
TCP

TCP Adaptive Retransmission Algorithm - Original

- Theory
  - Estimate RTT
  - Multiply by 2 to allow for variations
- Practice
  - Use exponential moving average (A = 0.1 to 0.2)
  - Estimate = (A) * measurement + (1- A) * estimate
- Problem
  - Did not handle variations well
  - Ambiguity for retransmitted packets
    - Was ACK in response to first, second, etc transmission?

TCP Adaptive Retransmission Algorithm – Karn-Partridge

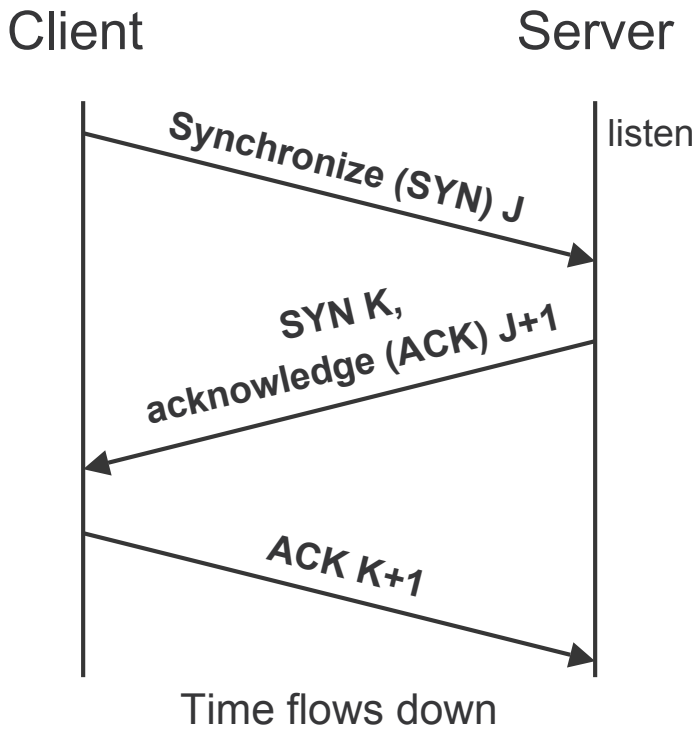
- Algorithm
  - Exclude retransmitted packets from RTT estimate
  - For each retransmission
    - Double RTT estimate
    - Exponential backoff from congestion
- Problem
  - Still did not handle variations well
  - Did not solve network congestion problems as well as desired

TCP Adaptive Retransmission Algorithm – Jacobson

- Algorithm
  - Estimate variance of RTT
    - Calculate mean interpacket RTT deviation to approximate variance
    - Use second exponential moving average
    - Deviation = (B) * |RTT_Estimate – Measurement| + (1–B) * deviation
    - B = 0.25, A = 0.125 for RTT_estimate
  - Use variance estimate as component of RTT estimate
    - Next_RTT = RTT_Estimate + 4 * Deviation
  - Protects against high jitter
- Notes
  - Algorithm is only as good as the granularity of the clock
  - Accurate timeout mechanism is important for congestion control

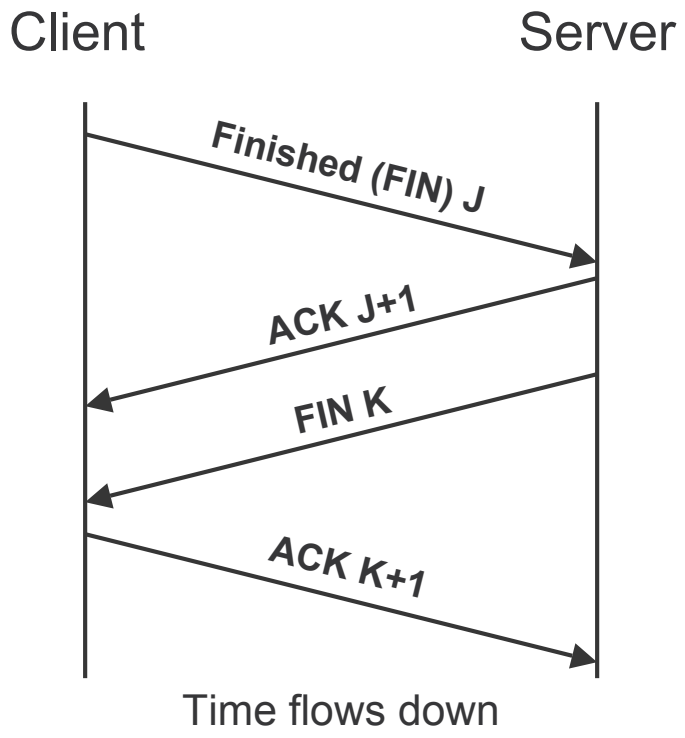
TCP Connection Establishment

- 3-Way Handshake
  - Sequence Numbers
    - J,K
  - Message Types
    - Synchronize (SYN)
    - Acknowledge (ACK)
  - Passive Open
    - Server listens for connection from client
  - Active Open
    - Client initiates connection to server

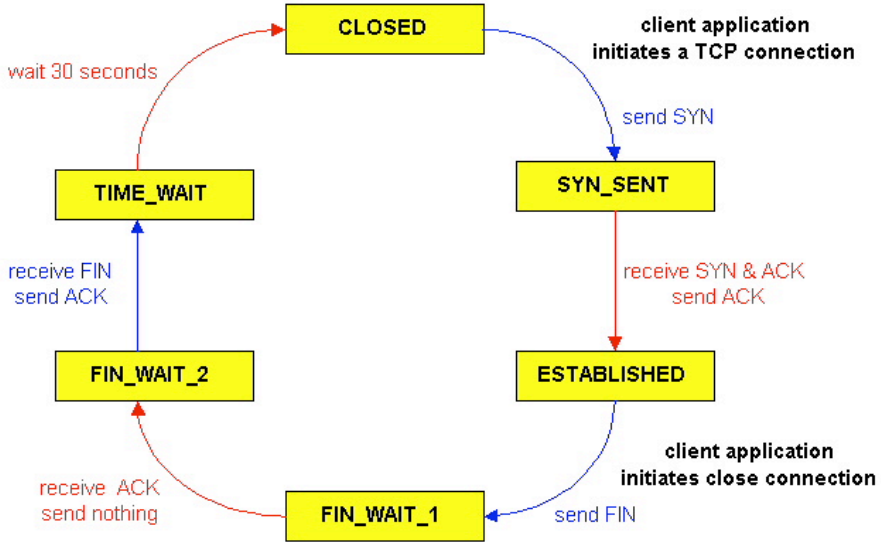


TCP Connection Termination

- Message Types
  - Finished (FIN)
  - Acknowledge (ACK)
- Active Close
  - Sends no more data
- Passive close
  - Accepts no more data

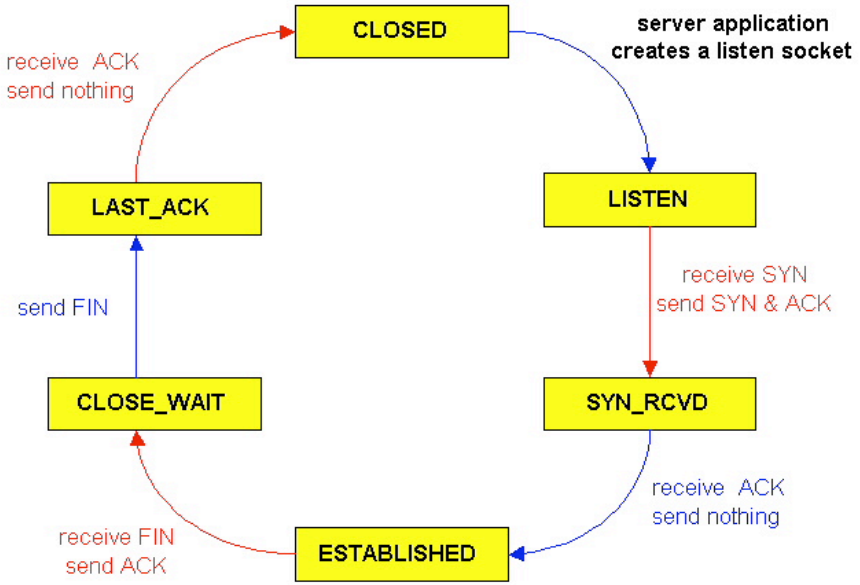


TCP Connection Management (cont)



TCP client lifecycle

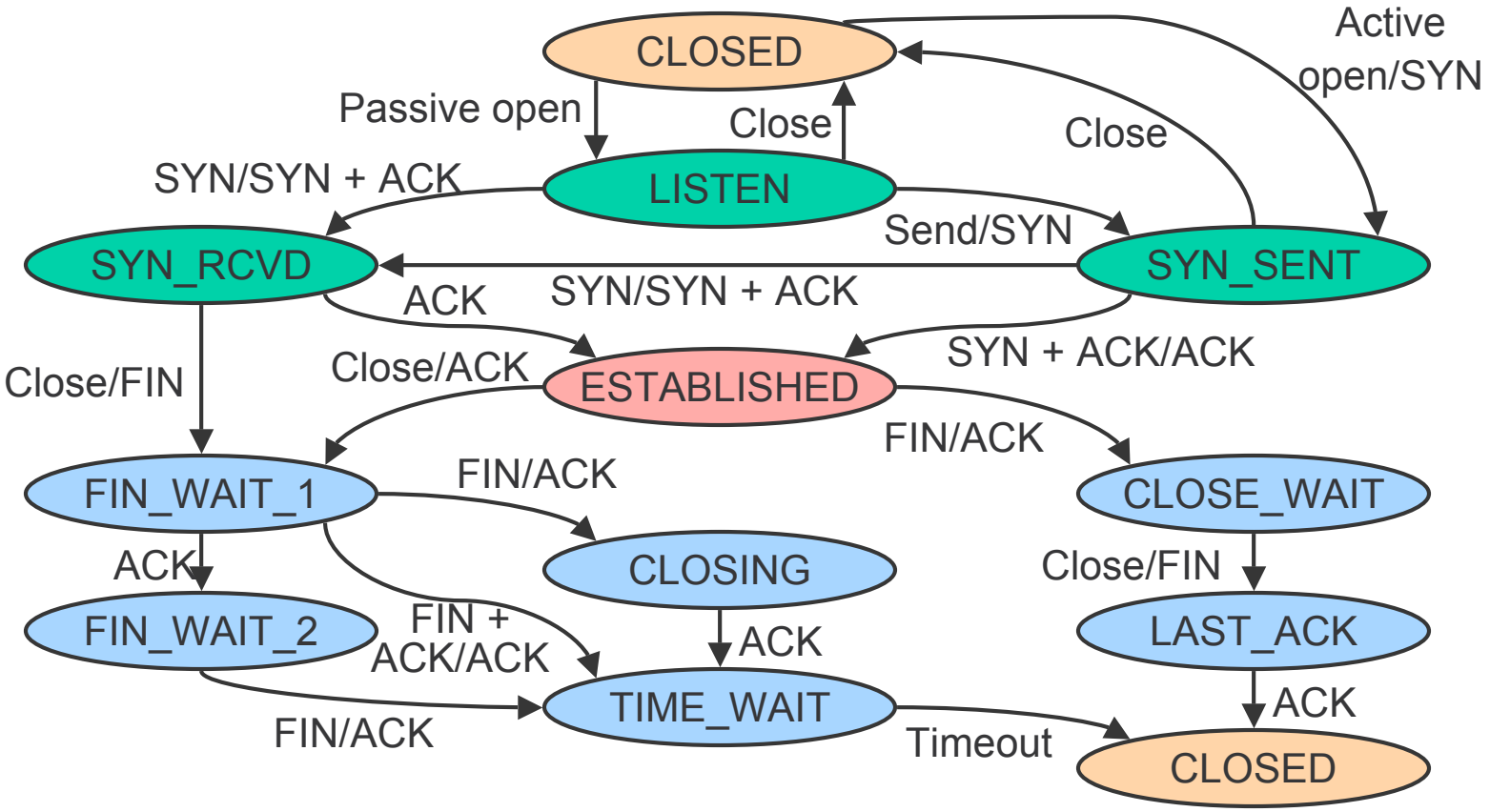
TCP server lifecycle



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

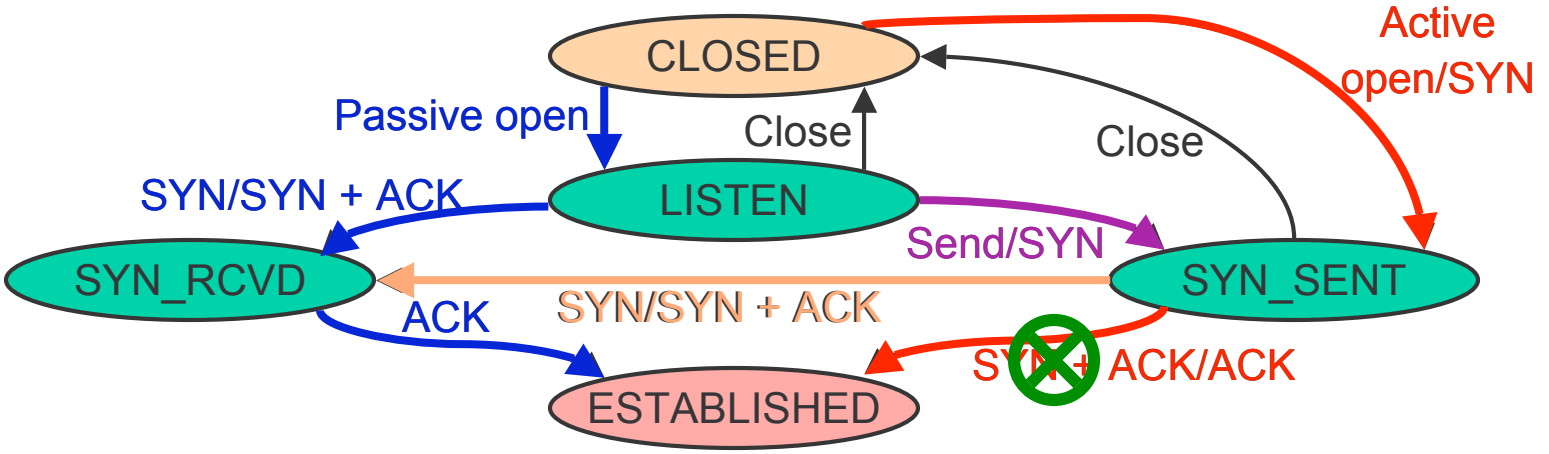
TCP State Transition Diagram



TCP State Transition Diagram

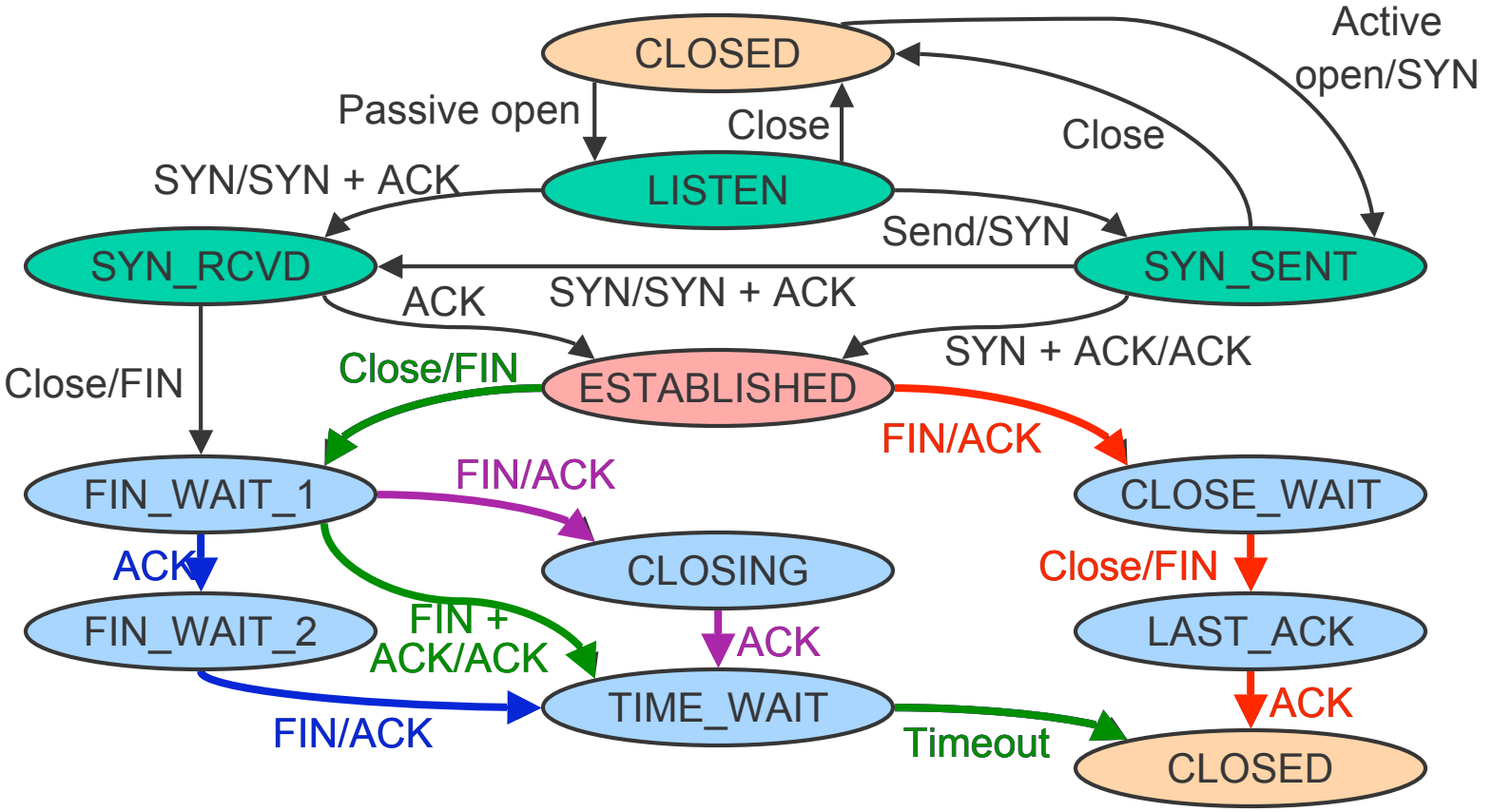
- Questions
  - State transitions
    - Describe the path taken by a server under normal conditions
    - Describe the path taken by a client under normal conditions
    - Describe the path taken assuming the client closes the connection first
  - TIME_WAIT state
    - What purpose does this state serve
    - Prove that at least one side of a connection enters this state
    - Explain how both sides might enter this state

TCP State Transition Diagram



	TCP A			TCP B
1.	CLOSED			LISTEN
2.	SYN-SENT	--> <SEQ=100><CTL=SYN>	-->	SYN-RECEIVED
3.	ESTABLISHED	<-- <SEQ=300><ACK=101><CTL=SYN,ACK>	<--	SYN-RECEIVED
4.	ESTABLISHED	--> <SEQ=101><ACK=301><CTL=ACK>	-->	ESTABLISHED
5.	ESTABLISHED	--> <SEQ=101><ACK=301><CTL=ACK><DATA>	-->	ESTABLISHED

TCP State Transition Diagram

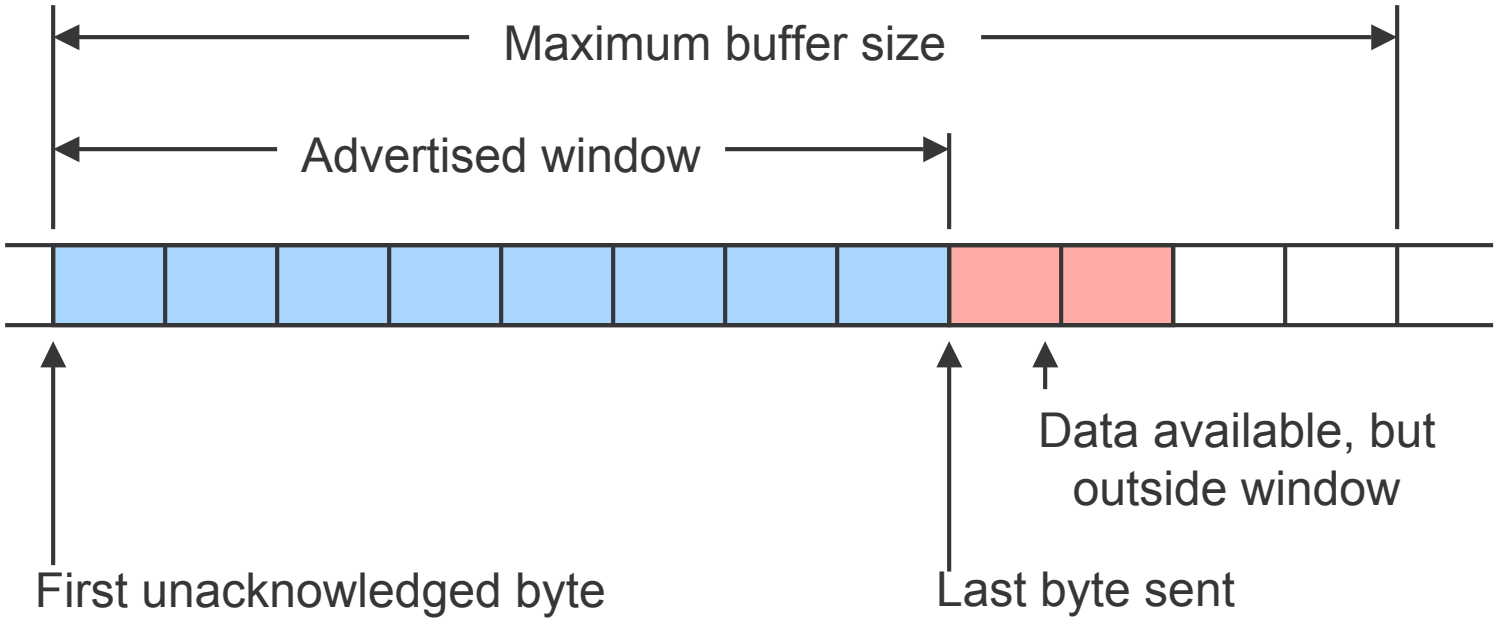


TCP Sliding Window Protocol

- Sequence numbers
  - Indices into byte stream
- ACK sequence number
  - Actually next byte expected as opposed to last byte received
- Advertised window
  - Enables dynamic receive window size
- Receive buffers
  - Data ready for delivery to application until requested
  - Out-of-order data out to maximum buffer capacity
- Sender buffers
  - Unacknowledged data
  - Unsent data out to maximum buffer capacity

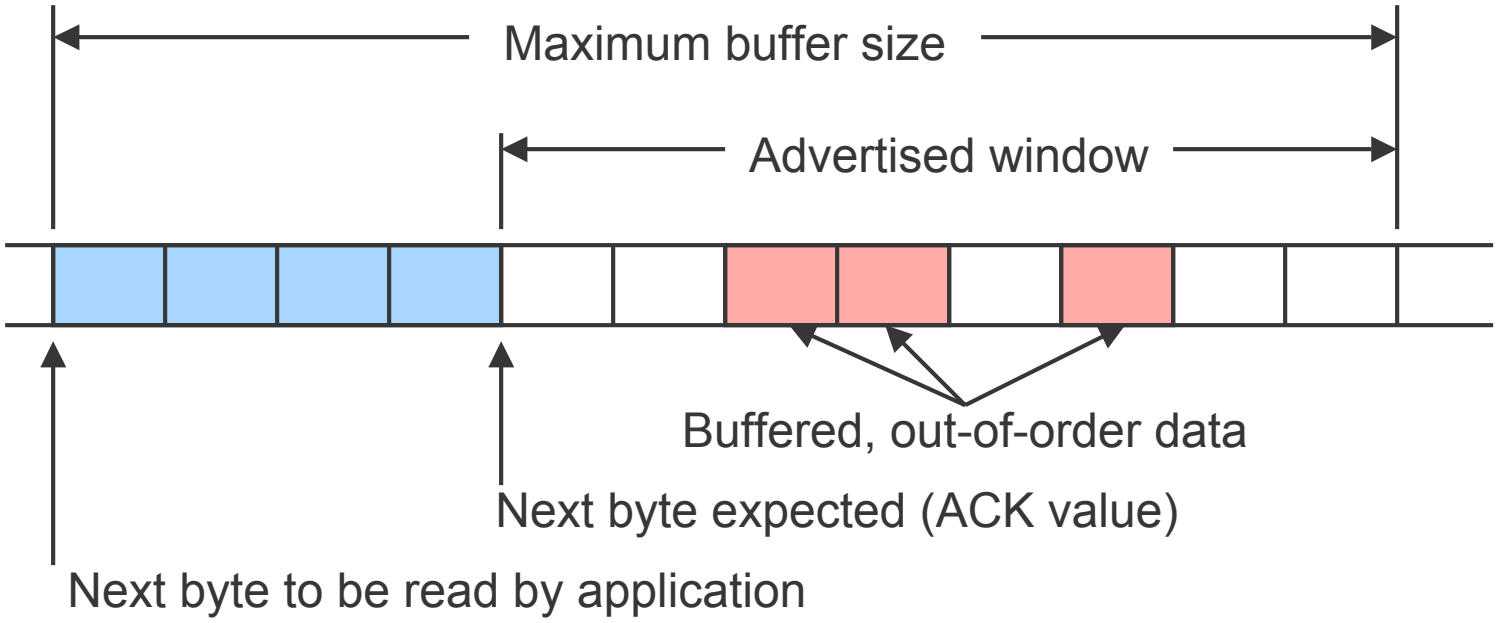
TCP Sliding Window Protocol – Sender Side

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

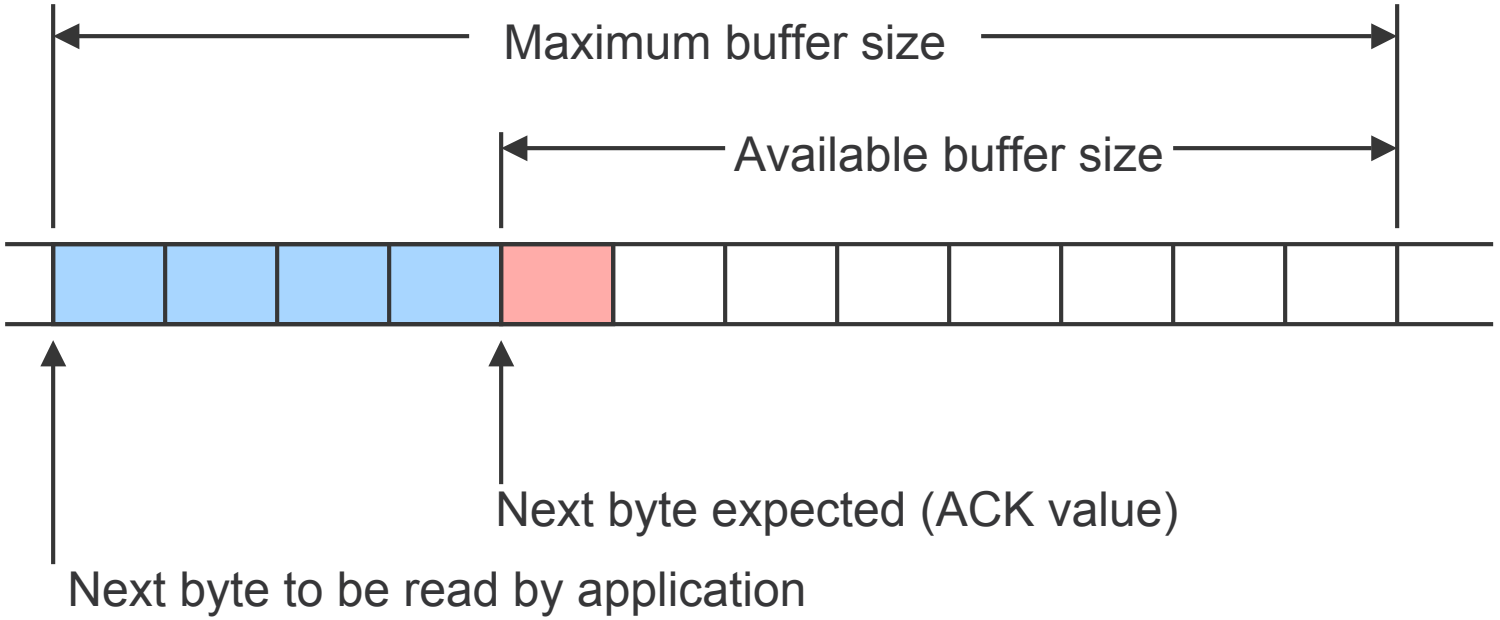


TCP Sliding Window Protocol – Receiver Side

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

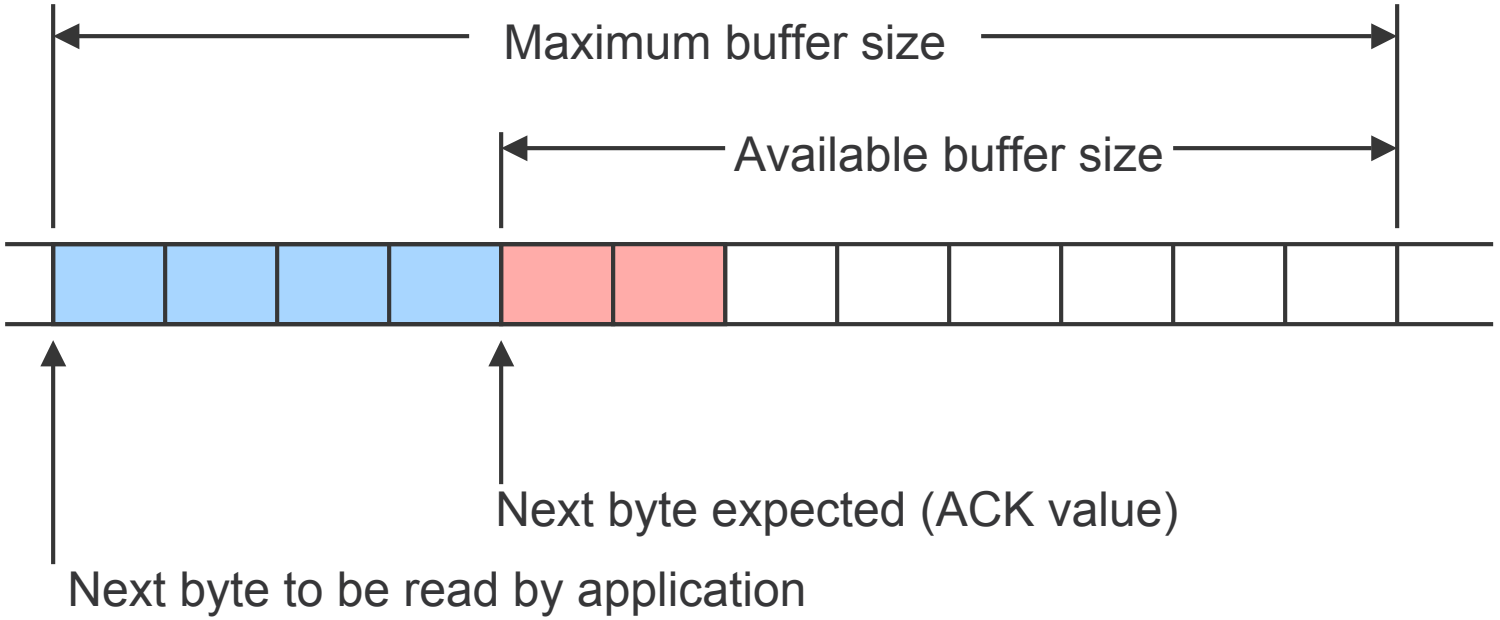


TCP ACK generation - 1



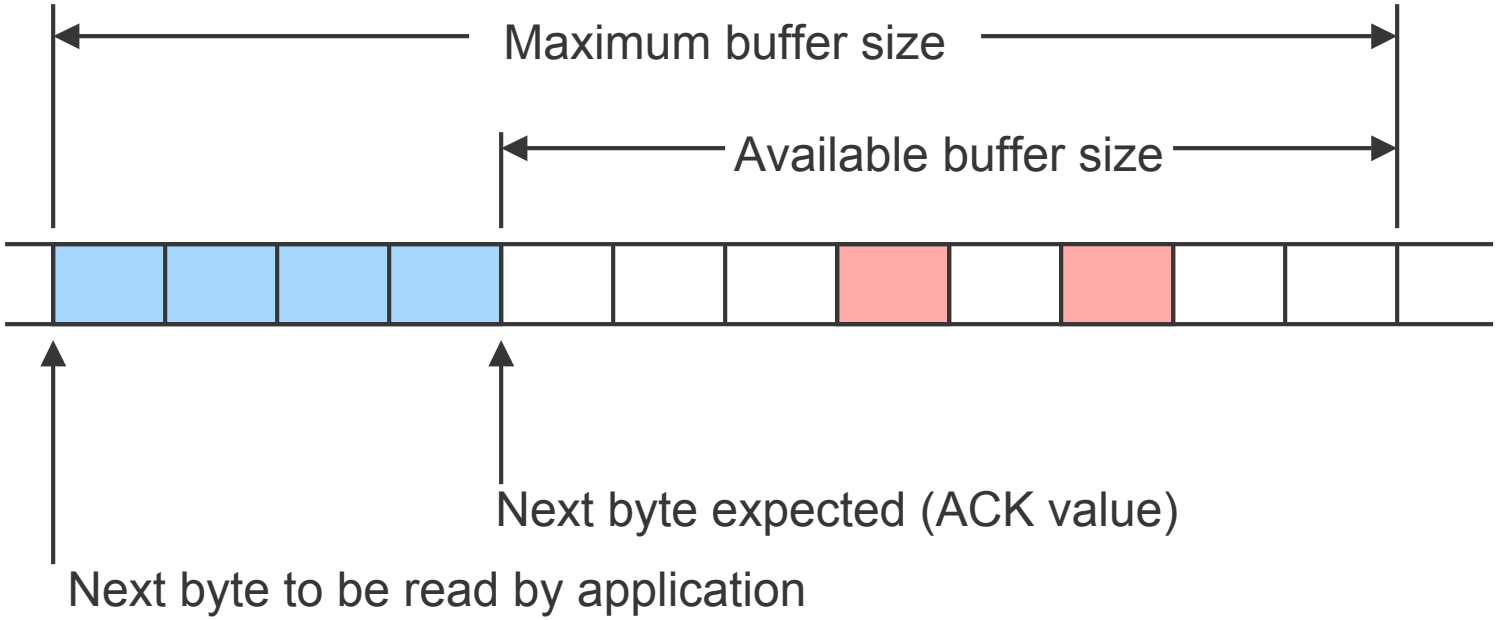
- Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed
  - Delayed ACK. Wait up to 500ms for next segment.

TCP ACK generation - 2



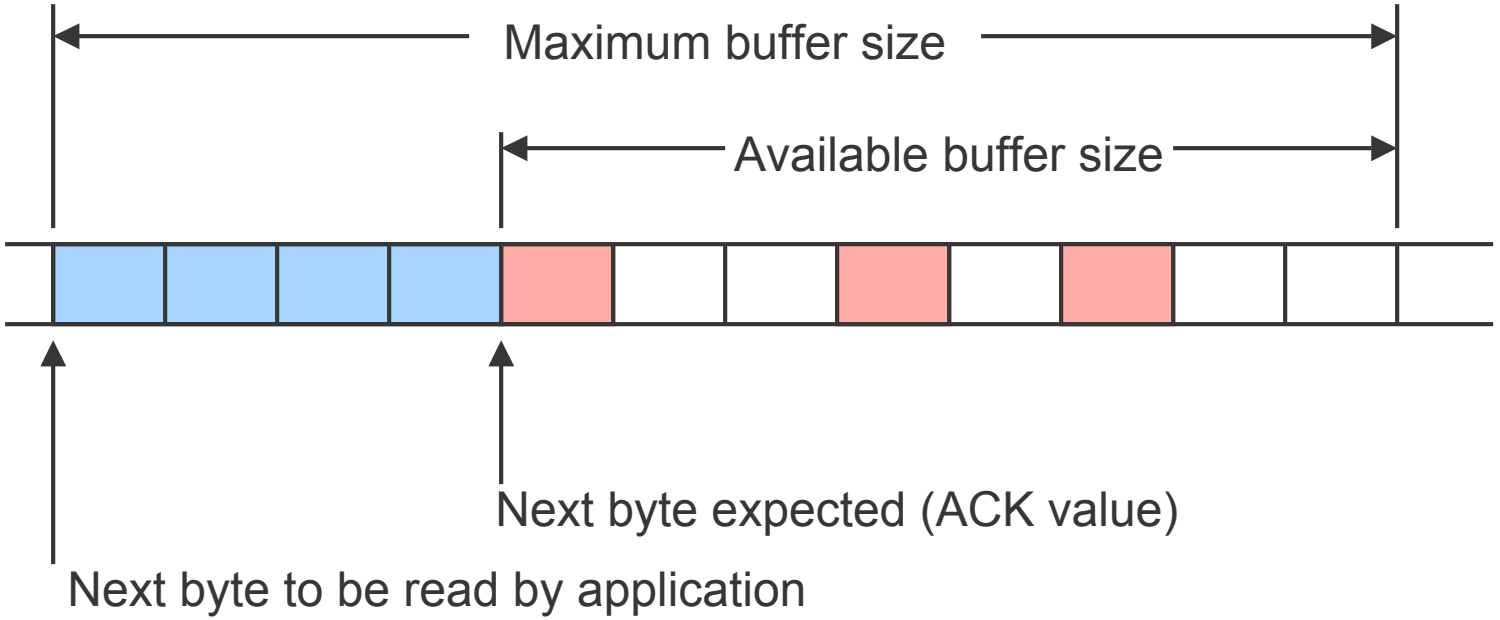
- Arrival of in-order segment with expected seq #. One other segment has ACK pending
  - Immediately send single cumulative ACK, ACKing both in-order segments

TCP ACK generation - 3



- Arrival of out-of-order segment higher-than-expect seq. # Gap detected
  - Immediately send duplicate ACK, indicating seq. # of next expected byte

TCP ACK generation - 4



- Arrival of segment that partially or completely fills gap
  - Immediate send ACK, provided that segment starts at lower end of gap

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

Fast Retransmit

- What's the problem with time-out?
  - time-out period often relatively long
- Detect lost segments via duplicate ACKs.
  - Sender often sends many segments back-to-back
  - If segment is lost, there will likely be many duplicate ACKs.
- If sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:
  - fast retransmit: resend segment before timer expires
- Why 3?

Fast retransmit algorithm:

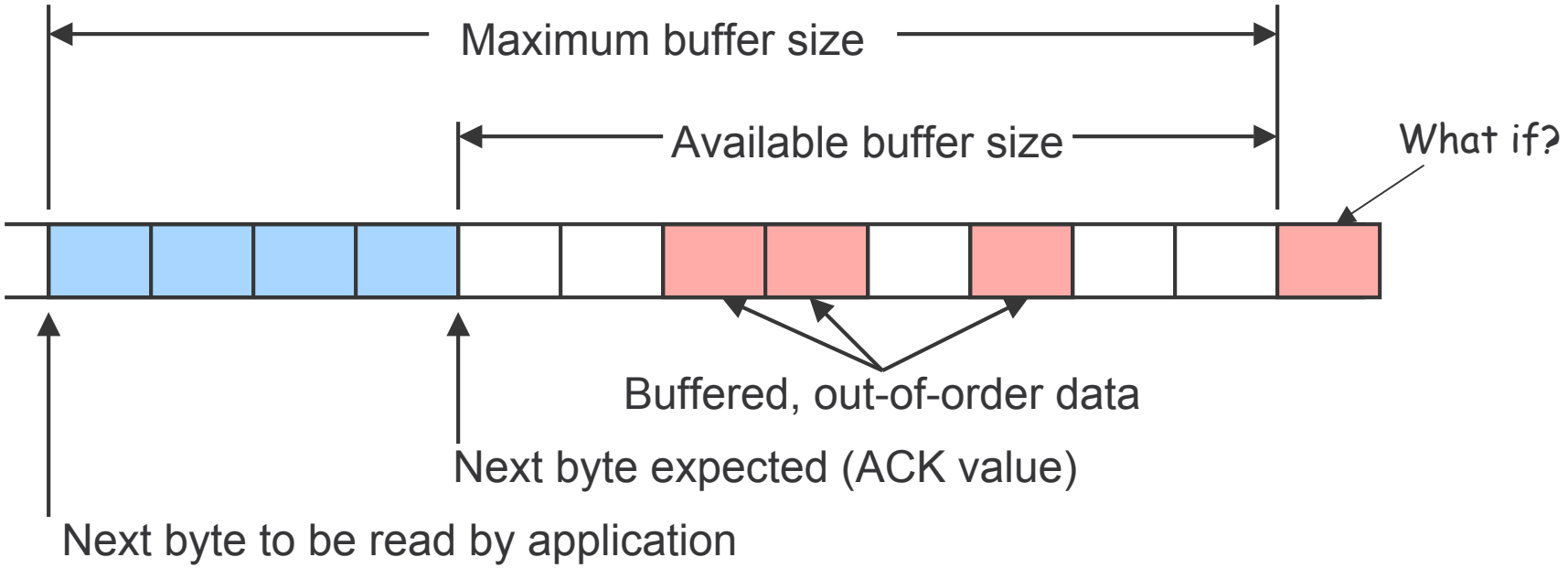
```
event: ACK received, with ACK field value of y
            if (y > SendBase) {
                SendBase = y
                if (there are currently not-yet-acknowledged segments)
                    start timer
              }
            else {
                increment count of dup ACKs received for y
                if (count of dup ACKs received for y = 3) {
                    resend segment with sequence number y
                 }
```

a duplicate ACK for already ACKed segment

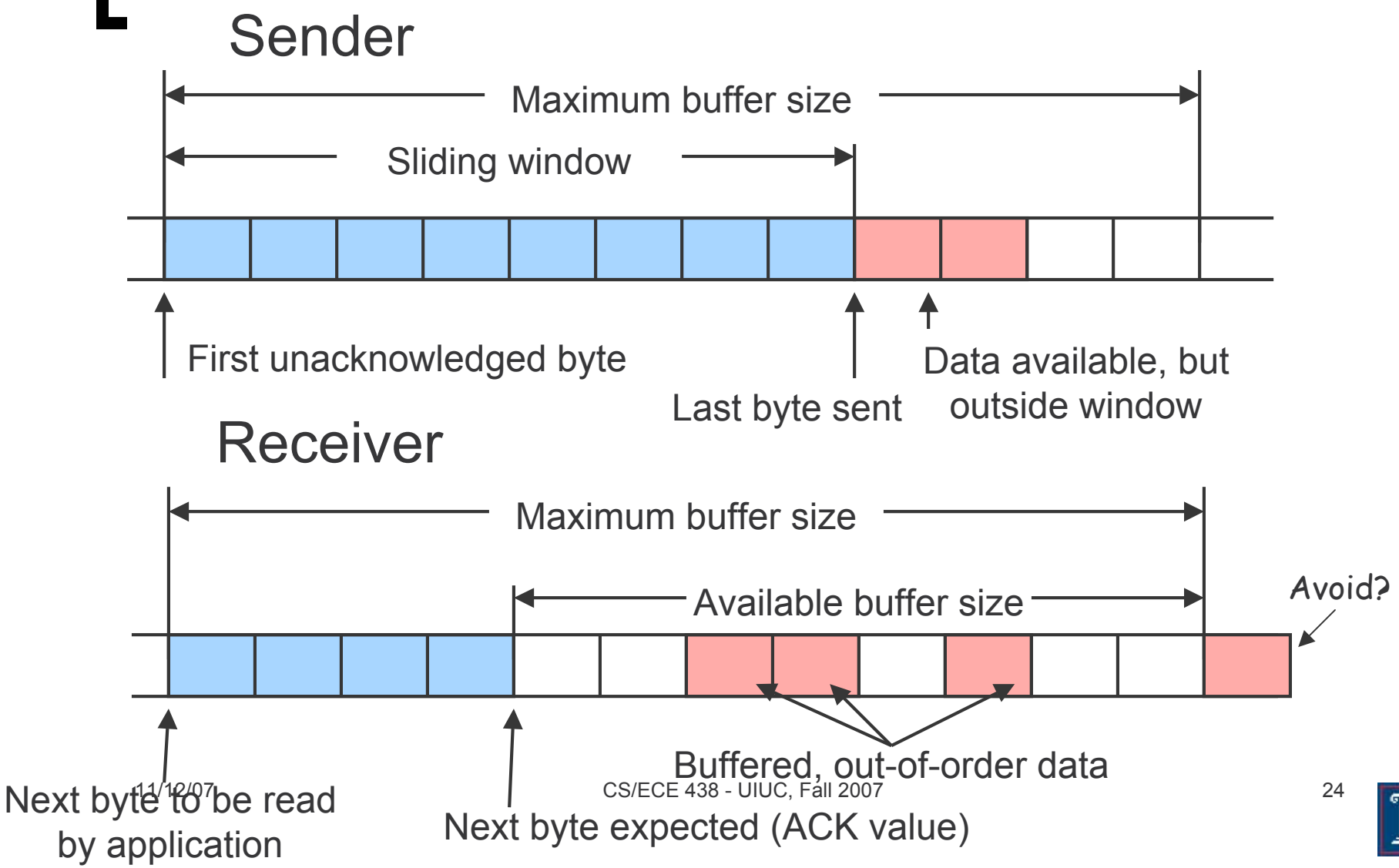
fast retransmit

A 4th situation?

- Receiver side



TCP Flow Control



Flow Control vs. Congestion Control

- Flow control
  - Preventing senders from overrunning the capacity of the receivers
- Congestion control
  - Preventing too much data from being injected into the network, causing switches or links to become overloaded
- TCP provides both
  - flow control based on advertised window
  - congestion control discussed later in class

TCP Flow Control

- Receiving side
  - Receive buffer size = **`MaxRcvBuffer`**
  - **`LastByteRcvd - LastByteRead < = MaxRcvBuffer`**
  - **`AdvertisedWindow = MaxRcvBuffer - (NextByteExpected - NextByteRead)`**
    - Shrinks as data arrives and
    - Grows as the application consumes data
- Sending side
  - Send buffer size = **`MaxSendBuffer`**
  - **`LastByteSent - LastByteAcked < = AdvertisedWindow`**
  - **`EffectiveWindow = AdvertisedWindow - (LastByteSent - LastByteAcked)`**
    - **`EffectiveWindow > 0 to send data`**
  - **`LastByteWritten - LastByteAcked < = MaxSendBuffer`**
  - **`block sender if (LastByteWritten - LastByteAcked) + y > MaxSenderBuffer`**