TCP Internals (cont'd)

Pre-Class Discussion

- Recap: TCP lifecycle
  - 3-way handshake
  - Sliding window-based transport
  - 4-way termination
- **Edge case testing of TCP states**
  - What edge cases can we test in an attempt to uncover invalid or unexpected TCP states?
- Discuss with the person next to you
- Share your idea later

Announcements

- HW4 released last Saturday (due May 1)
- Last call: early teaching feedback for Francis
- ***Reliable Communication Competition (Spring 2026)***
  - MP3: reliable file transfer
    - TCP-like variant & open-ended variant
  - Leaderboard displays team rankings
    - Edit Google Form response and anonymize your team name
    - Do not overfit — final results will be based on different tests
  - 🏆 Top 3 teams in each variant receive
    - Gift cards
    - Bragging rights

Learning Objectives

- **TCP state machine**
- Retransmission timeout
- Flow control

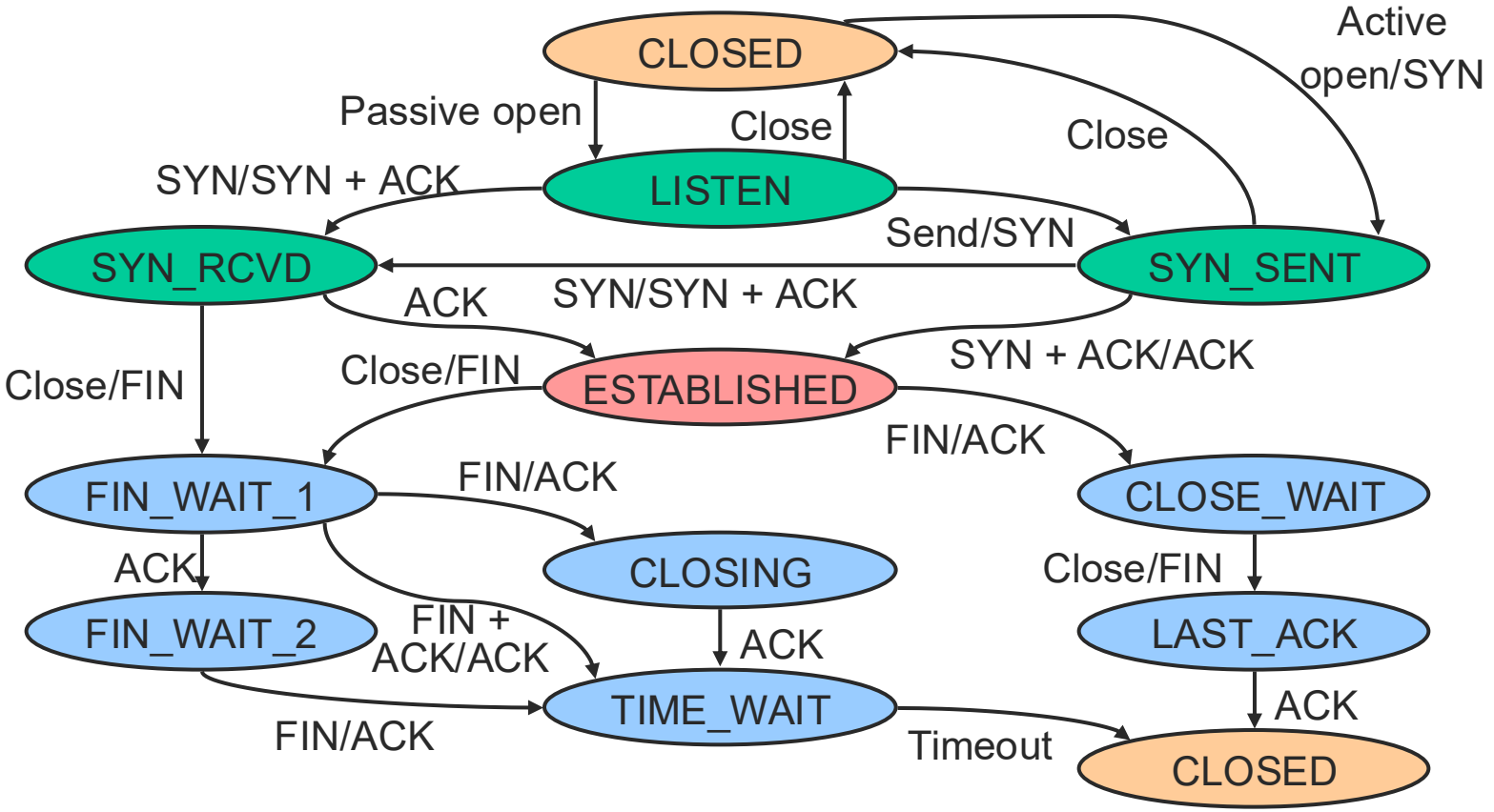
TCP State Transitions

- Under normal conditions
  - Ordered connection establishment
  - Graceful connection termination
- Edge cases?
  - Simultaneous close
  - Last ACK lost
  - ...
- Finite state machine (FSM) ensures reliability
  - Provides formal, verifiable specification

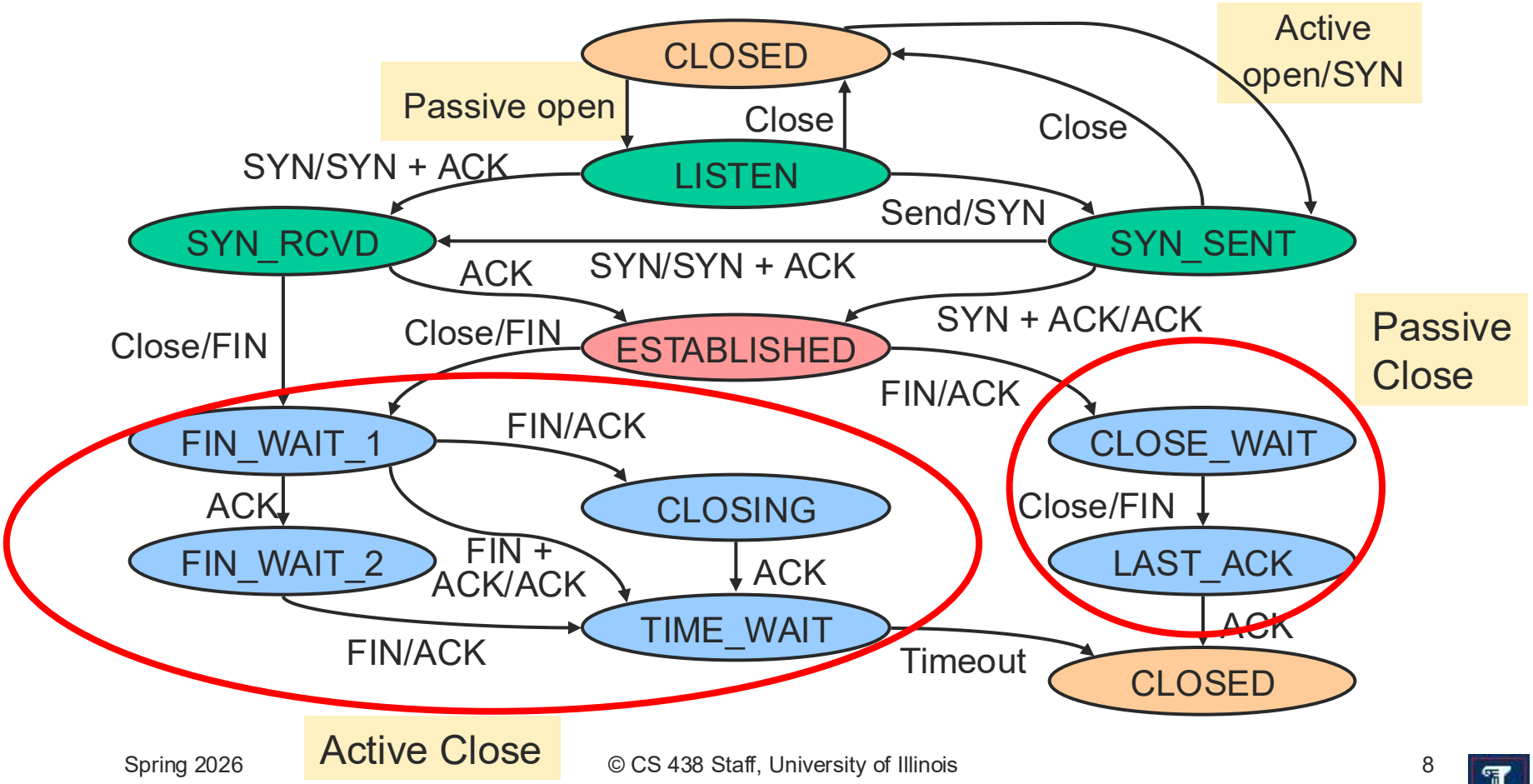
TCP State Descriptions

CLOSED	Disconnected
LISTEN	Waiting for incoming connection
SYN_RCVD	Connection request received
SYN_SENT	Connection request sent
ESTABLISHED	Connection ready for data transport
CLOSE_WAIT	Connection closed by peer
LAST_ACK	Connection closed by peer, closed locally, await ACK
FIN_WAIT_1	Connection closed locally
FIN_WAIT_2	Connection closed locally and ACK'd
CLOSING	Connection closed by both sides simultaneously
TIME_WAIT	Wait for network to discard related packets

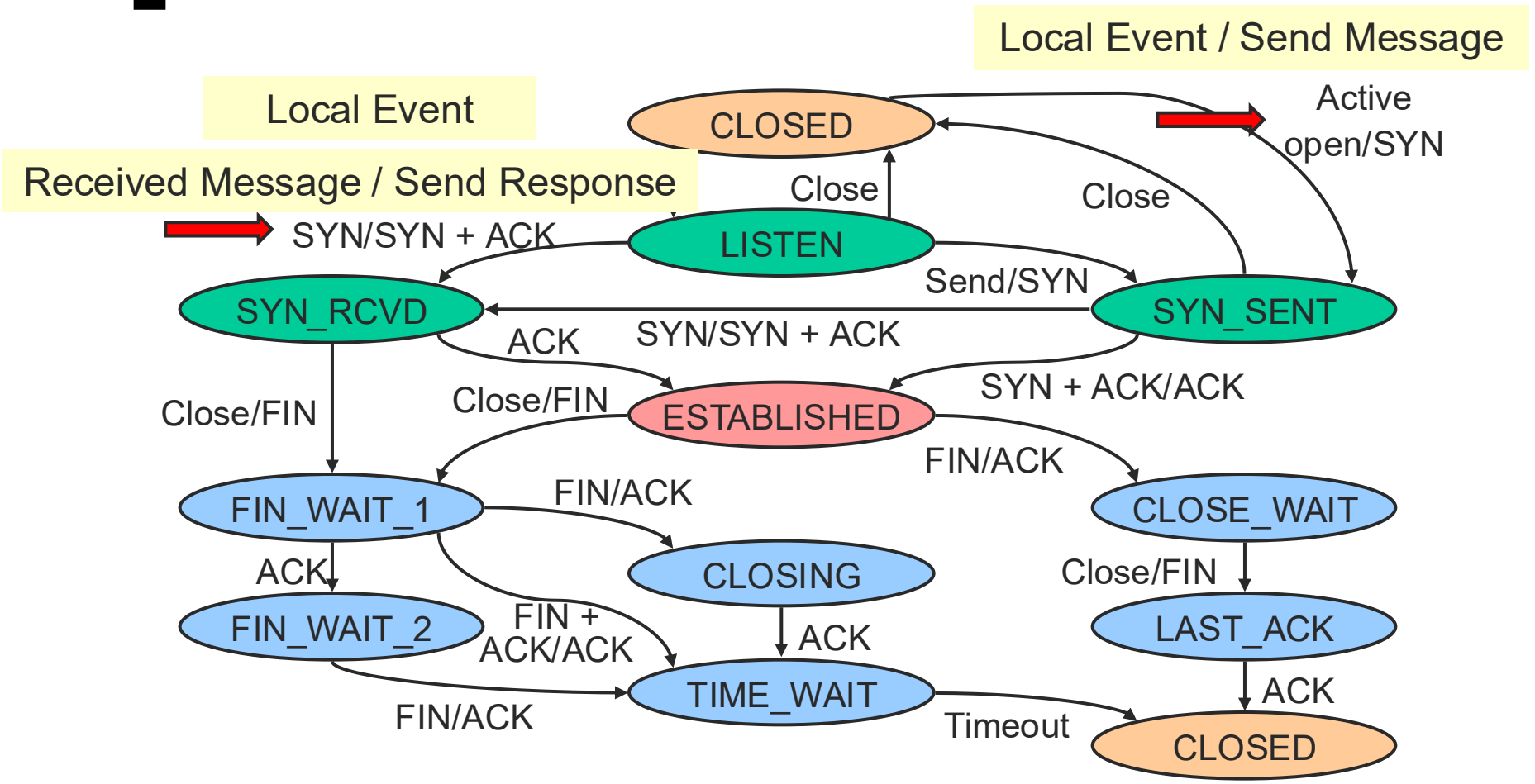
TCP State Machine



TCP State Machine

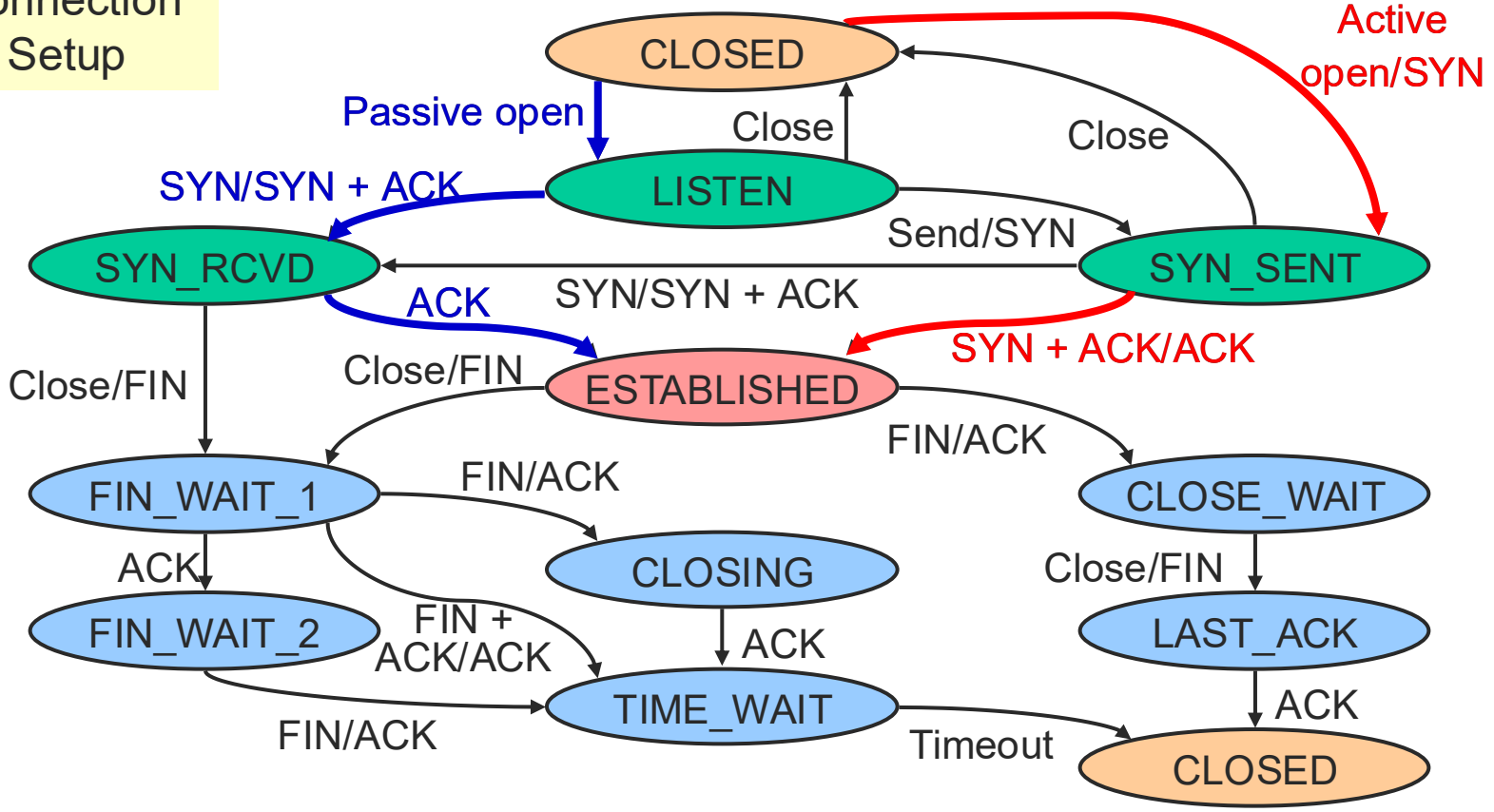


TCP State Machine



TCP State Machine

Connection Setup

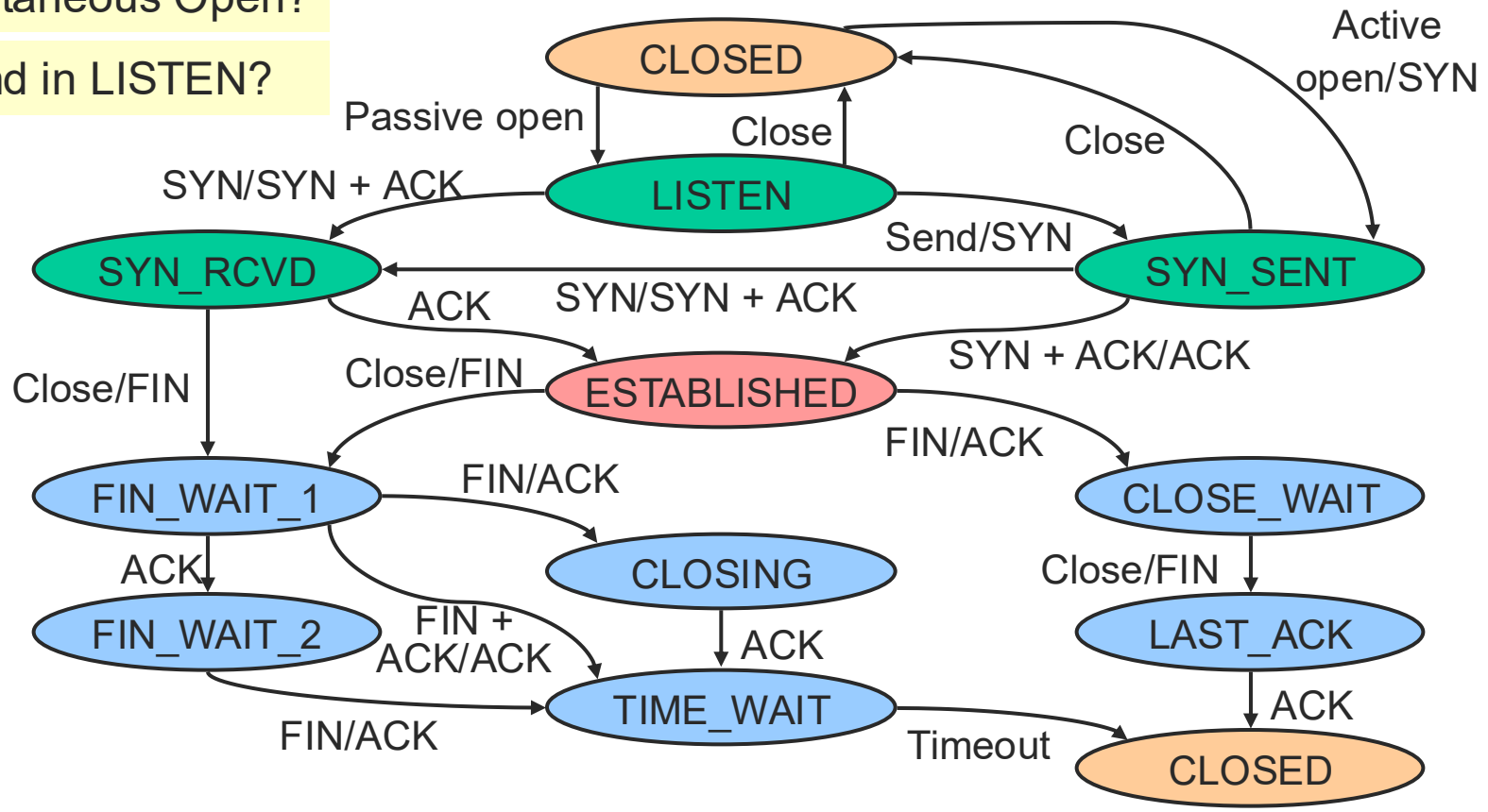


A POSIX View of Connection Setup

TCP State Machine

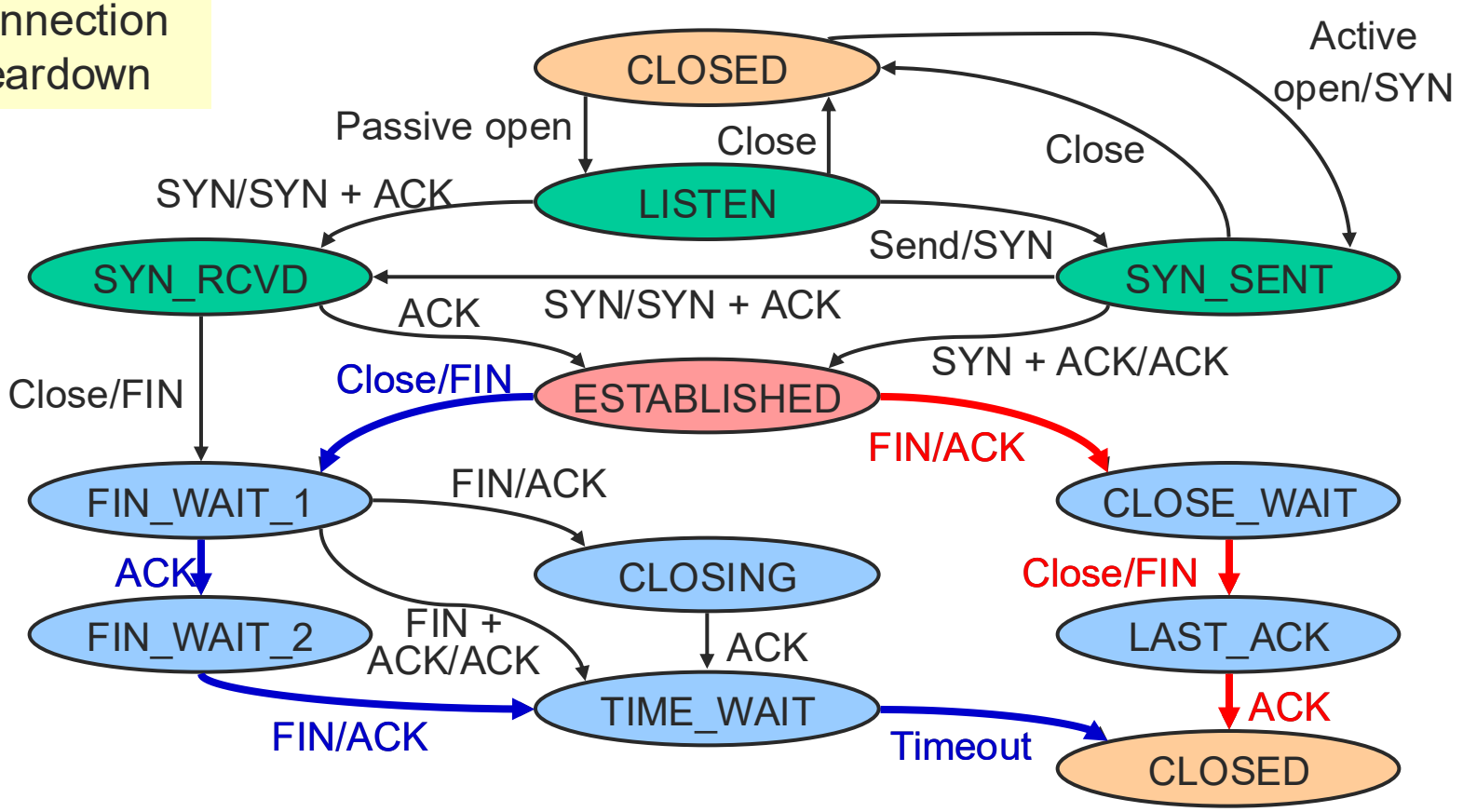
Simultaneous Open?

Send in LISTEN?

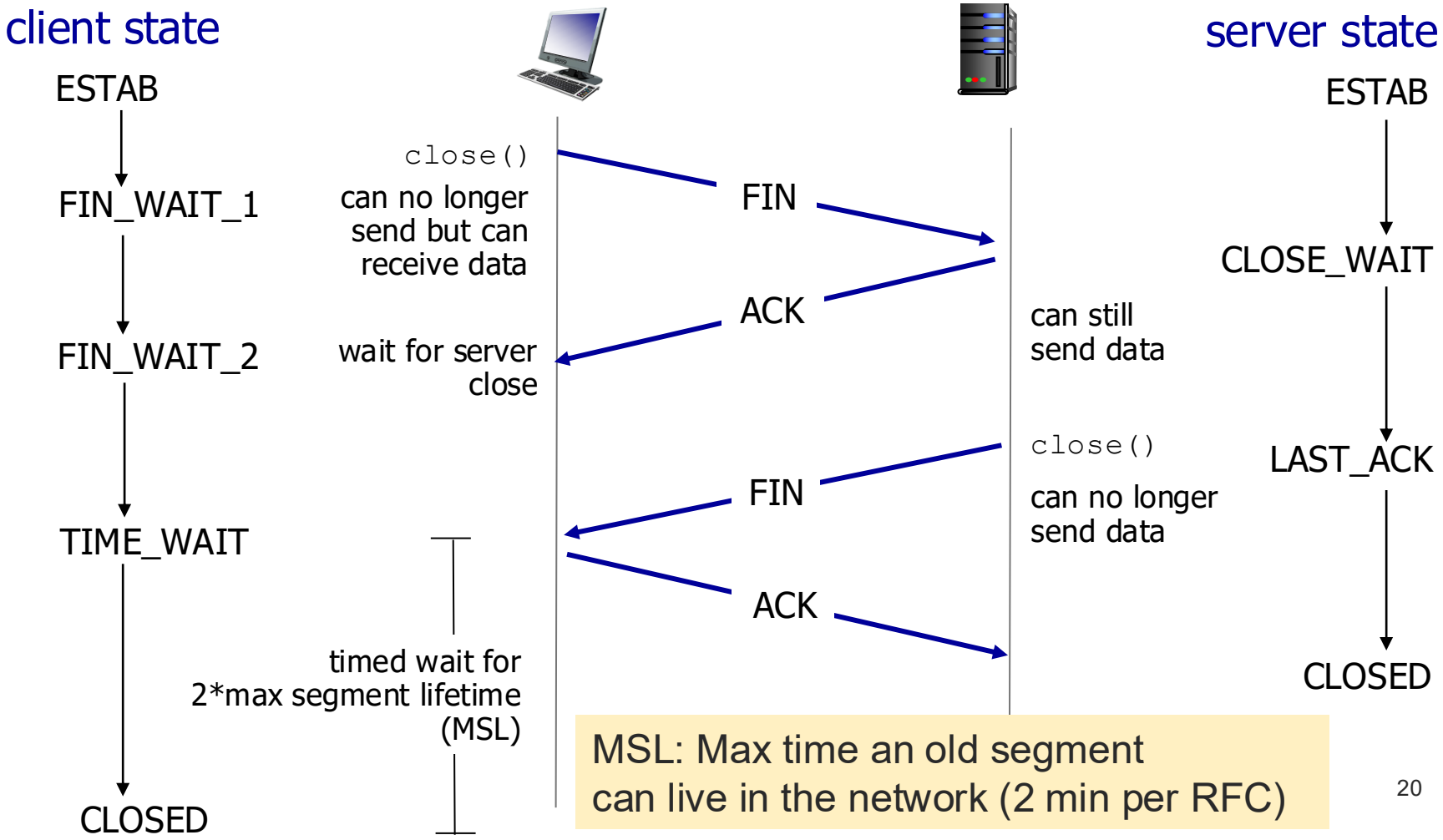


TCP State Machine

Connection Teardown



A POSIX View of Connection Teardown

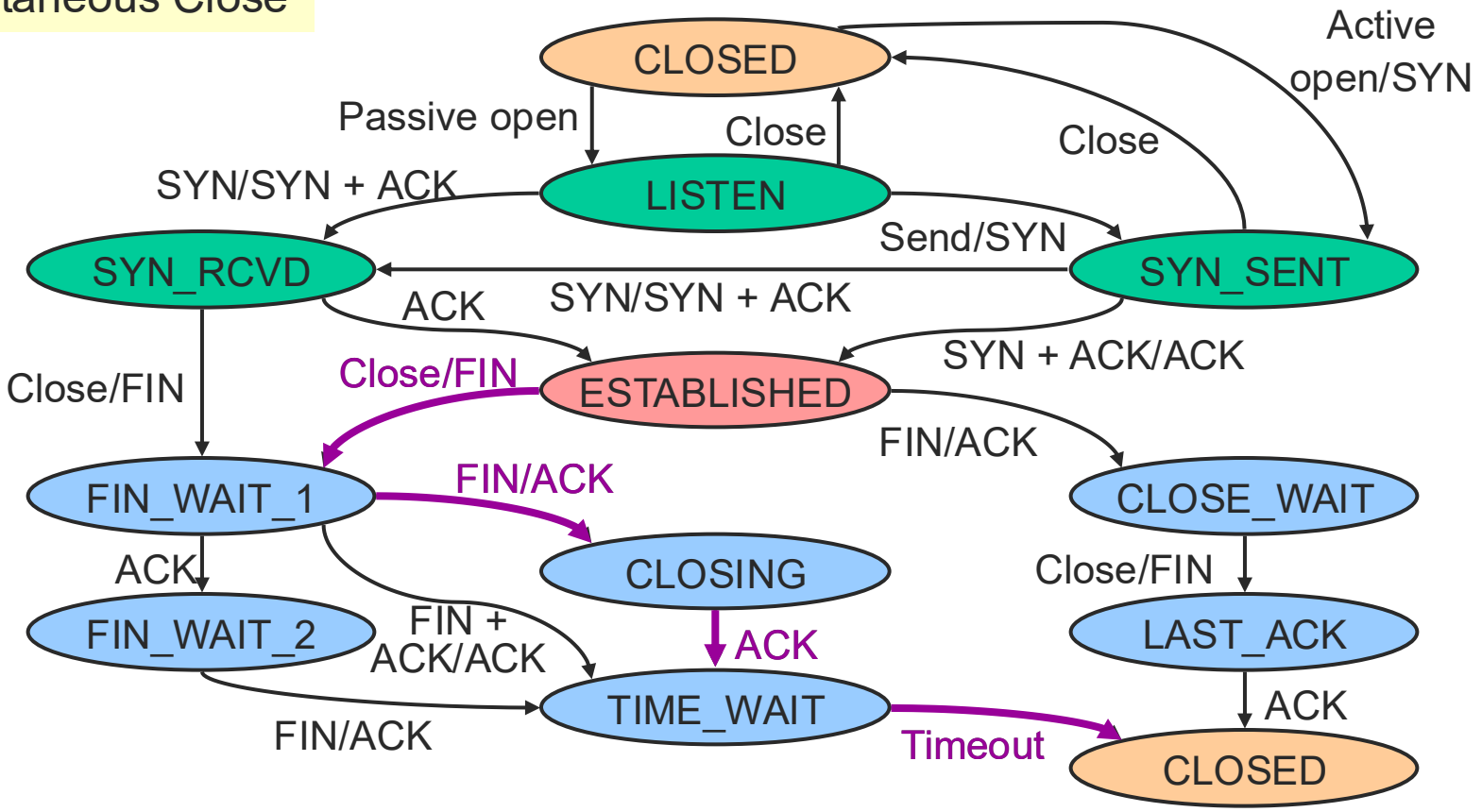


TCP TIME_WAIT State

- TIME_WAIT state
  - Connection remains in this state for 2x the maximum segment lifetime (MSL)
- What purposes does the TIME_WAIT state serve?
  - Last ACK from client might be lost
    - thus two MSL: for ACK and retransmitted FIN
  - Reject segments from an old connection
    - otherwise, may be accidentally accepted by a new connection

TCP State Machine

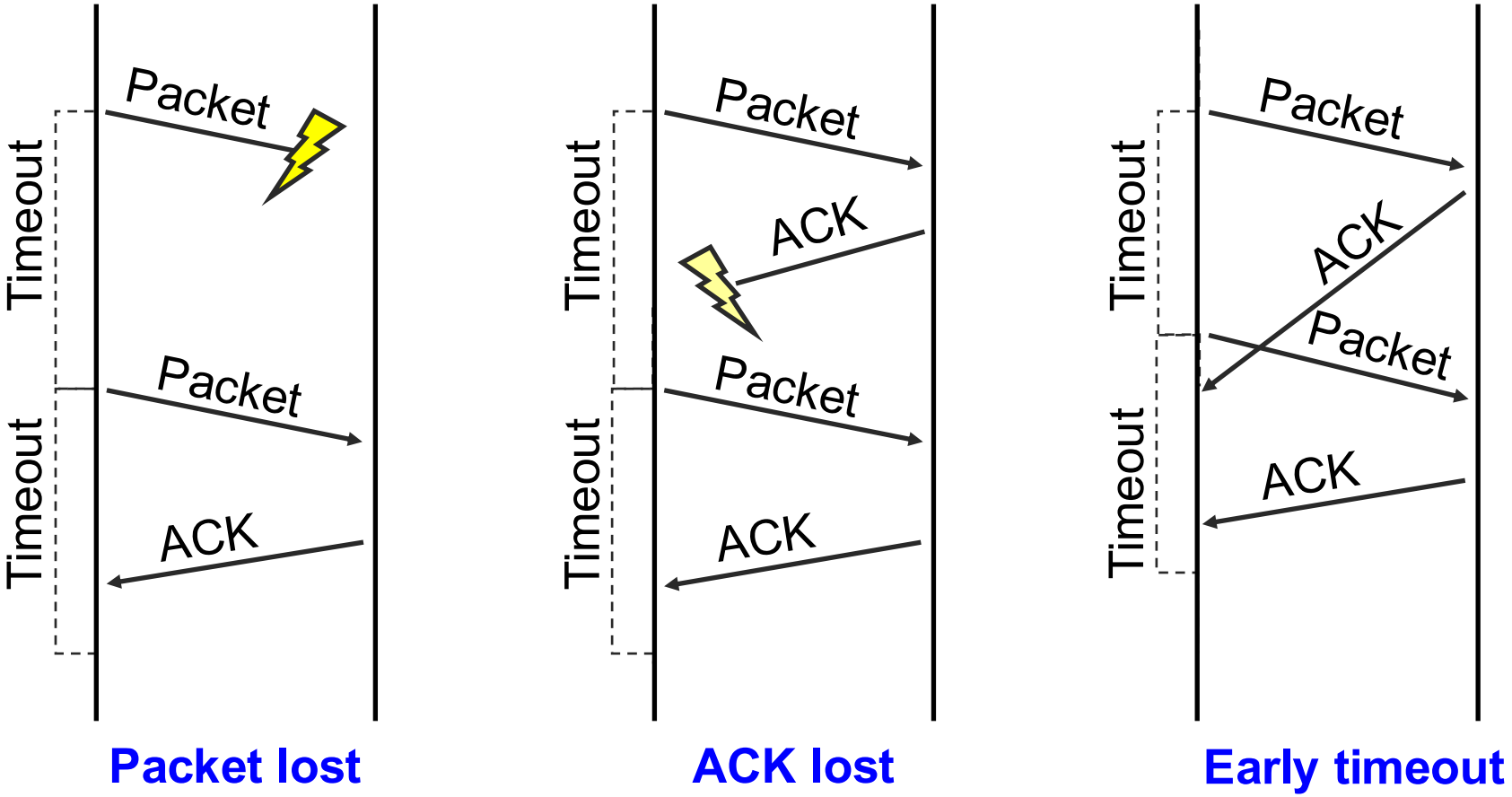
Simultaneous Close



Learning Objectives

- TCP state machine
- **Retransmission timeout**
- Flow control

Reasons for Retransmissions



TCP Retransmission Timeout

- How should TCP sender set its retransmission timeout (RTO)?
  - Too short?
    - Premature timeout → wasted retransmissions
  - Too long?
    - Slow reaction to packet/ACK loss → excessive delays
  - Only need to be longer than the next RTT?
    - We don't know when the ACK is supposed to arrive
    - RTT/network is stochastic!

TCP Retransmission Timeout

Algorithm sketch

1. Estimate RTT
   - *SampleRTT*: from packet transmission until ACK receipt
     - Want smoother estimated RTT
   - *EstimatedRTT*: moving average of *SampleRTT* measurements
     - Not just current one
2. Choose RTO
   - Such that after RTO, ACK is *unlikely* to arrive
   - Prefer a small RTO if possible

TCP Adaptive Retransmission Algorithm – Original

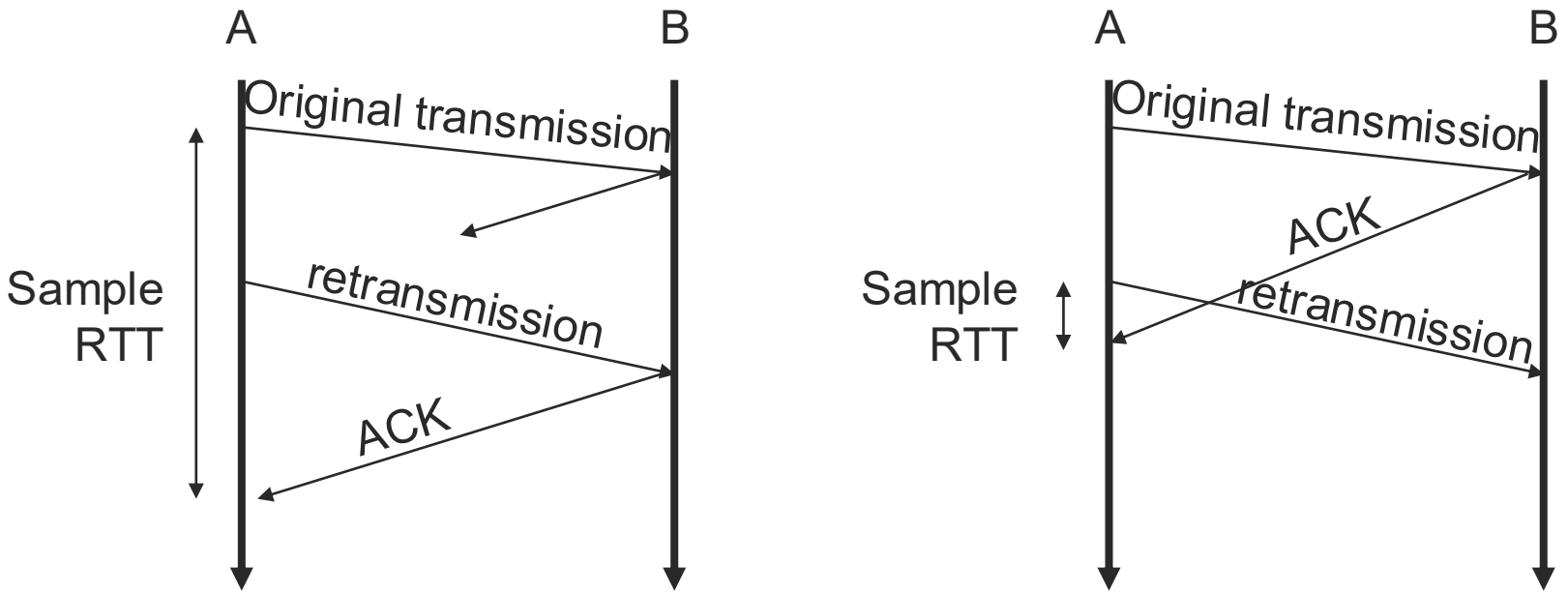
1. Estimate RTT with EWMA (exponentially weighted moving average)
   - $EstimatedRTT = \alpha \times SampleRTT + (1-\alpha) \times EstimatedRTT$
     - E.g., $\alpha = 0.1$ to 0.2
   - Influence of past samples decreases exponentially fast

2. Set $RTO = 2 \times EstimatedRTT$
   - To allow variations (since RTT is stochastic)

- Problems?

TCP Adaptive Retransmission Algorithm – Original

- Problem 1: Ambiguous *SampleRTT* after retransmission
  - Was it in response to first, second, ... transmission?

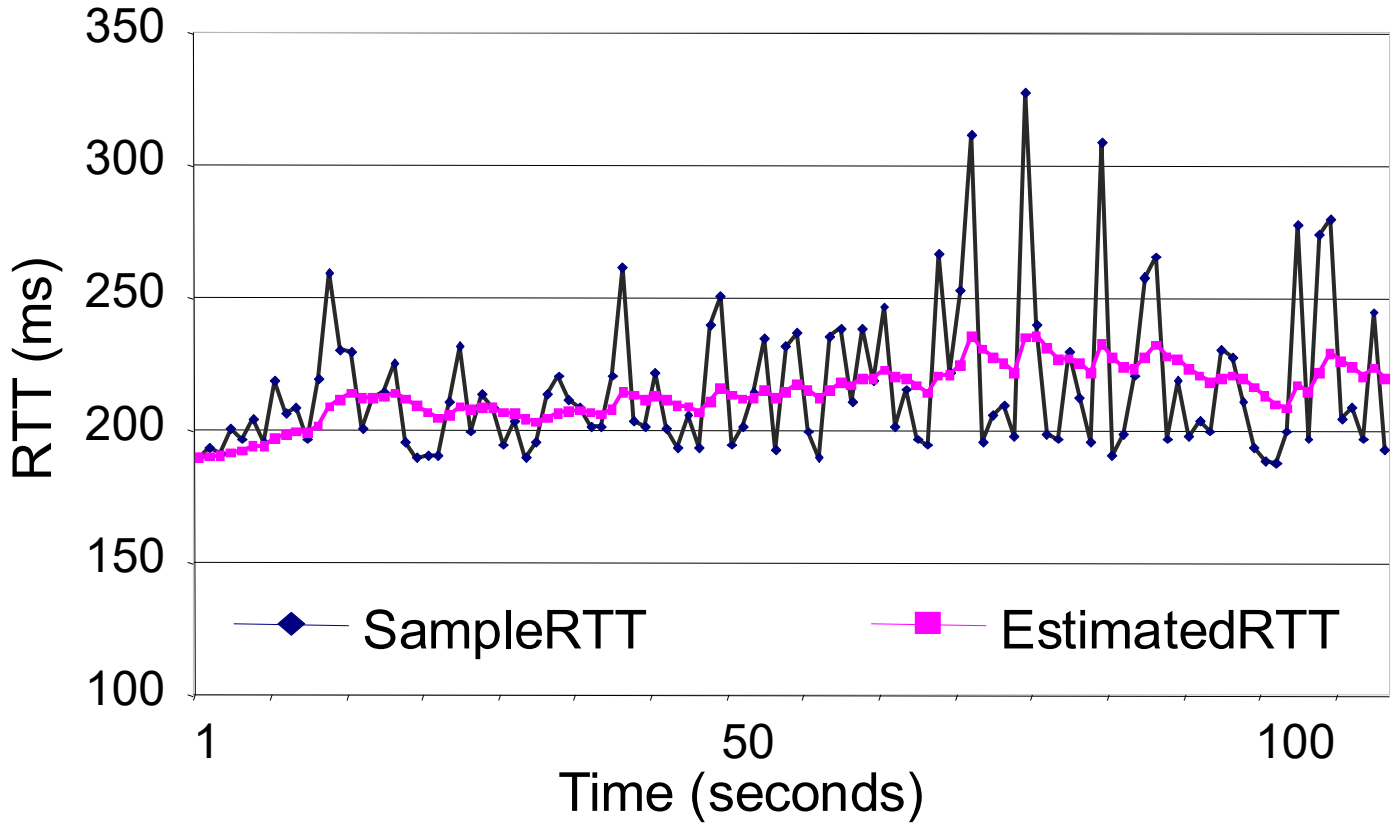


TCP Adaptive Retransmission Algorithm – Karn-Partridge

- Exclude retransmitted packets from RTT estimate
  - To avoid under/over-estimation
- For each retransmission, double RTO
  - Exponential backoff from congestion (the likely cause of lost packets)

TCP Adaptive Retransmission Algorithm – Karn-Partridge

- Problem 2: RTO = 2 × *EstimatedRTT*



TCP Adaptive Retransmission Algorithm – Karn-Partridge

- Problem 2: RTO = 2 × *EstimatedRTT*
- Did not consider the *variance* of the RTT samples
  - Smaller variance → More accurate *EstimatedRTT*
    - No reason for 2x
  - Larger variance → Maybe normal to exceed 2x
    - Unreliable outlier detection

TCP Adaptive Retransmission Algorithm – Jacobson

1. Maintain the moving average *EstimatedRTT*
2. Estimate RTT deviation
   - *Diff* = | *SampleRTT* – *EstimatedRTT* |
   - *Deviation* = β × *Diff* + (1–β) × *Deviation*
     - E.g., β = 0.25
3. RTO = *EstimatedRTT* + 4 × *Deviation*
   - What does this remind you?

We will see that accurate timeout mechanism is important for congestion control

Learning Objectives

- TCP state machine
- Retransmission timeout
- **Flow control**

Flow Control vs. Congestion Control

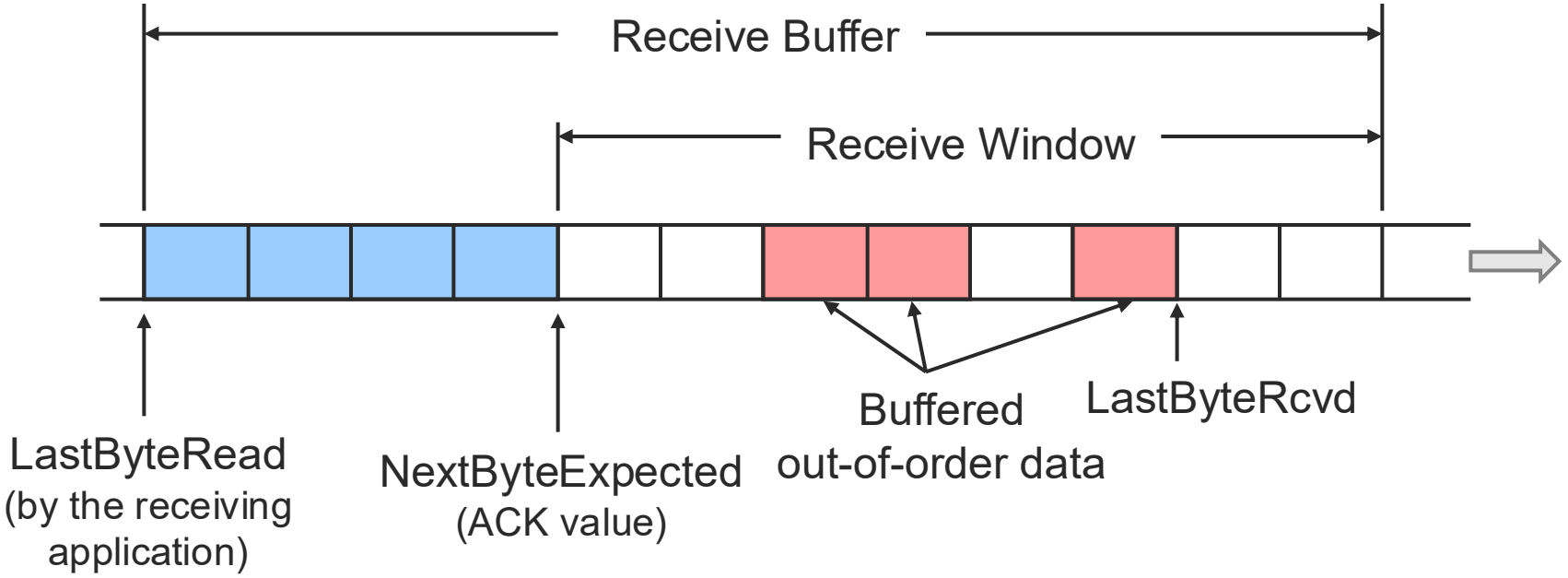
- Flow control
  - Prevents senders from overrunning the capacity of the **receiver**
- Congestion control
  - Prevents too much data from being injected into the **network**, causing switches or links to become overloaded
- TCP provides both
  - Flow control is based on receiver's advertised window
  - Congestion control focuses on adjusting the send window (next lecture)

TCP Flow Control: Receiver

- Receiver
  - Receive window: out-of-order data to receive
  - Receive buffer: data ready for delivery to the application but not requested yet

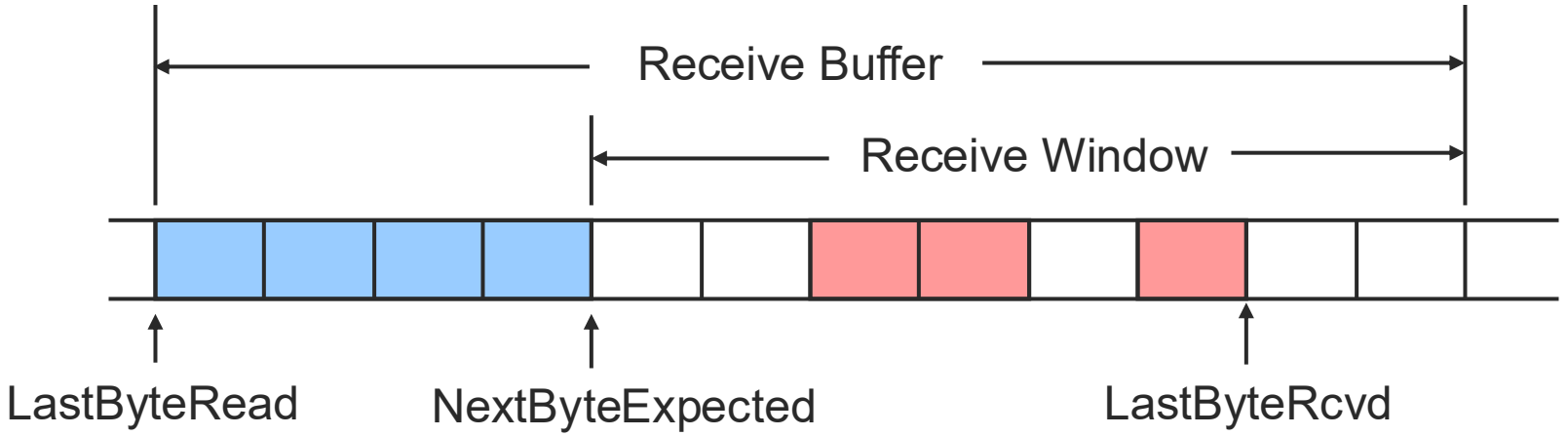
TCP Flow Control: Receiver

- LastByteRead < NextByteExpected
- NextByteExpected <= LastByteRcvd + 1
- Buffer bytes between LastByteRead and LastByteRcvd



TCP Flow Control: Receiver

- Receive buffer size (MaxRcvBuffer)
  - LastByteRcvd – LastByteRead <= MaxRcvBuf
- Receive window is advertised (AdvertWindow)
  - = MaxRcvBuf – (NextByteExpected – LastByteRead)
  - Shrinks as data arrives and
  - Grows as the application consumes data

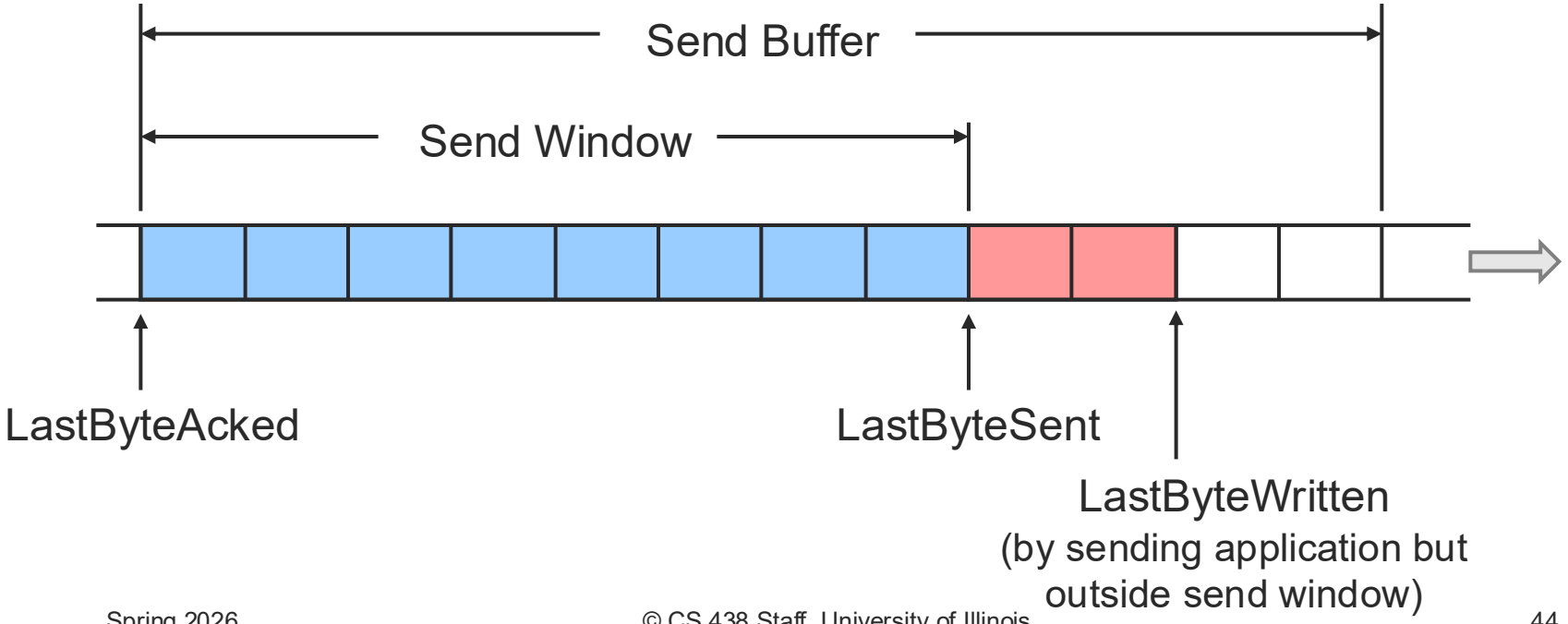


TCP Flow Control: Sender

- Sender
  - Send window: sent but unacknowledged data
  - Send buffer: data written by the application but not sent out yet

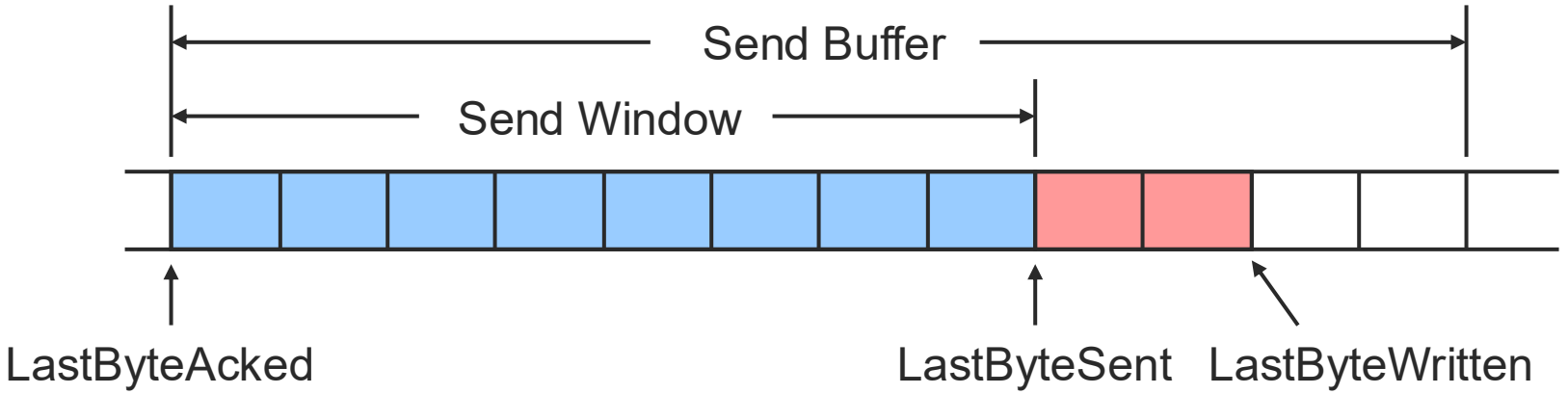
TCP Flow Control: Sender

- LastByteAcked <= LastByteSent
- LastByteSent <= LastByteWritten
- Buffer bytes between LastByteAcked and LastByteWritten



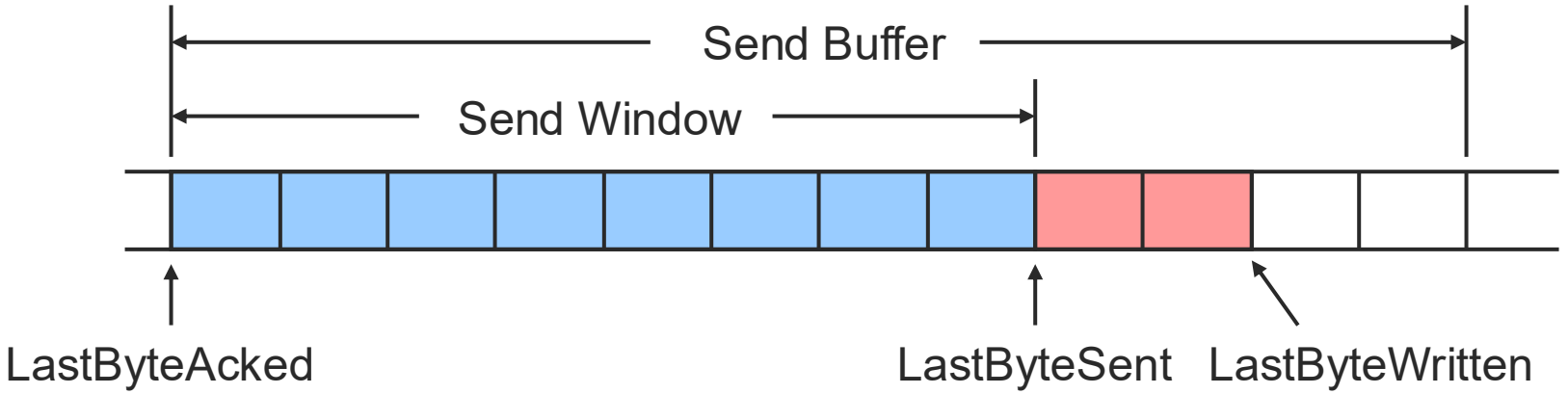
TCP Flow Control: Sender

- Send buffer size (MaxSendBuffer)
  - LastByteWritten – LastByteAcked <= MaxSendBuffer
- Block sender if
  - (LastByteWritten - LastByteAcked) + *x* > MaxSenderBuffer



TCP Flow Control: Sender

- Send window is restricted by receiver's advertised window
  - LastByteSent – LastByteAcked <= AdvertWindow
- Effective window (congestion control ignored for now)
  - = AdvertWindow – (LastByteSent – LastByteAcked)
  - Can send data only if effective window > 0



Advertised Window Limits Rate

- Sender can send no faster than AdvertWindow / RTT
  - Receiver implicitly limits sender to rate that receiver can sustain
  - If sender is going too fast, receiver's window advertisements get smaller & smaller

TCP Flow Control

- Problem: Slow receiving application
  - Advertised window goes to 0
  - Receiver may not spontaneously send a window update when buffer space is freed, or the update may be lost
  - As a result, sender cannot send more data
- Solution
  - Sender periodically sends 1-byte window probes, ignoring advertised window of 0
  - Eventually window opens
  - Sender learns of opening from next ACK of 1-byte segment

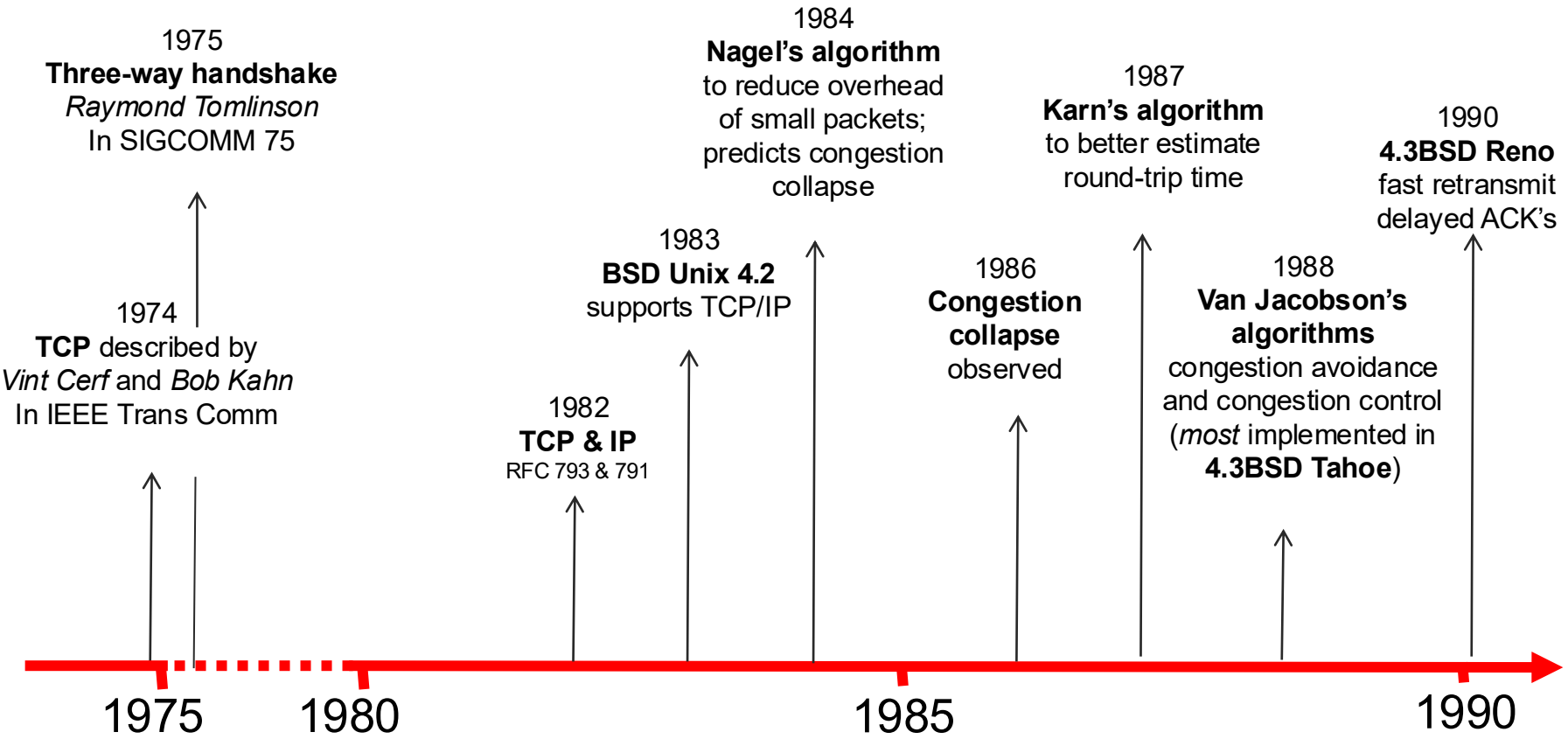
TCP Flow Control

- Problem: Slow application reads data in tiny pieces
  - Receiver advertises tiny window
  - Sender fills tiny window
  - Known as silly window syndrome
- Solution
  - Advertise window opening only when MSS or half the buffer is available

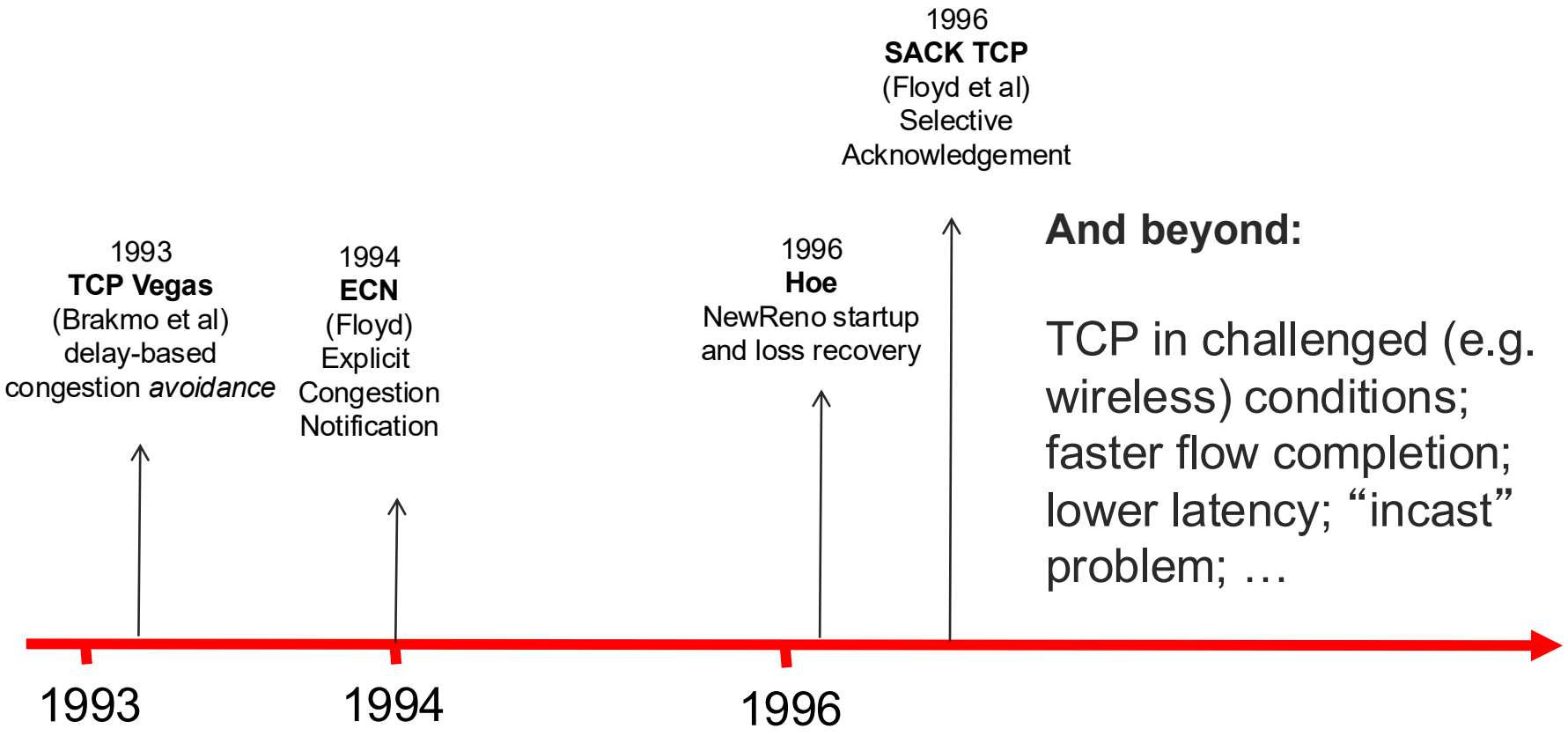
Sender-Side Efficiency

- Problem: Sending application delivers tiny pieces of data to TCP
  - Each piece sent as a segment, returned as ACK
  - Very inefficient
- Solution
  - Delay transmission to accumulate more data
  - Nagle's algorithm
    - Available data is ≥ MSS and window allows:
      - send full segment
    - Otherwise, if there's unacknowledged data in flight:
      - accumulate data until an ACK
  - Do we always want this feature?

Evolution of TCP



TCP Through the 1990s



And beyond:

TCP in challenged (e.g. wireless) conditions; faster flow completion; lower latency; "incast" problem; …