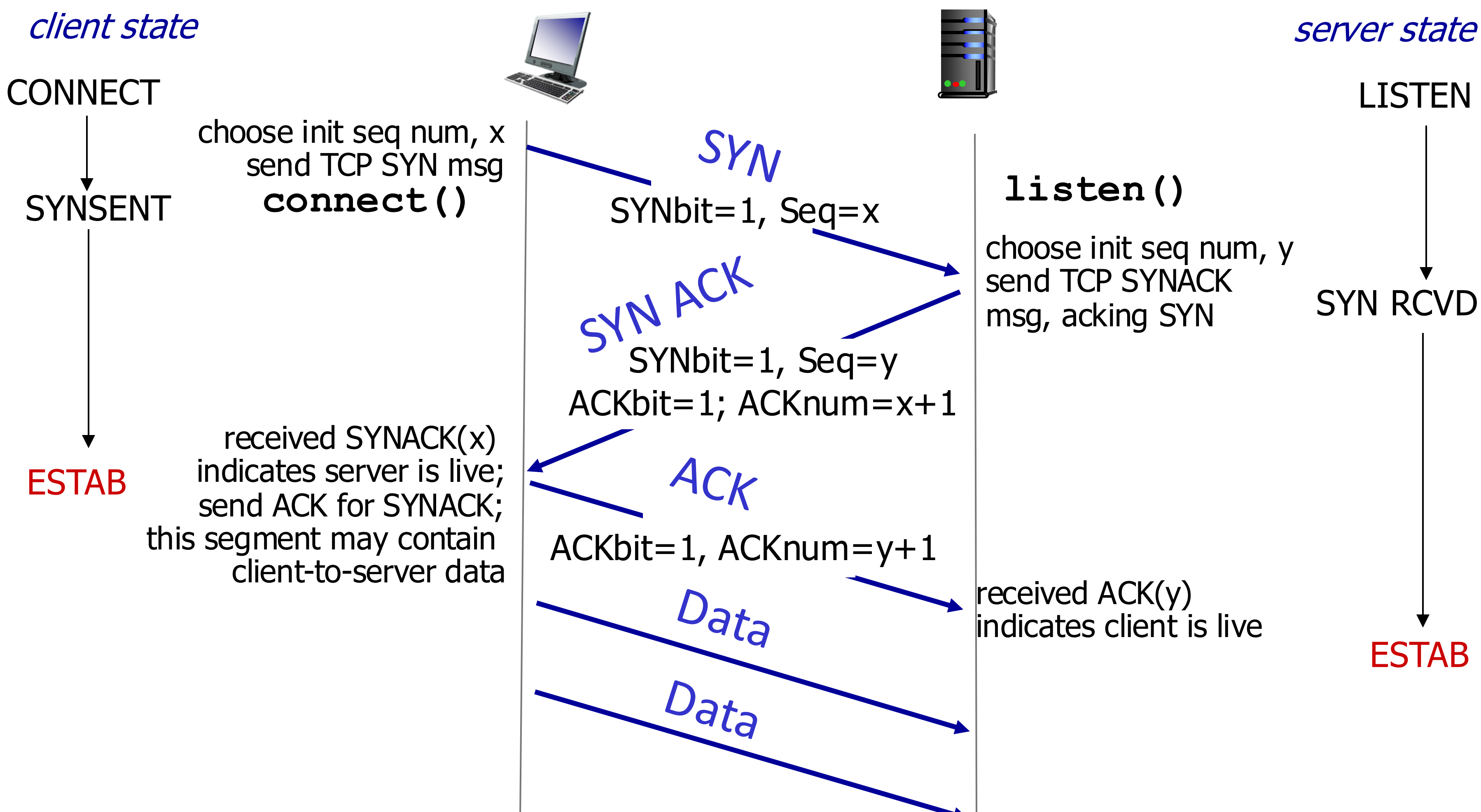
agree on connection parameters



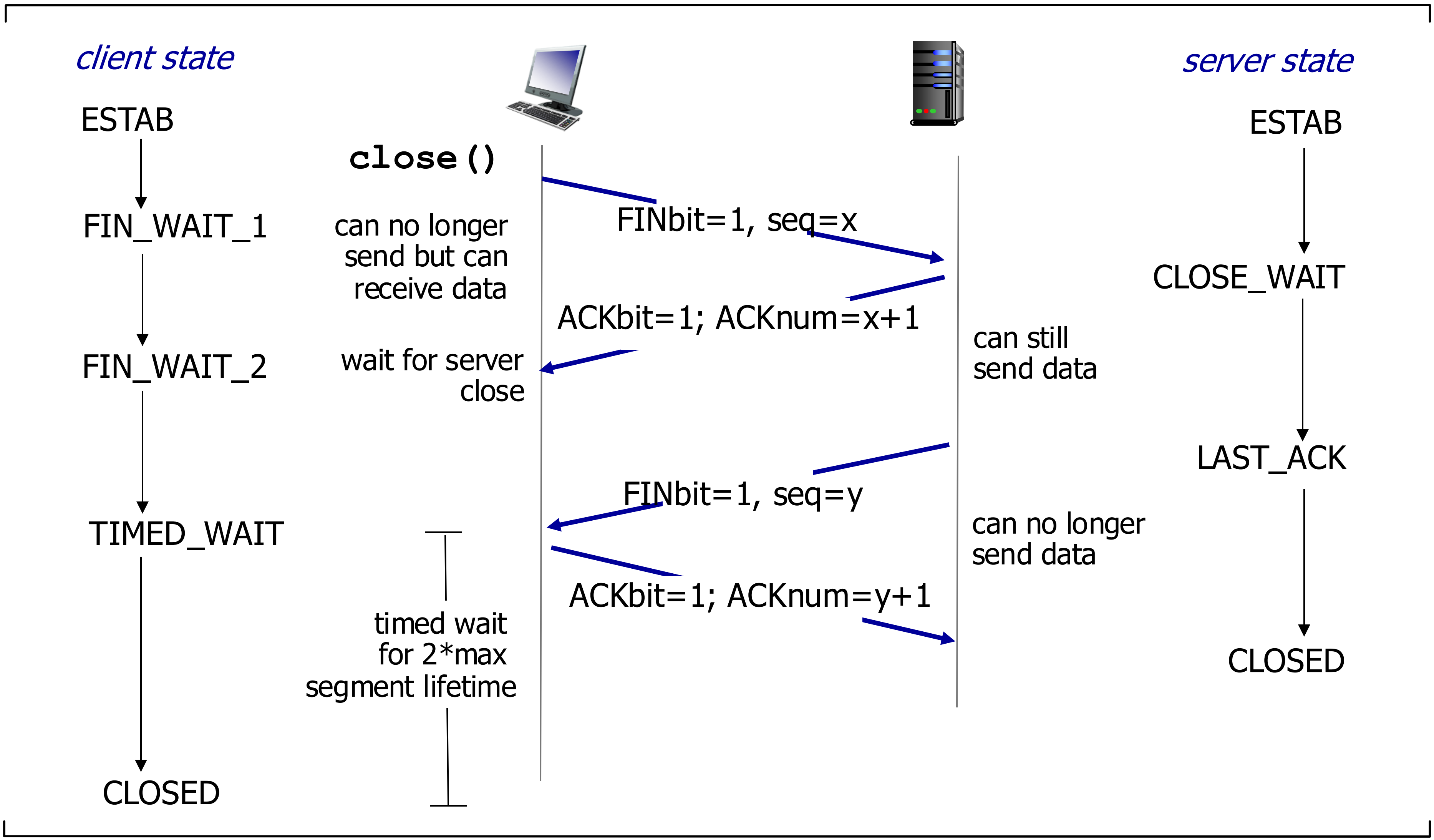

```
connect (sockfd,..)
```

```
listen (sockfd,n);
connfd = accept();
```

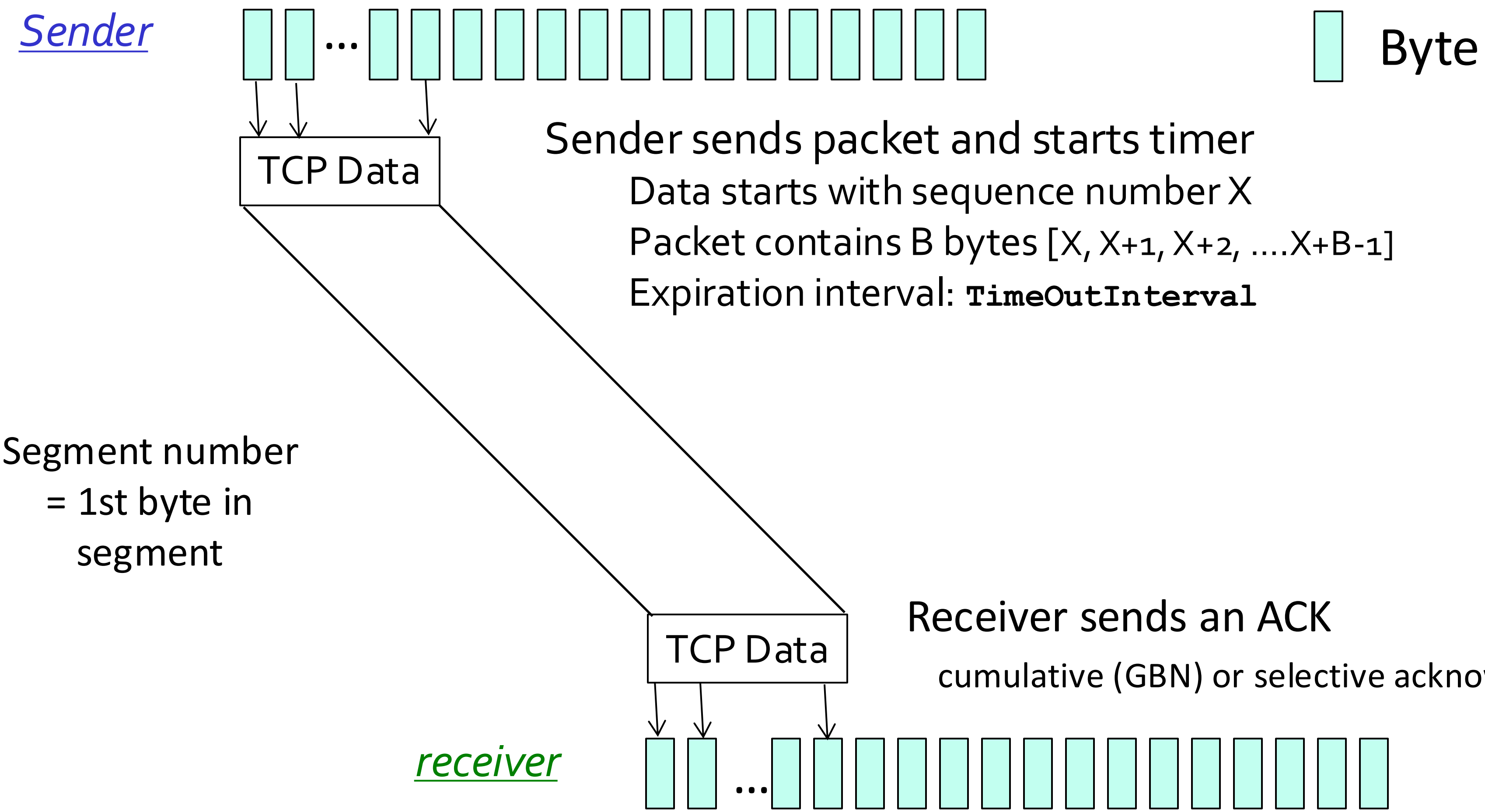
Establishing a TCP Connection: 3-way handshake



Closing a TCP connection



TCP Segments



Sender sends packet and starts timer

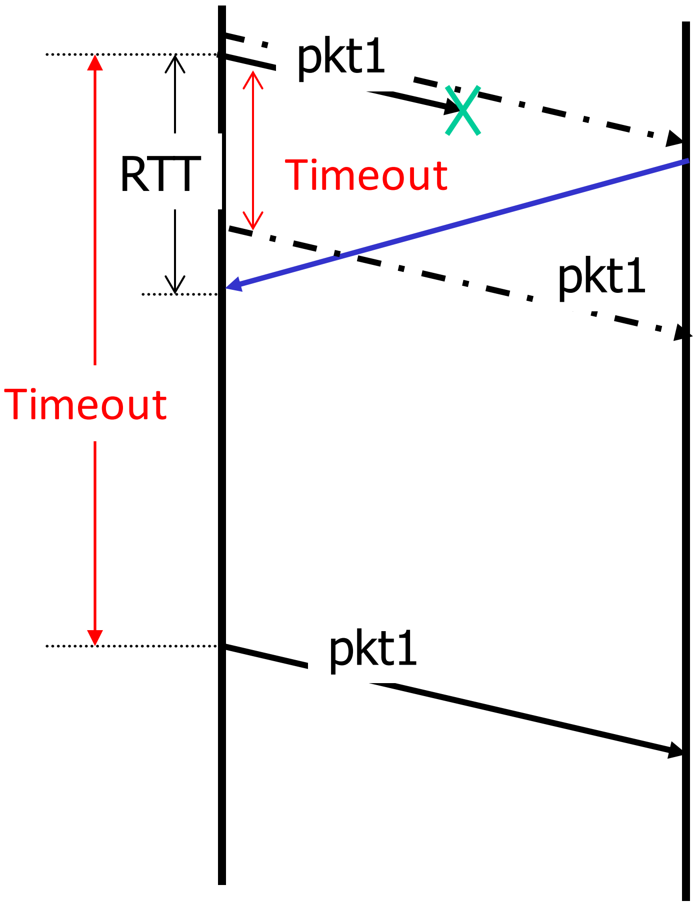
- Data starts with sequence number X
- Packet contains B bytes [X, X+1, X+2, ...X+B-1]
- Expiration interval: **`TimeOutInterval`**

Segment number
= 1st byte in segment

Receiver sends an ACK

- cumulative (GBN) or selective acknowledges

TCP retransmission timer



Timeout too long – inefficient

Timeout too short – duplicate packets

Make timeout proportional to RTT

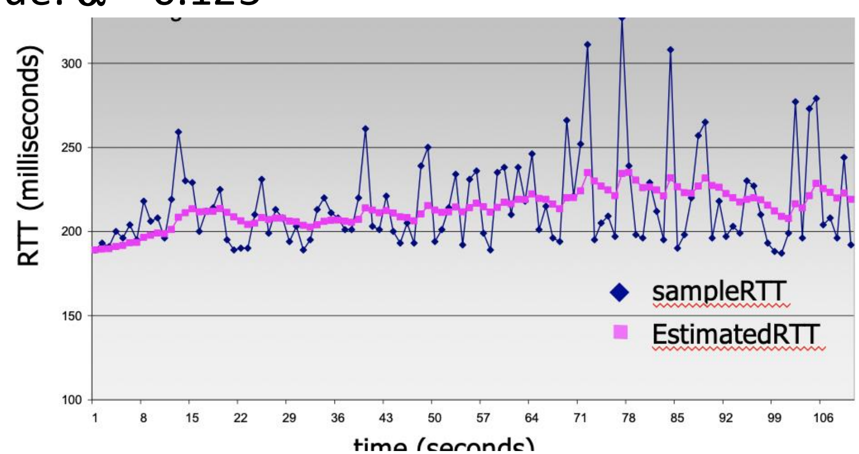
$$\text{EstimatedRTT} = (1-\alpha)*\text{EstimatedRTT} + \alpha*\text{SampleRTT}$$

(exponential weighted moving average)

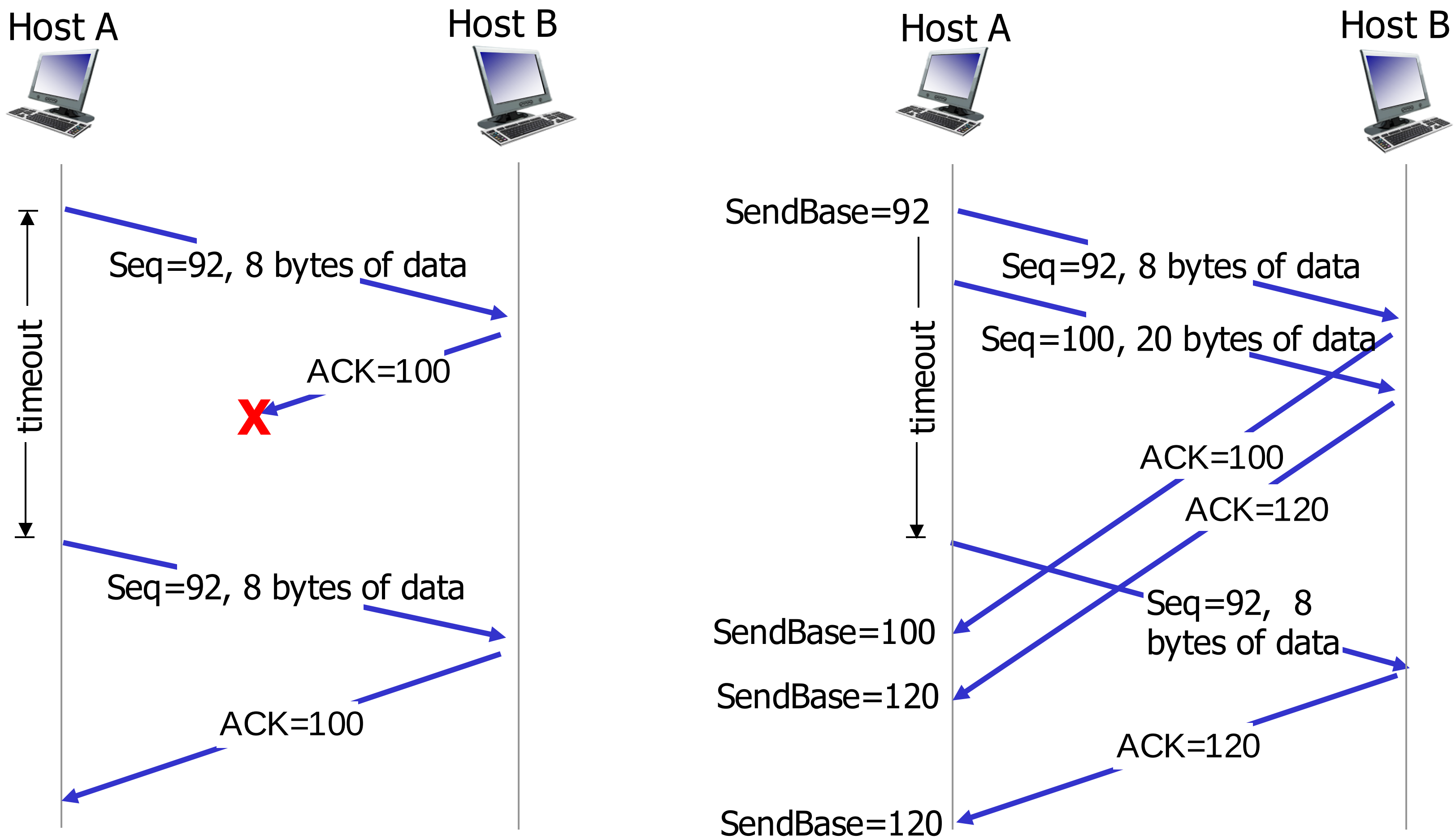
SampleRTT = t segment sent – t segment acknowledged

This time keeps changing with network conditions

- influence of past sample decreases exponentially fast
- typical value: α = 0.125



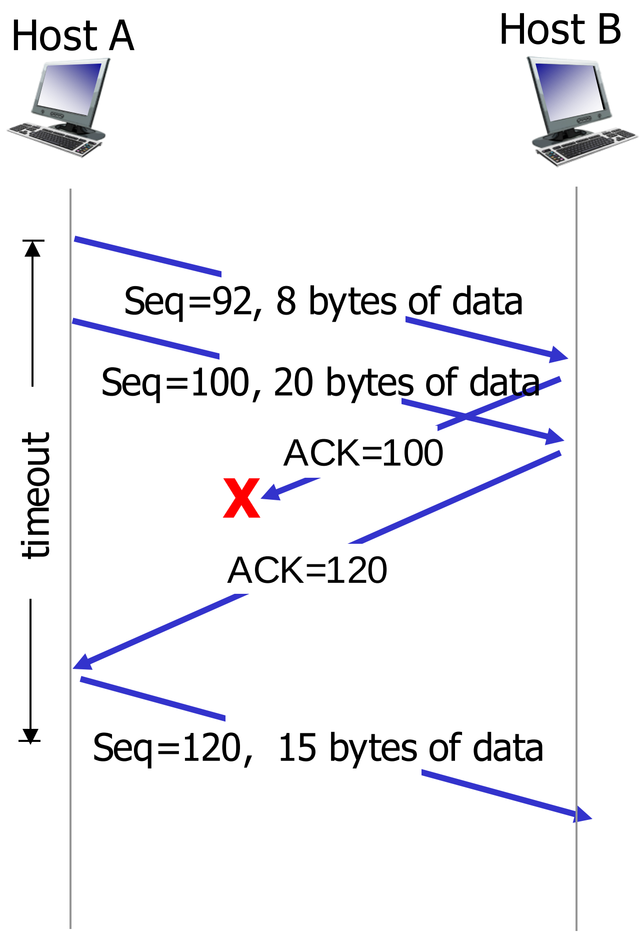
TCP: retransmission scenarios



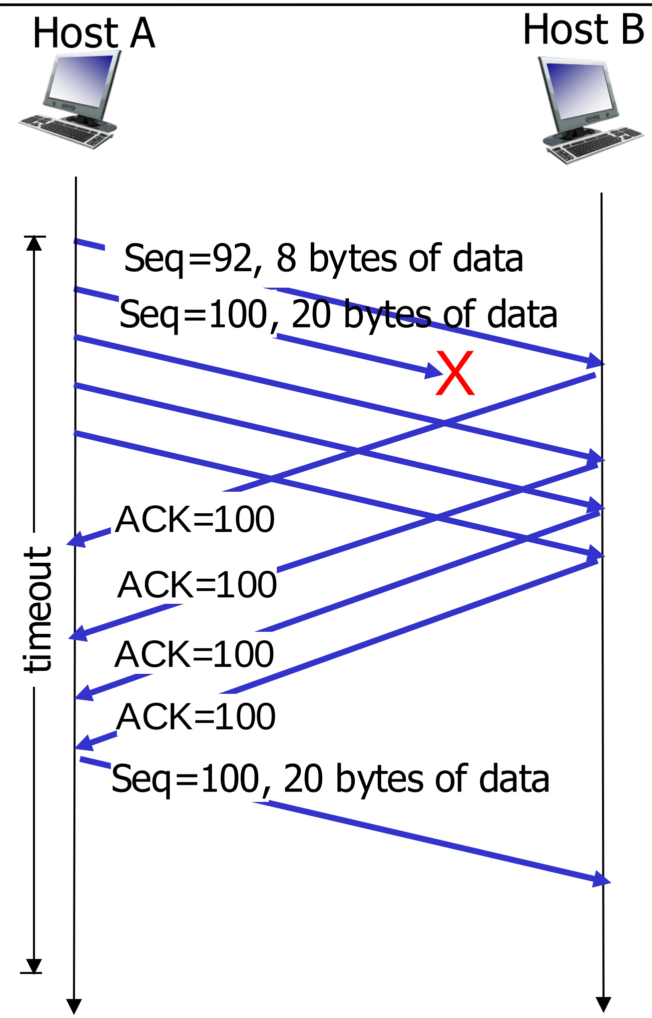
lost ACK scenario

premature timeout

TCP: retransmission scenarios



TCP fast retransmit

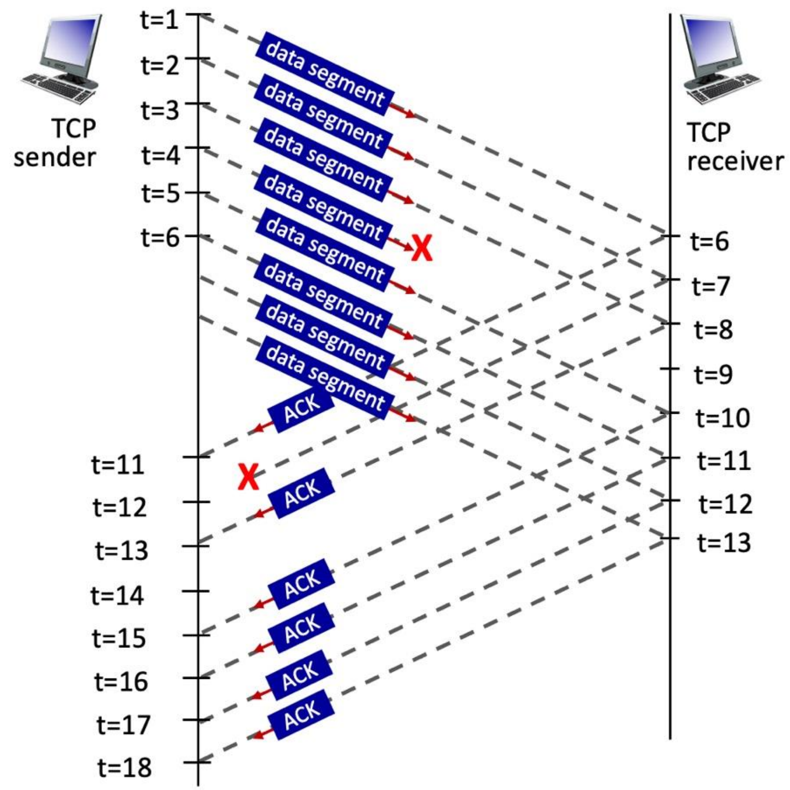


fast retransmit after sender receipt of triple duplicate ACK

TCP example

TCP sender sends 8 TCP segments at t = 1, 2, 3, 4, 5, 6, 7, 8. Suppose the initial value of the sequence number is 0 and every segment sent to the receiver each contains 100 bytes. The delay between the sender and receiver is 5 time units, and so the first segment arrives at the receiver at t = 6.

- Sender's sequence numbers: 0, 100, 200, 300,….
- Receiver's acknowledgement numbers: 100, 200, 300, 300,..



TCP ACK generation [RFC 1122, RFC 2581]

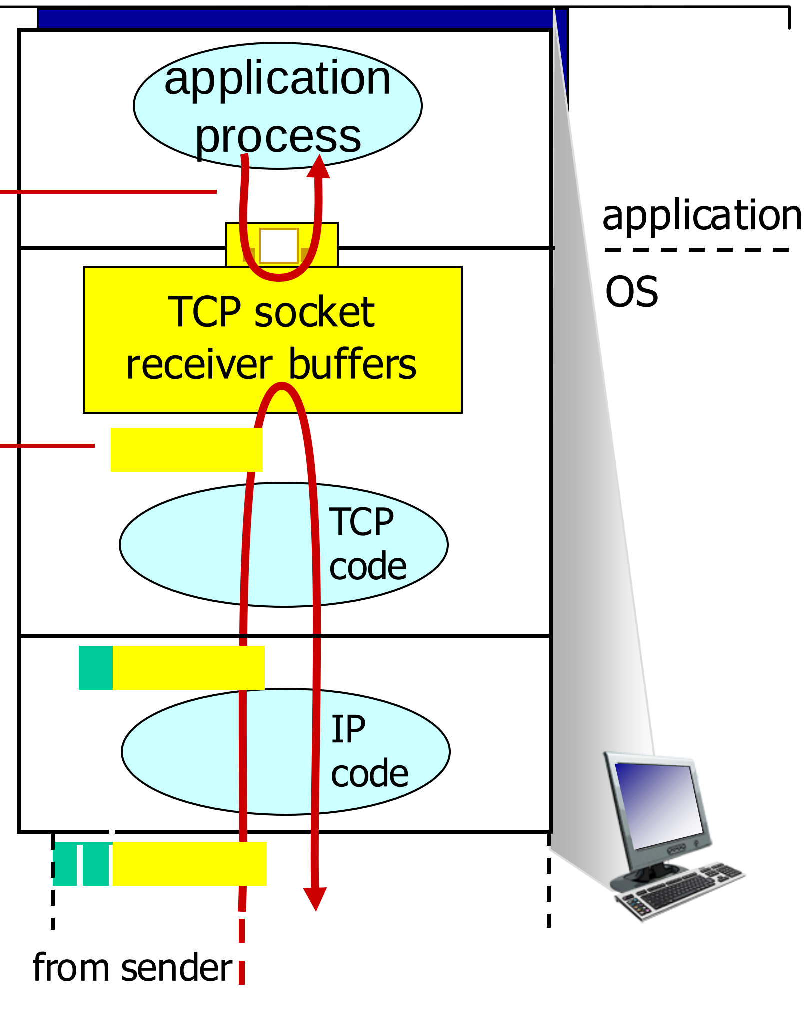
event at receiver	*TCP receiver action*
arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
arrival of in-order segment with expected seq #. One other segment has ACK pending	immediately send single cumulative ACK, ACKing both in-order segments
arrival of out-of-order segment higher-than-expect seq. # . Gap detected	immediately send *duplicate ACK,* indicating seq. # of next expected byte
arrival of segment that partially or completely fills gap	immediate send ACK, provided that segment starts at lower end of gap

TCP flow control

"no one can drink from a firehose"

application may remove data from TCP socket buffers ….

… slower than TCP receiver is delivering (sender is sending)



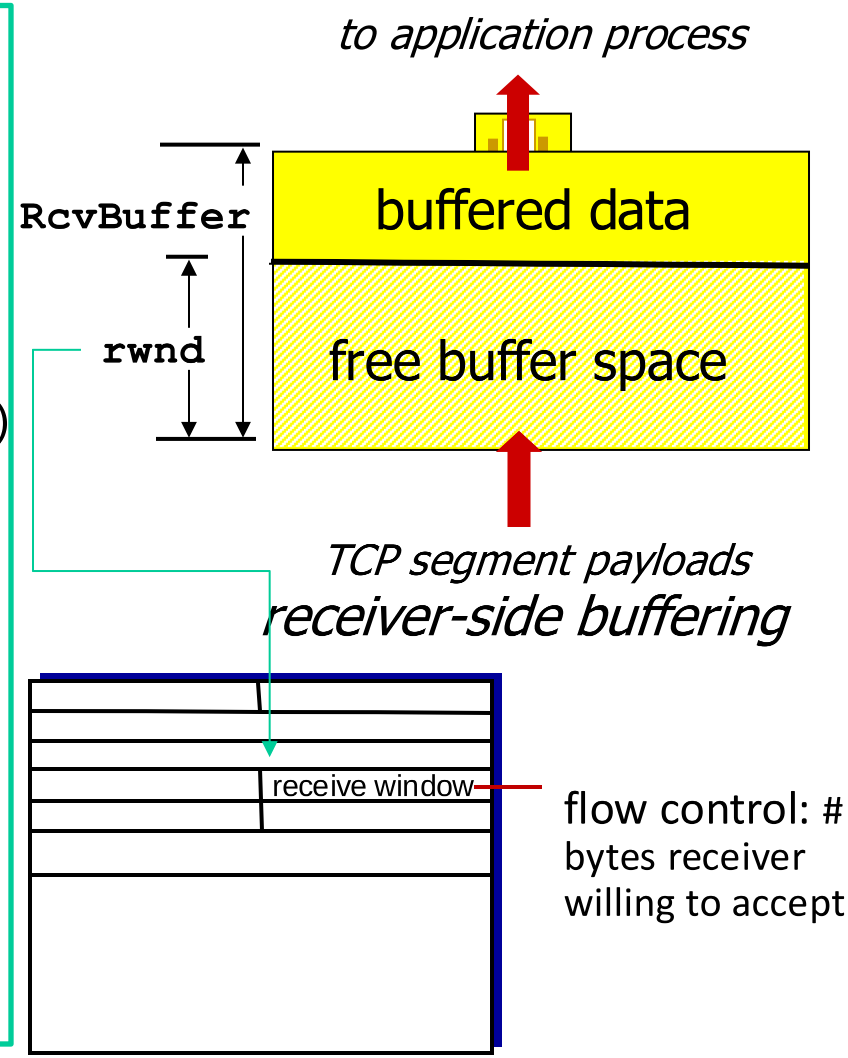
receiver protocol stack

flow control

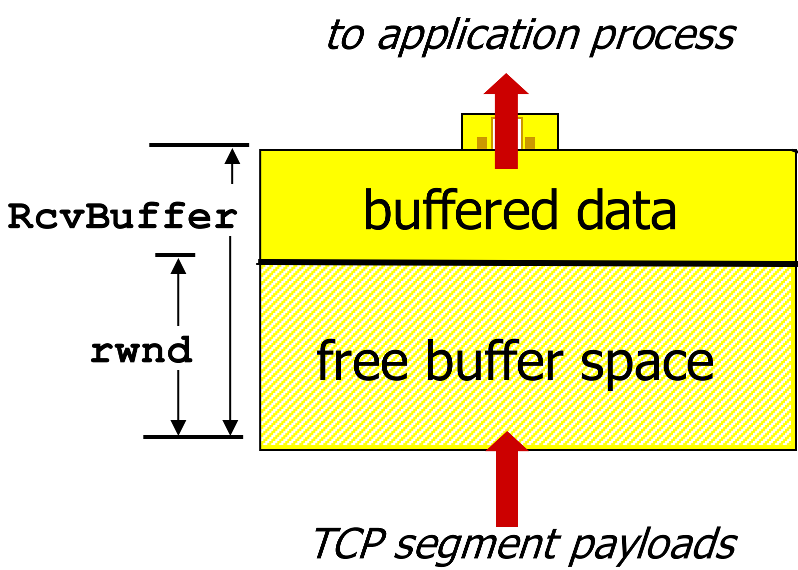
receiver controls sender, so sender won't overflow receiver's buffer by transmitting too much, too fast

Receiver window - rwnd

- receiver "advertises" free buffer space by including **`rwnd`** (*receiver window*) value in TCP header of receiver-to-sender segments
  - **`RcvBuffer`** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **`RcvBuffer`**
- sender limits amount of unacked ("in-flight") data to receiver's **`rwnd`** value
- guarantees receive buffer will not overflow

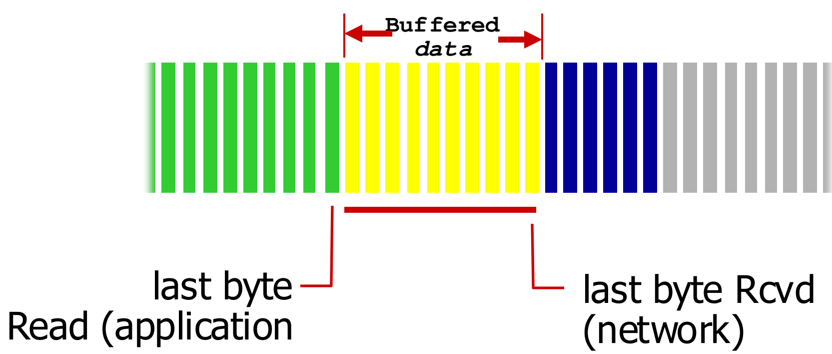


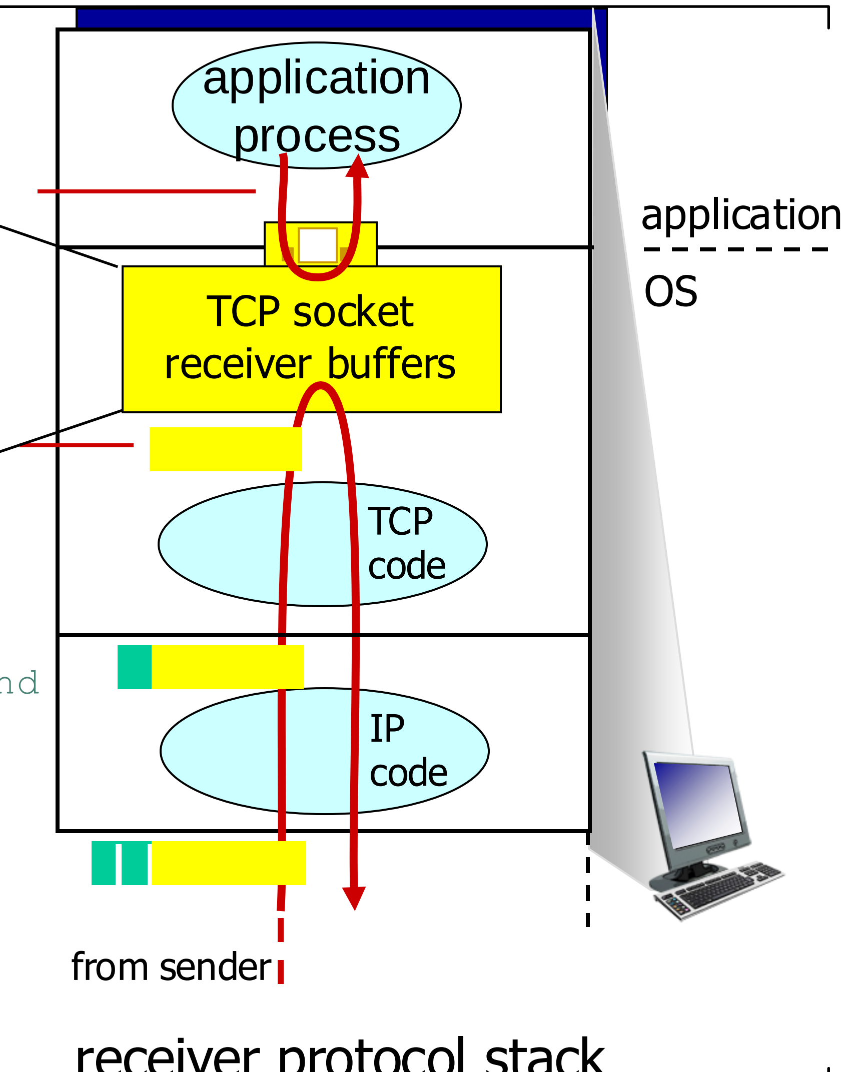
TCP flow control



`rwnd=RcvBuffer-[LastByteRcvd-LastByteRead]`

`Sender keeps: LastByteSent - LastByteAcked ≤ rwnd`





Summary

Today:

- TCP reliable data transfer
- TCP connection
- TCP segments
- TCP sender and receiver ACK generation
- TCP flow control

Canvas discussion:

- Reflection
- Exit ticket

Next time:

- read 3.6 and 3.7 of K&R (TCP congestion control)
- follow on Canvas! Material and announcements

Any questions?