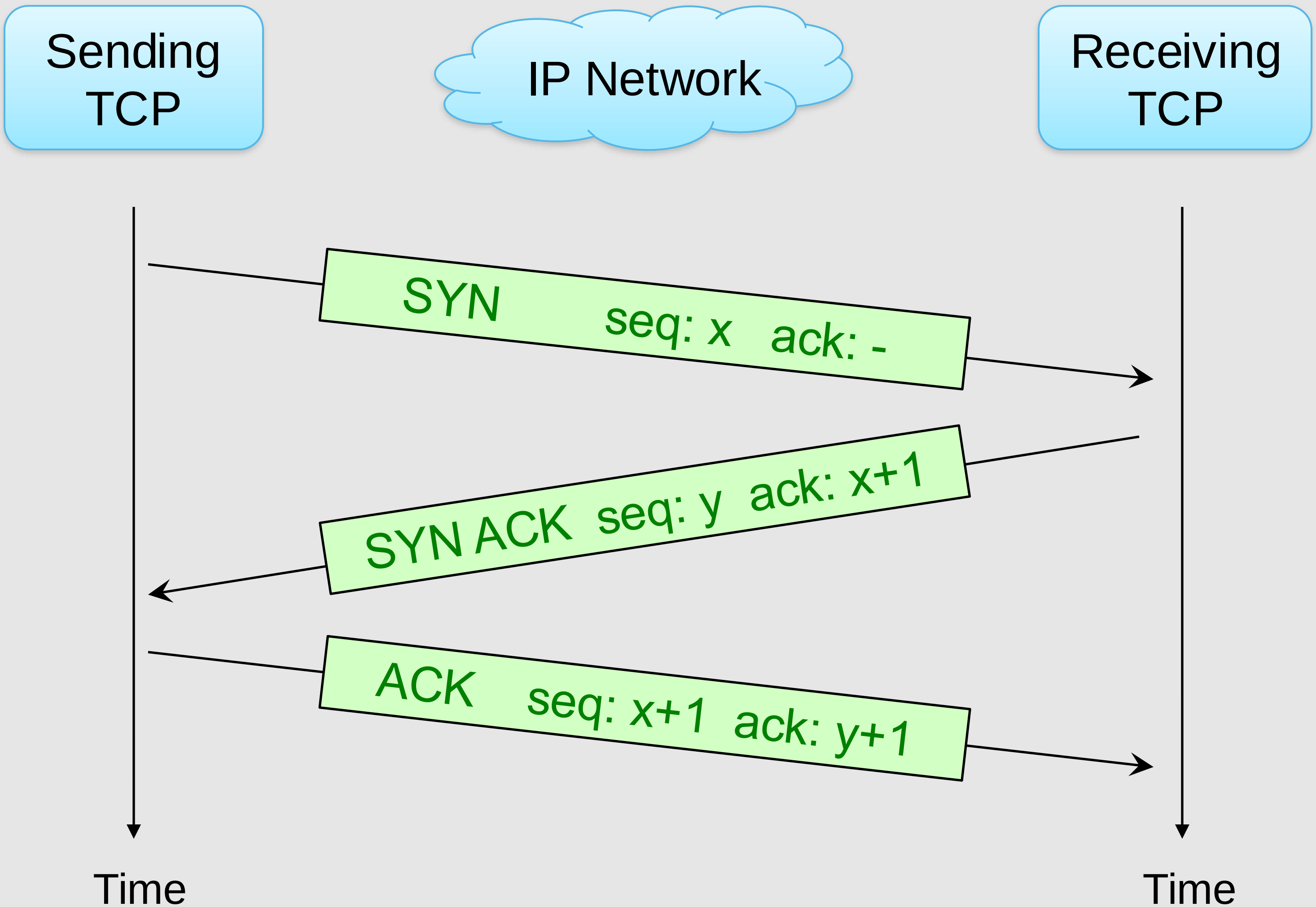
Transmission Control Protocol

Connection Establishment and Termination

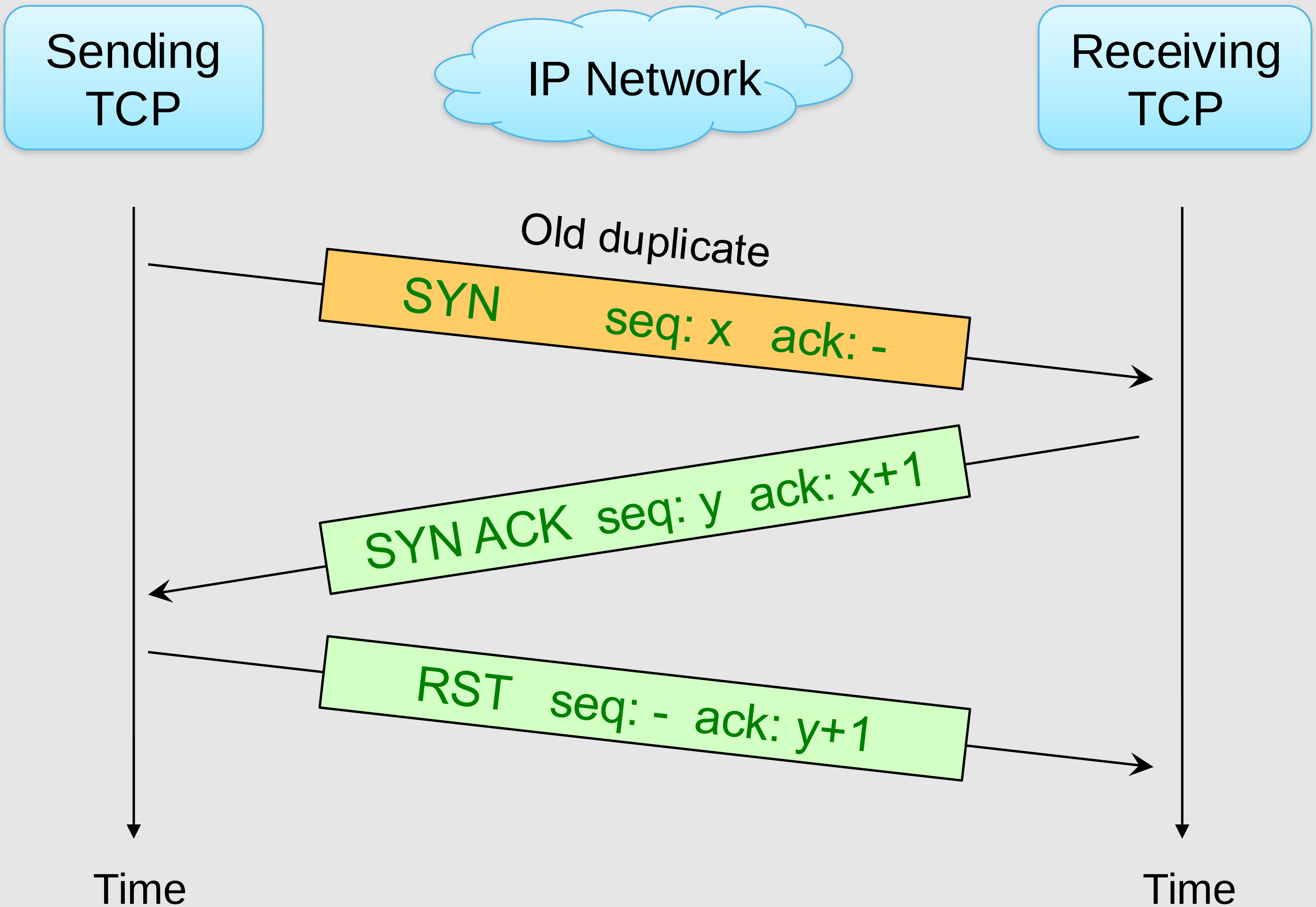
Quick Review -- Control Field

Flag	Description
URG	The value of the urgent pointer is valid
ACK	The value of the acknowledgment number is valid
PSH	Push the data (pass data to receiver as quickly as possible, later)
RST	The connection must be reset (e.g., blocked)
SYN	Synchronize the sequence numbers during connection establishment (start a new connection)
FIN	The sender has no more data to transmit (end)

Three-Way Handshake

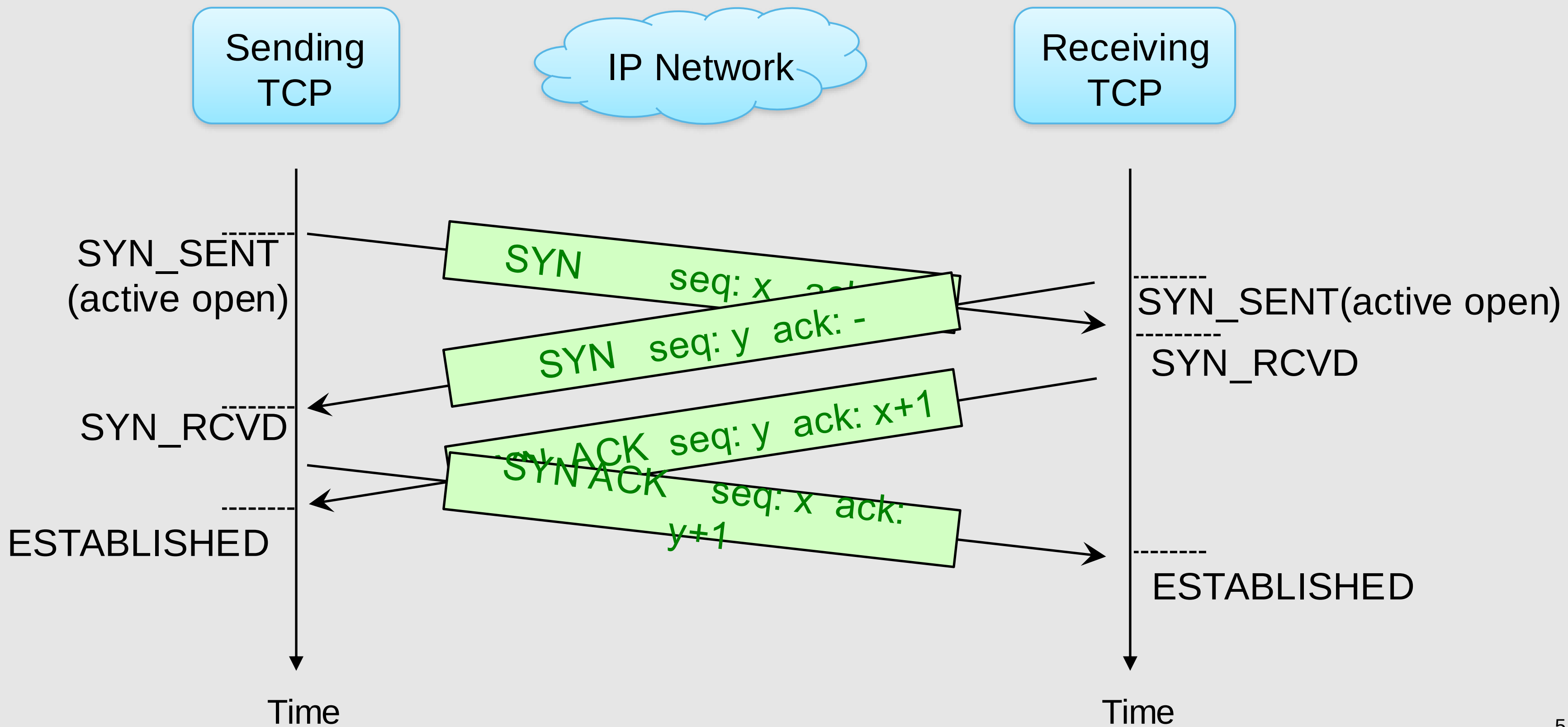


Three-Way Handshake



Simultaneous Open

- A connects with B, B connects with A at same time (pass each other in the network)
  - Only one connection will be established!
  - Using only 2 ports (one on A, one on B)

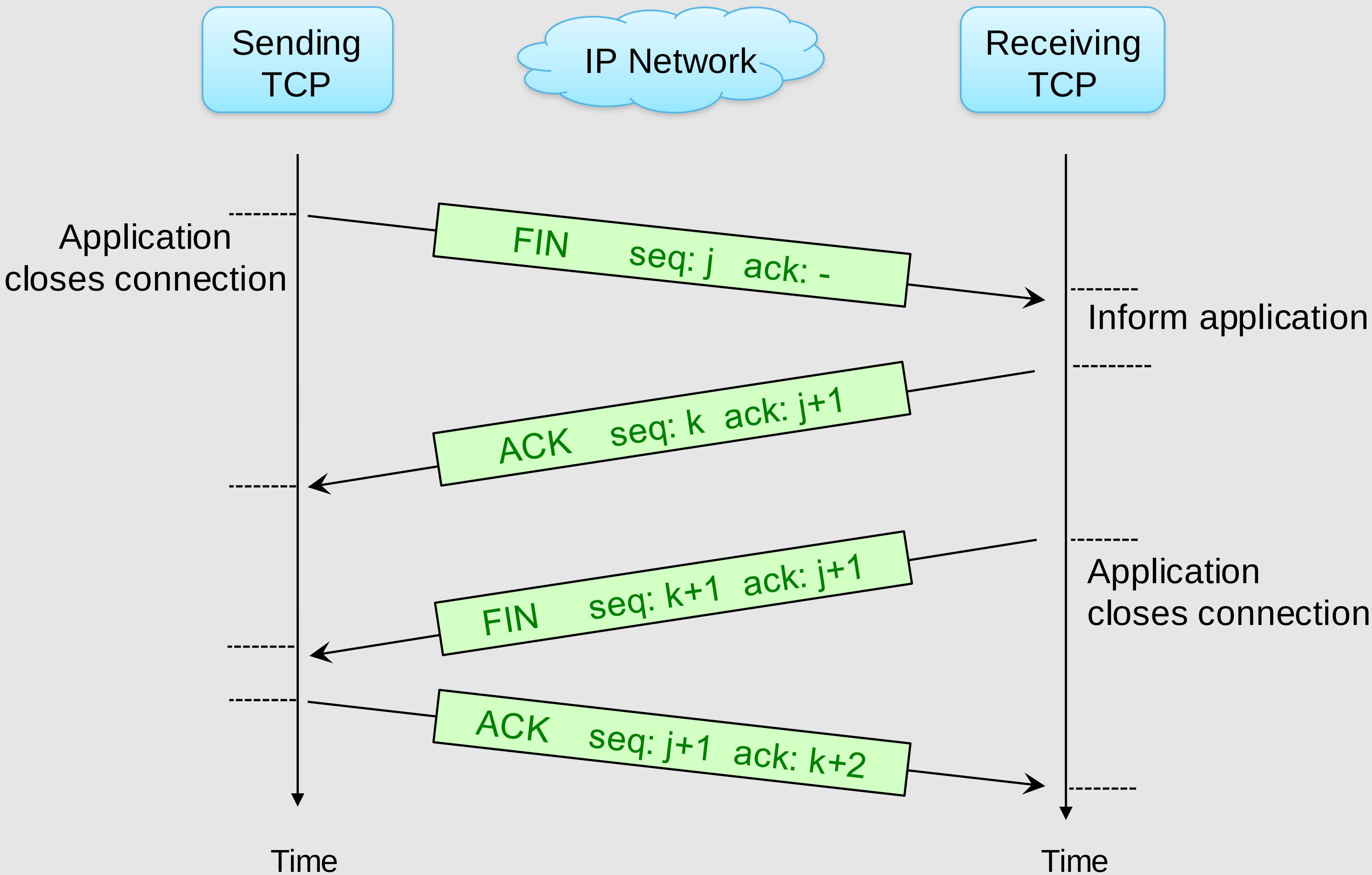


Connection Termination Issues

- Asymmetric release
  - abrupt, without coordination, data may be lost
- Symmetric release
  - each direction released independently of the other
  - each side can close its transmission

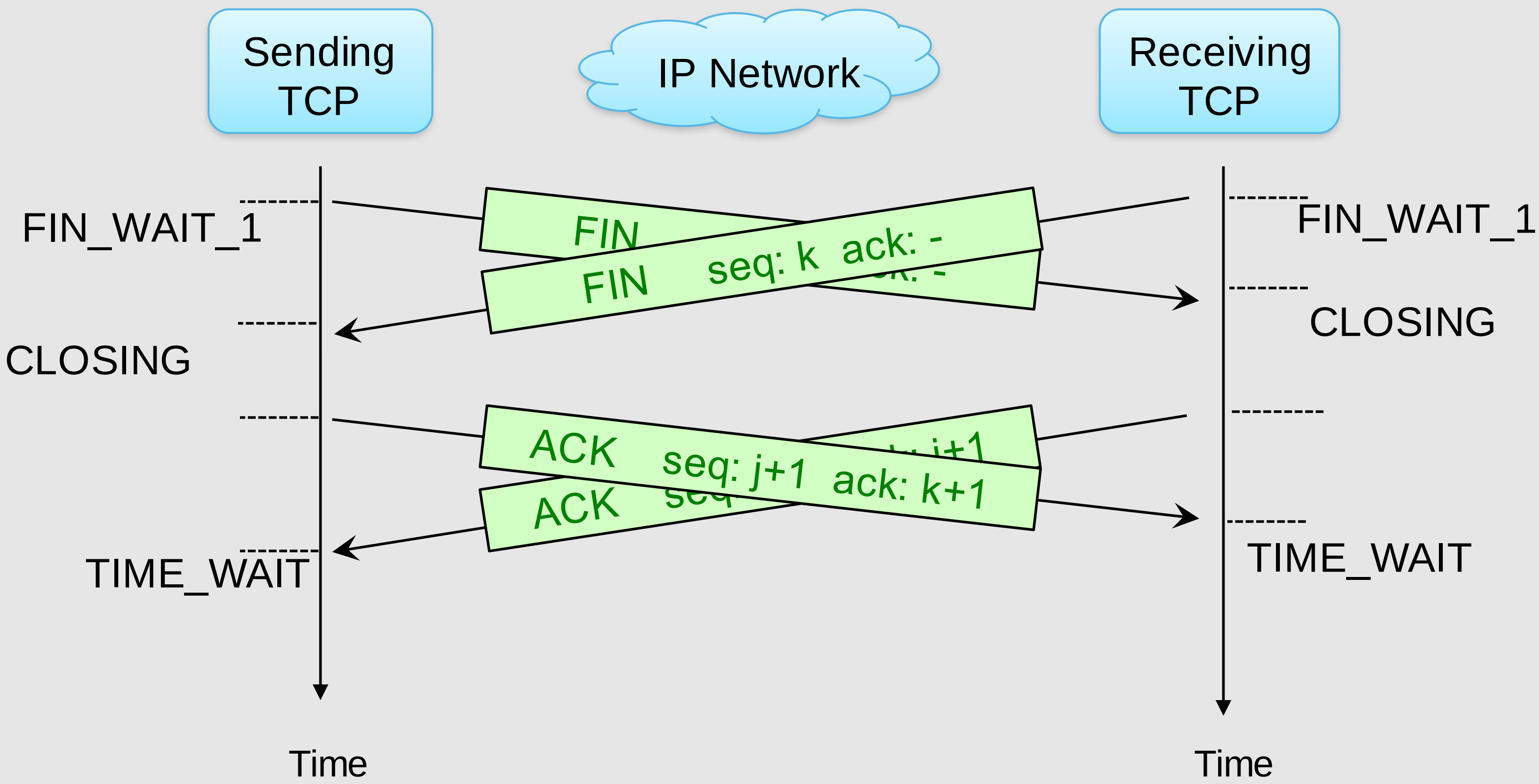
- To avoid data loss in a symmetric release…
  - no side should disconnect until it is convinced that the other side is also prepared to disconnect

TCP Connection Termination



Simultaneous Close

- A sends FIN to B, B sends FIN to A at same time (pass each other in the network)



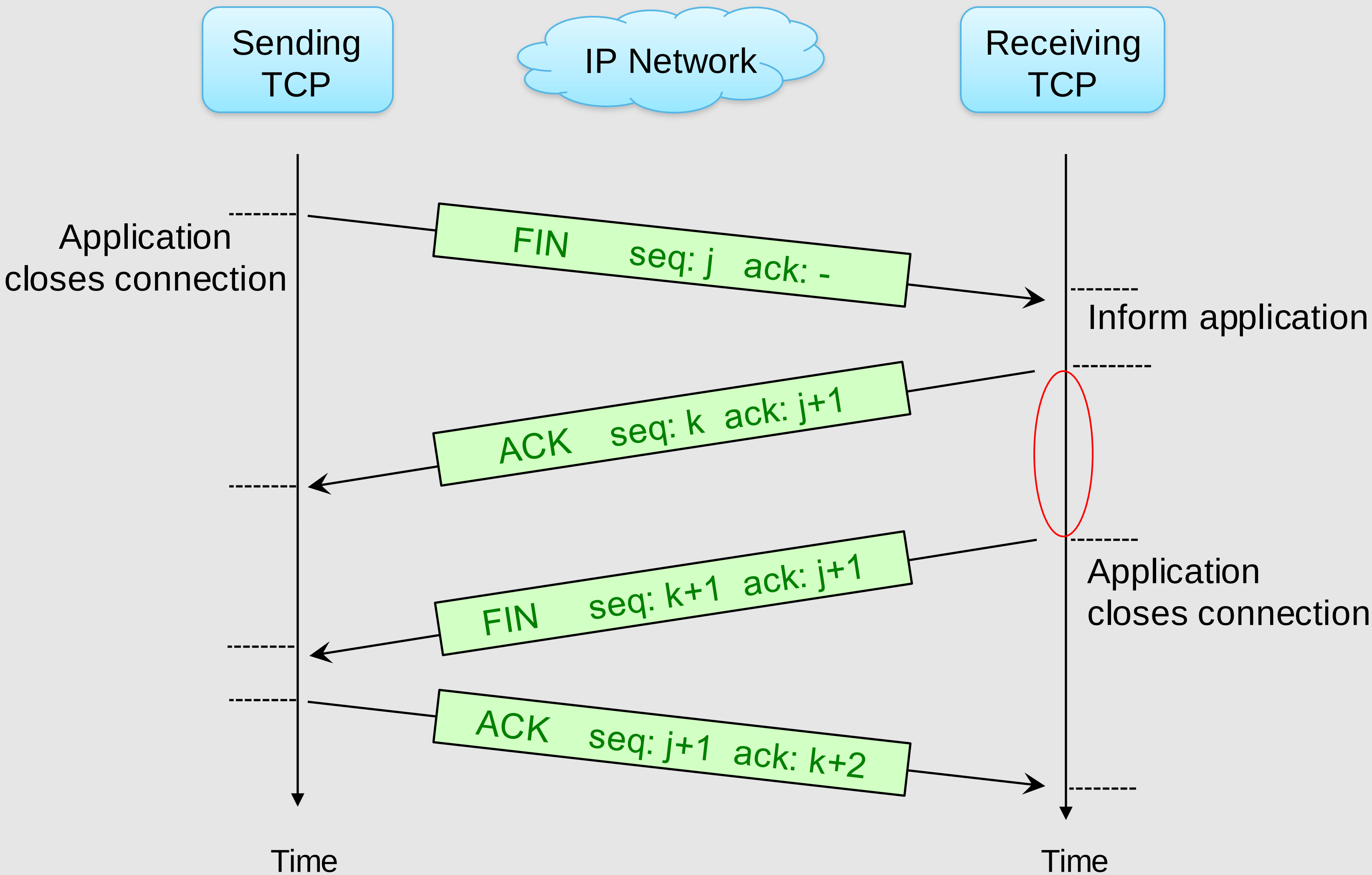
Connection Reset

- A connection can also be aborted with a RST segment (hard reset)
  - normally reserved for error conditions, not normal termination

Half Closed Connection

- One end of connection (e.g. client→server) terminates (sends FIN and receives ACK of FIN)
- Other end (server→client) remains open (sending data)
- Other end (server) later terminates (sends FIN and receives ACK of FIN), and connection is then completely closed

TCP Connection Termination



TCP Connection States

CLOSED

LISTEN

SYN_RCVD

SYN_SENT

ESTABLISHED

CLOSE_WAIT

FIN_WAIT_1

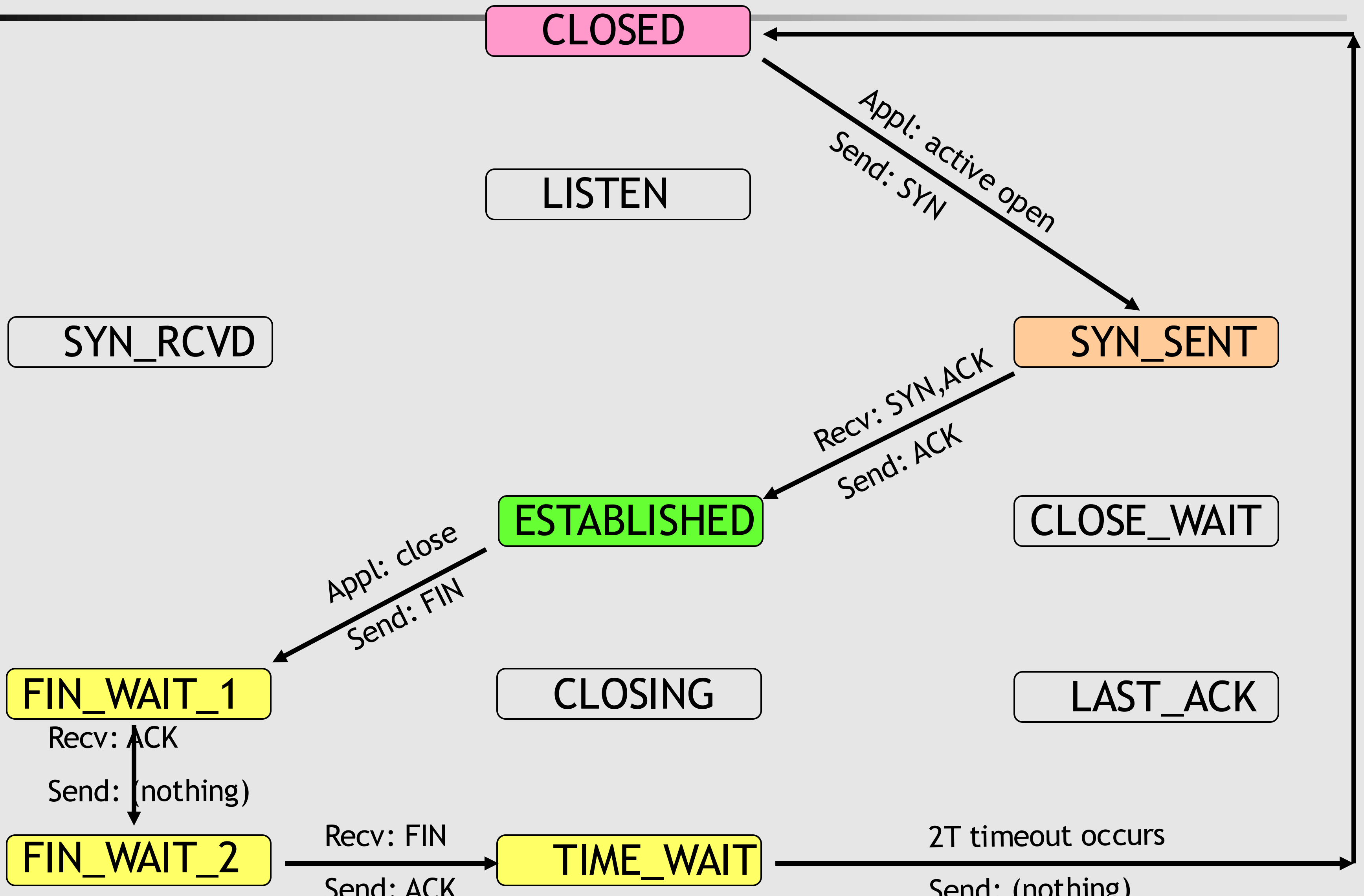
CLOSING

LAST_ACK

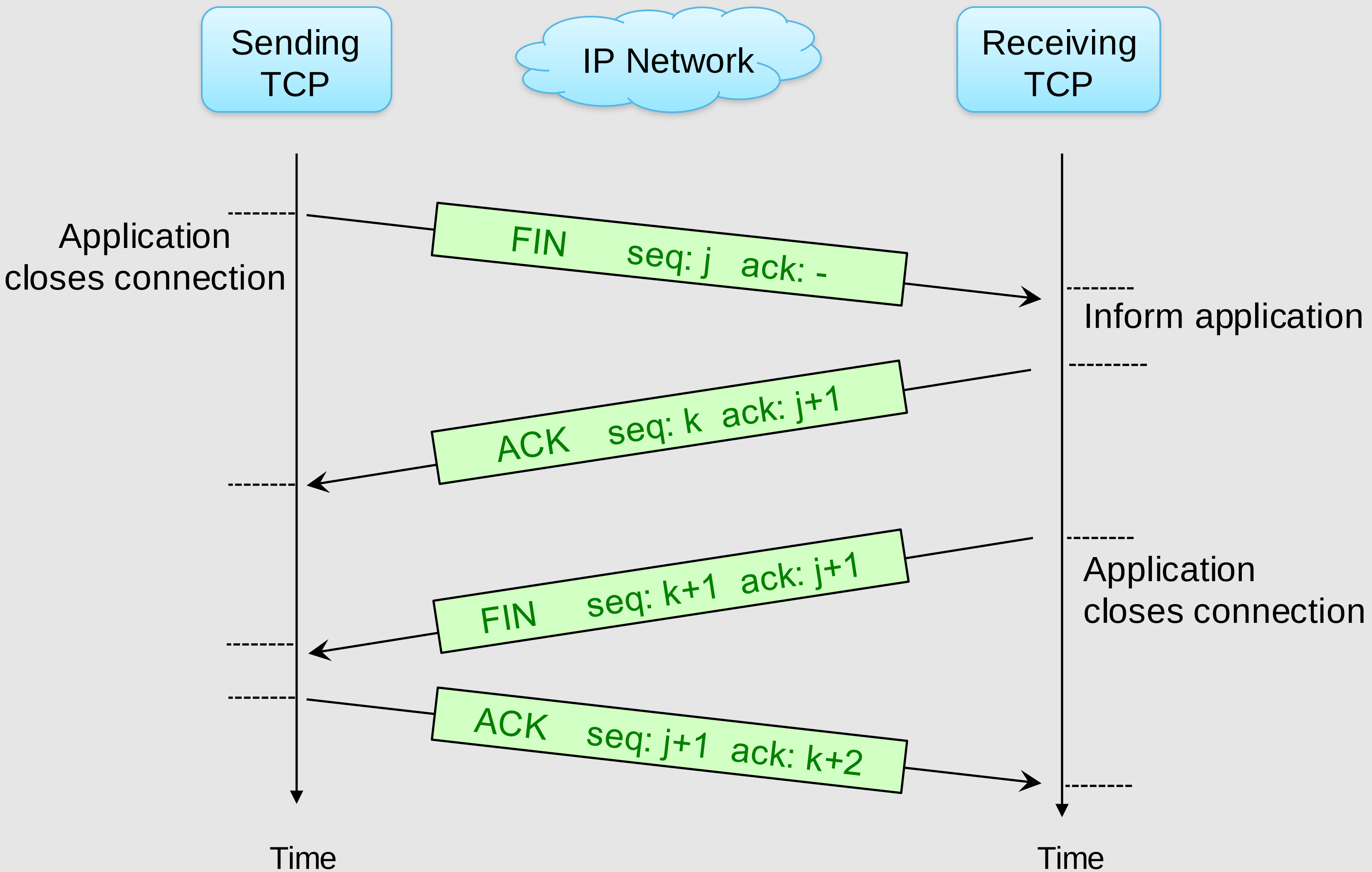
FIN_WAIT_2

TIME_WAIT

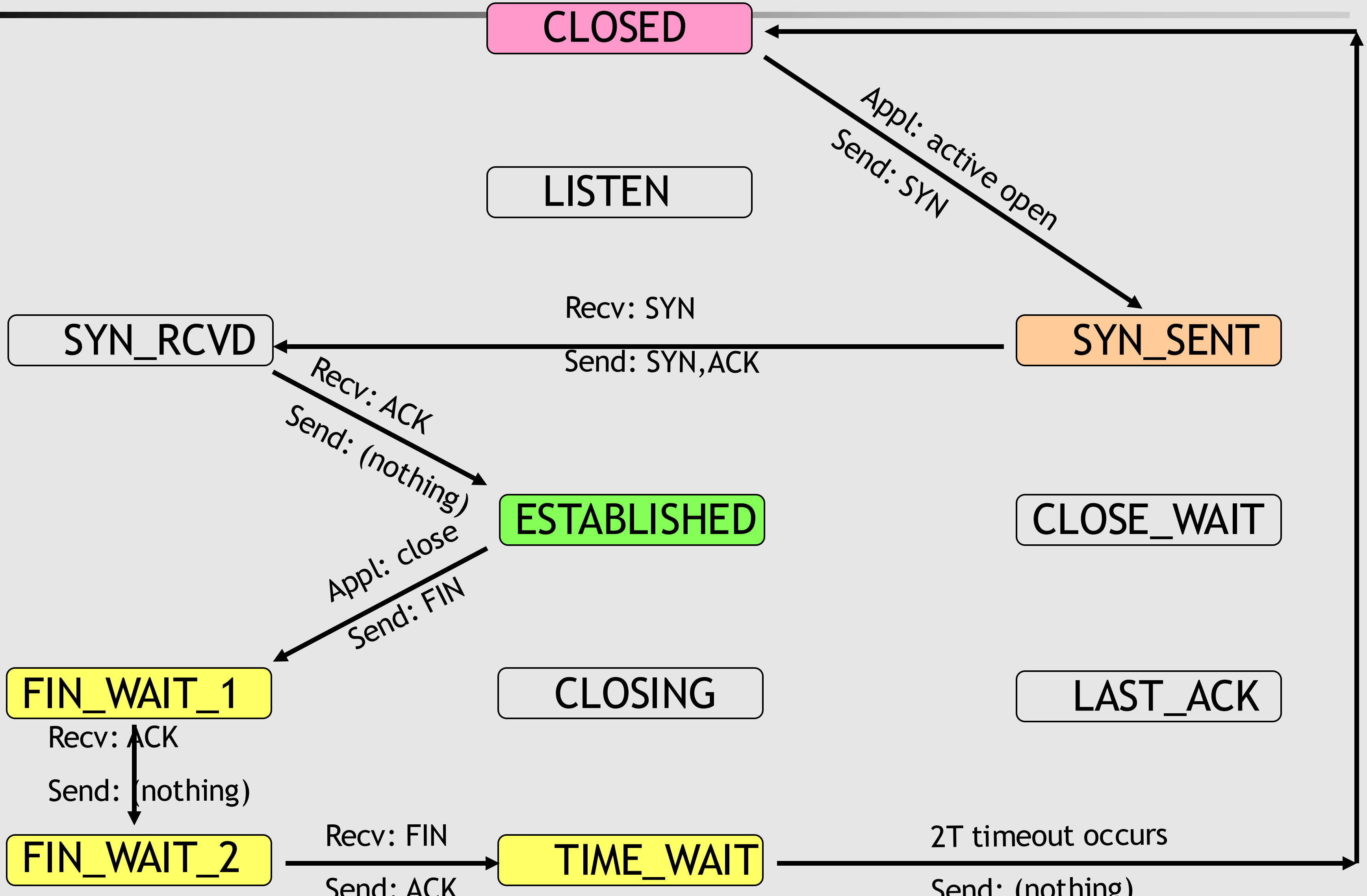
TCP: Normal *Client* Open and Close



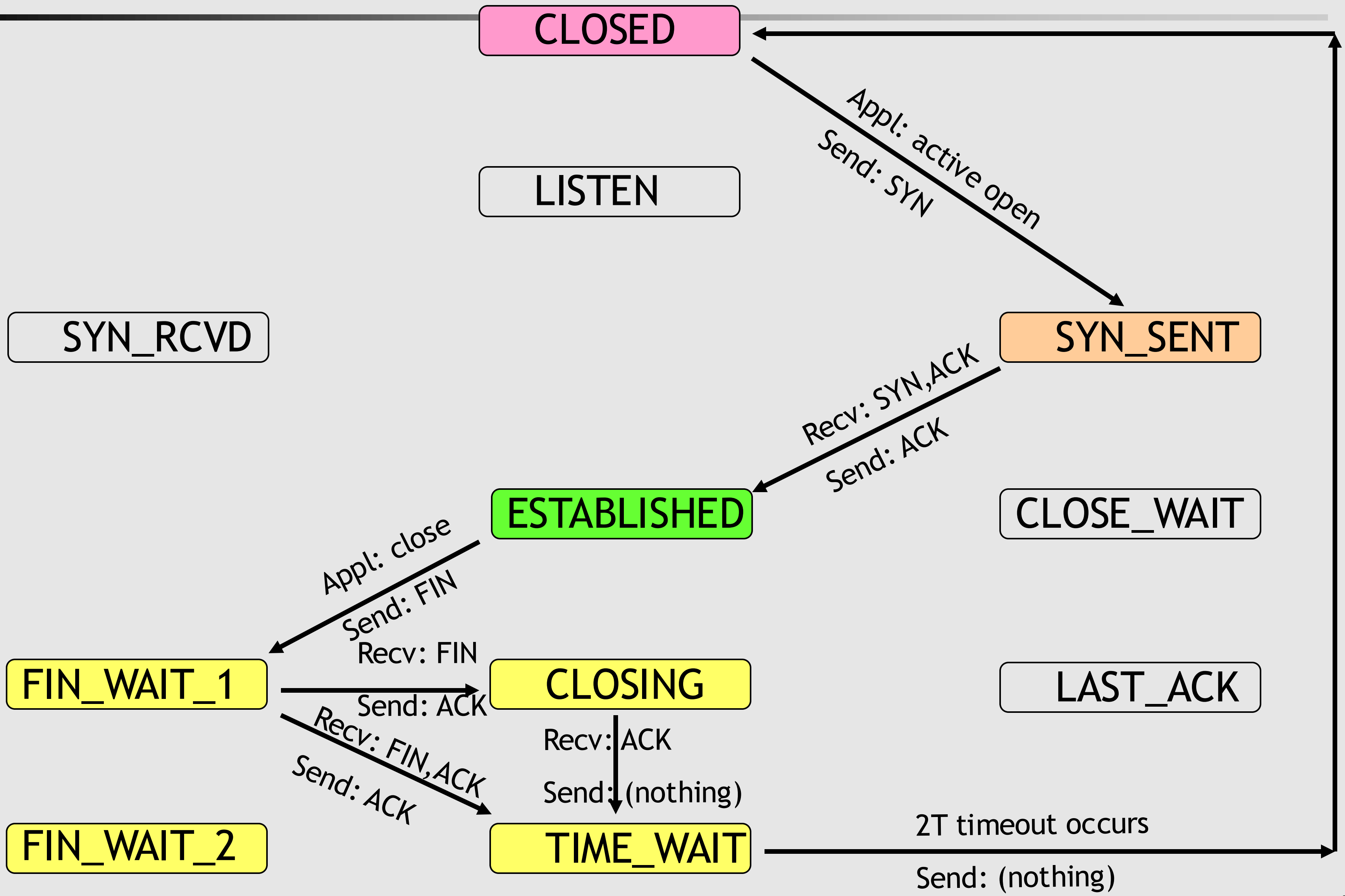
TCP Connection Termination



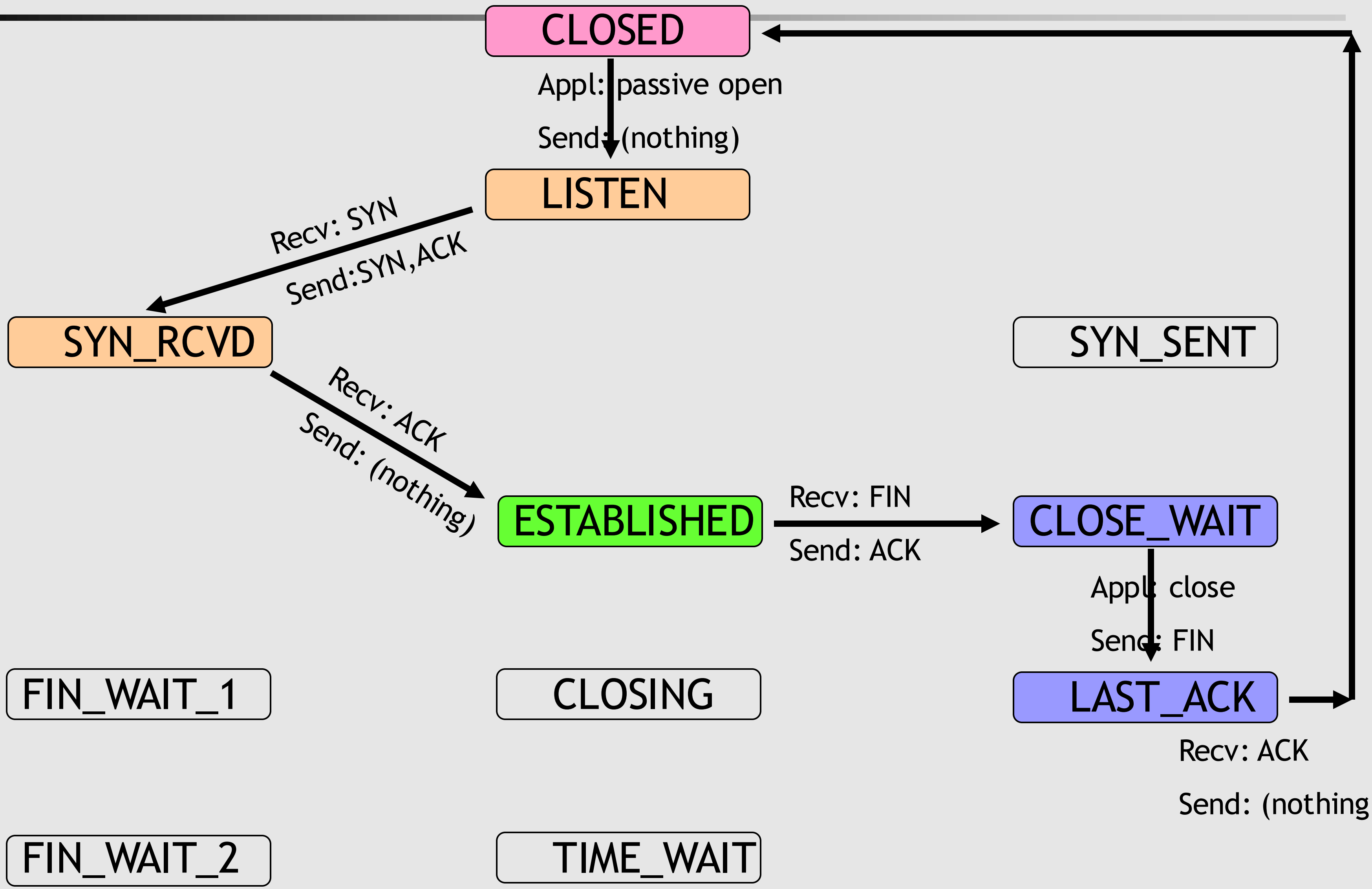
TCP: Simultaneous Open



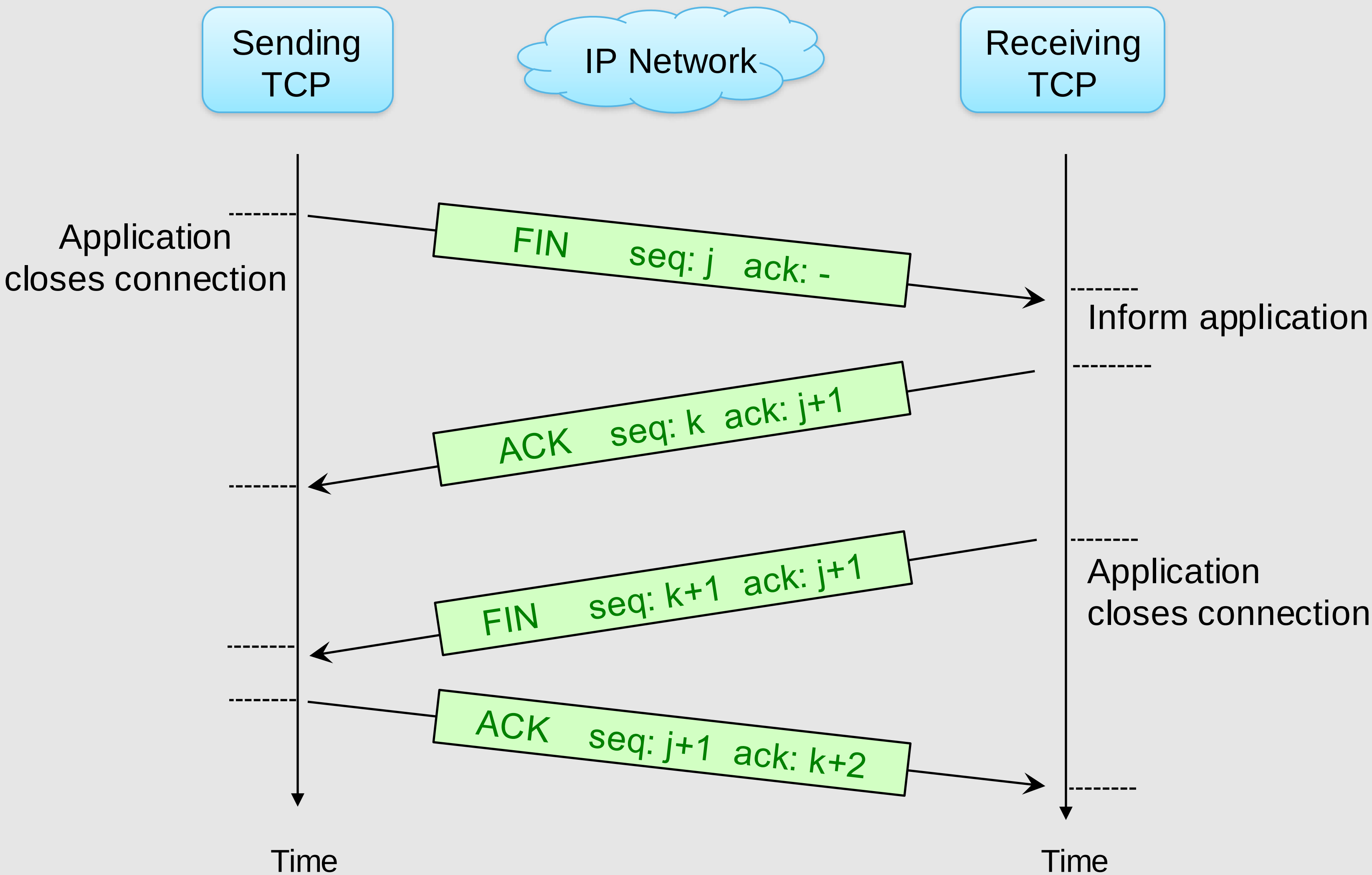
TCP: Simultaneous Close



TCP: Normal *Server* Open and Close



TCP Connection Termination



TCP State Machine

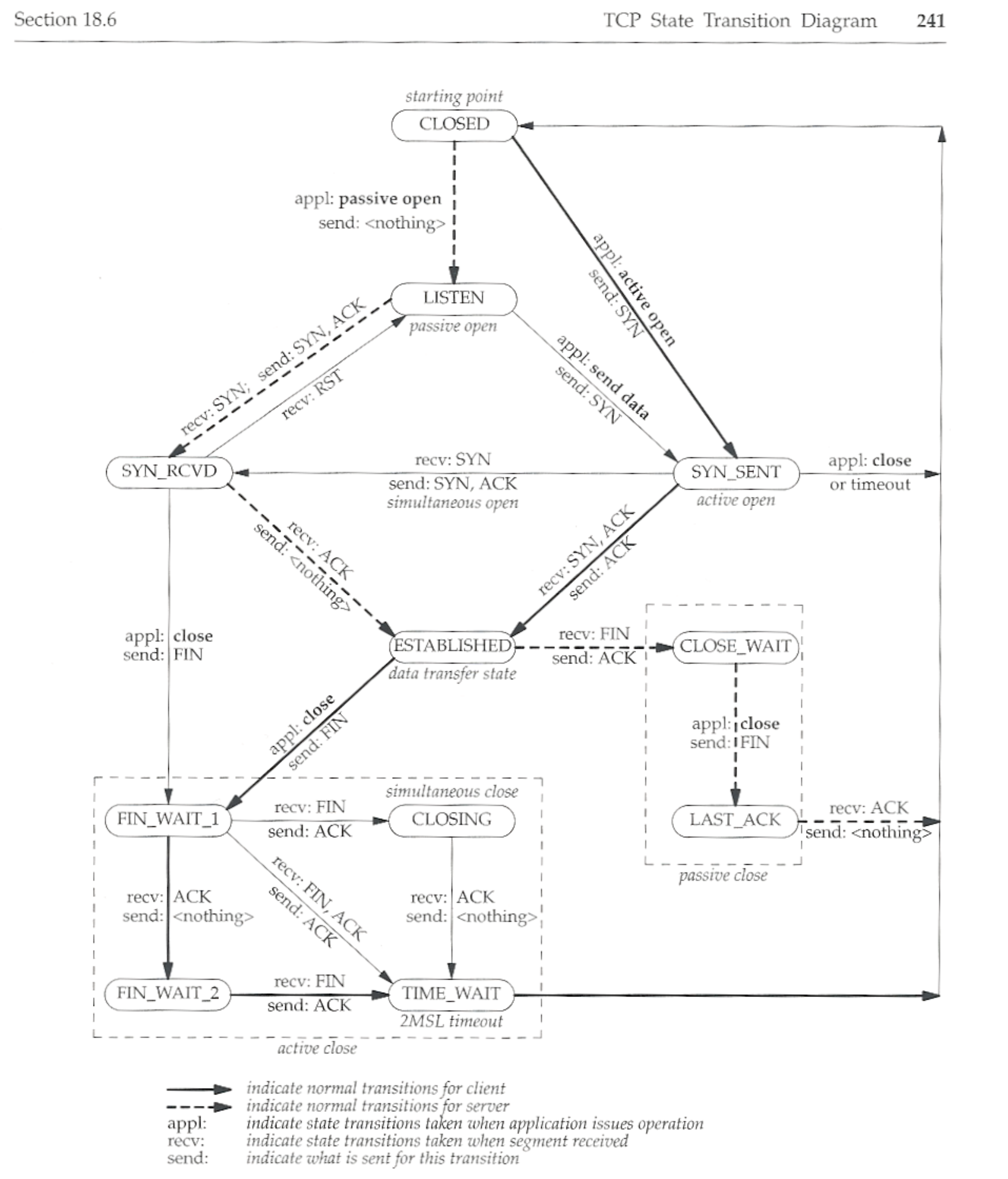


Figure 18.12 TCP state transition diagram.

- Source:
  W. Richard Stevens

Review: TCP Connection Management 1

Review: TCP Connection Management 1

Why is the TIME_WAIT state needed in the TCP state machine?

A	To ensure connections are opened on both hosts before real communications happen
B	To ensure connections are closed on both hosts before accepting another connection
C	To ensure packets in a closed connection die off before accepting a new connection
D	To ensure both hosts have received ACKs of the FIN packets

Review: TCP Connection Management 2

Review: TCP Connection Management 2

In TCP, why does connection termination need one more "way" then connection establishment?

A	Because one of the hosts may crash
B	Because there may be duplicate FIN/ACK packets
C	Because TCP needs to prevent a new connection from reusing the terminating one's sequence numbers
D	Because when one host finishes sending, the other host may still have data to send

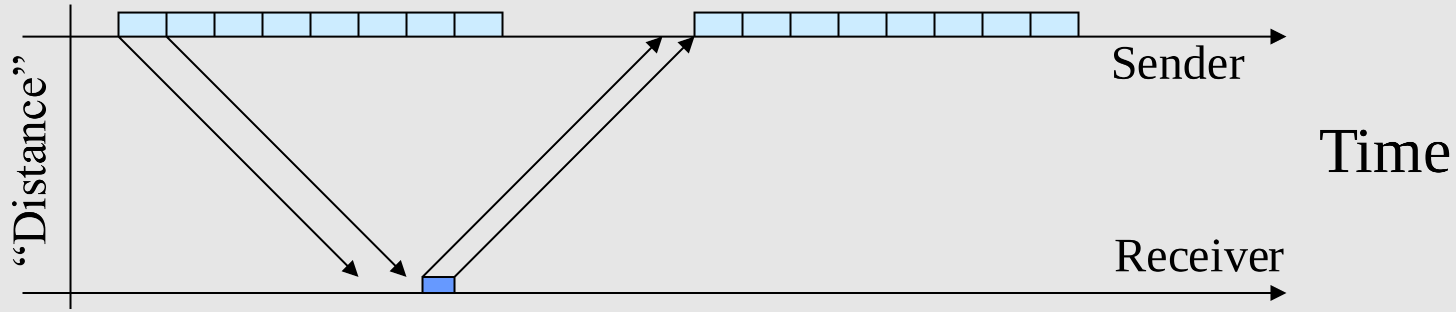
Transmission Control Protocol

Data Transfer

Data Transfer in TCP

- Data received from an application usually sent in segments of size MSS (Maximum Segment Size)
- ACKs carry the sequence number of the *next* byte receiver expects to receive
- *Unacknowledged data* = data sent by sender, but not yet acknowledged by the receiver
  - Data not yet received, or acknowledgment not yet received
- Sender is allowed to send a certain amount of unacknowledged data
  - Must be able to store, in case retransmission required
  - But also limited by storage capacity at receiving TCP

TCP Windowing Mechanism



- Like GBN, but some differences:

- Sender’s buffer size not sole consideration for maximum outstanding data – receiver’s buffer size also considered
  - Each ACK carries a *window advertisement* from receiver
  - Window: how many additional bytes (after last ACK’d byte) the receiver is prepared to accept
    - After sending, the sender must stop and wait for an acknowledgment
    - Even if buffer is available for sending TCP
  - Allows adjustment to “window size” to approach “full window”

Sliding Windows

- TCP *sliding window* mechanism allows multiple segments to be sent before any ACK is returned
  - Maintained by sending TCP
  - But attempts to reflect receiving TCP's buffer state also

- Left boundary of window = earliest unacknowledged byte
  - An acknowledgment advances this left boundary
- Right boundary of window = latest byte that can be sent
  - An updated window advertisement advances this right boundary

Sliding Window Protocol: 1 2 3 $\boxed{4\ 5\ 6\ 7\ 8}$ 9 10 11 12 ....

- Subtle consequence: window can shrink or expand

Sender's Sliding Window

Boundary	At byte	Advanced by
Left (earlier)	Earliest unacknowledged	ACK reception
Right (later)	Latest that can be (may have been) sent	Window advertisement

Sending TCP has 15 byte buffer
Receiving TCP has advertised 8 byte window
Sending application has written 10 bytes

Bytes 1 – 8 will be sent by sending TCP, since they can be

16	17	18	4	5	6	7	8	9	10	11	12	13	14	15

Case 1: Pure sliding

An ACK comes in, acknowledging upto byte 3, and same window update
(Receiving TCP has received 3 bytes.
Application has likely consumed them, leaving receiving TCP's buffer with 8 free bytes, as before.)
Bytes 9 – 11 can now be sent, and 9 – 10 *will* be sent by sending TCP
Bytes 1 – 3 can be purged, making room for application to write more bytes
If application now writes more bytes, Byte 11 can and will be sent also

Sender’s Sliding Window

Boundary	At byte	Advanced by
Left (earlier)	Earliest unacknowledged	ACK reception
Right (later)	Latest that can be (may have been) sent	Window advertisement

Sending TCP has 15 byte buffer
Receiving TCP has advertised 8 byte window
Sending application has written 10 bytes

Bytes 1 – 8 will be sent by sending TCP, since they can be

16	17	18	4	5	6	7	8	9	10	11	12	13	14	15

Case 2: Pure filling

An ACK comes in, acknowledging upto byte 3, and window update of 5
(Receiving TCP has received 3 bytes.
Application has likely NOT consumed them, leaving receiving TCP’s buffer with 3 fewer free bytes.)
Bytes 1 – 3 can be purged, making room for application to write more bytes, but:
Nothing more can be sent by sending TCP at this time

Sender's Sliding Window

Boundary	At byte	Advanced by
Left (earlier)	Earliest unacknowledged	ACK reception
Right (later)	Latest that can be (may have been) sent	Window advertisement

Sending TCP has 15 byte buffer
Receiving TCP has advertised 8 byte window
Sending application has written 10 bytes

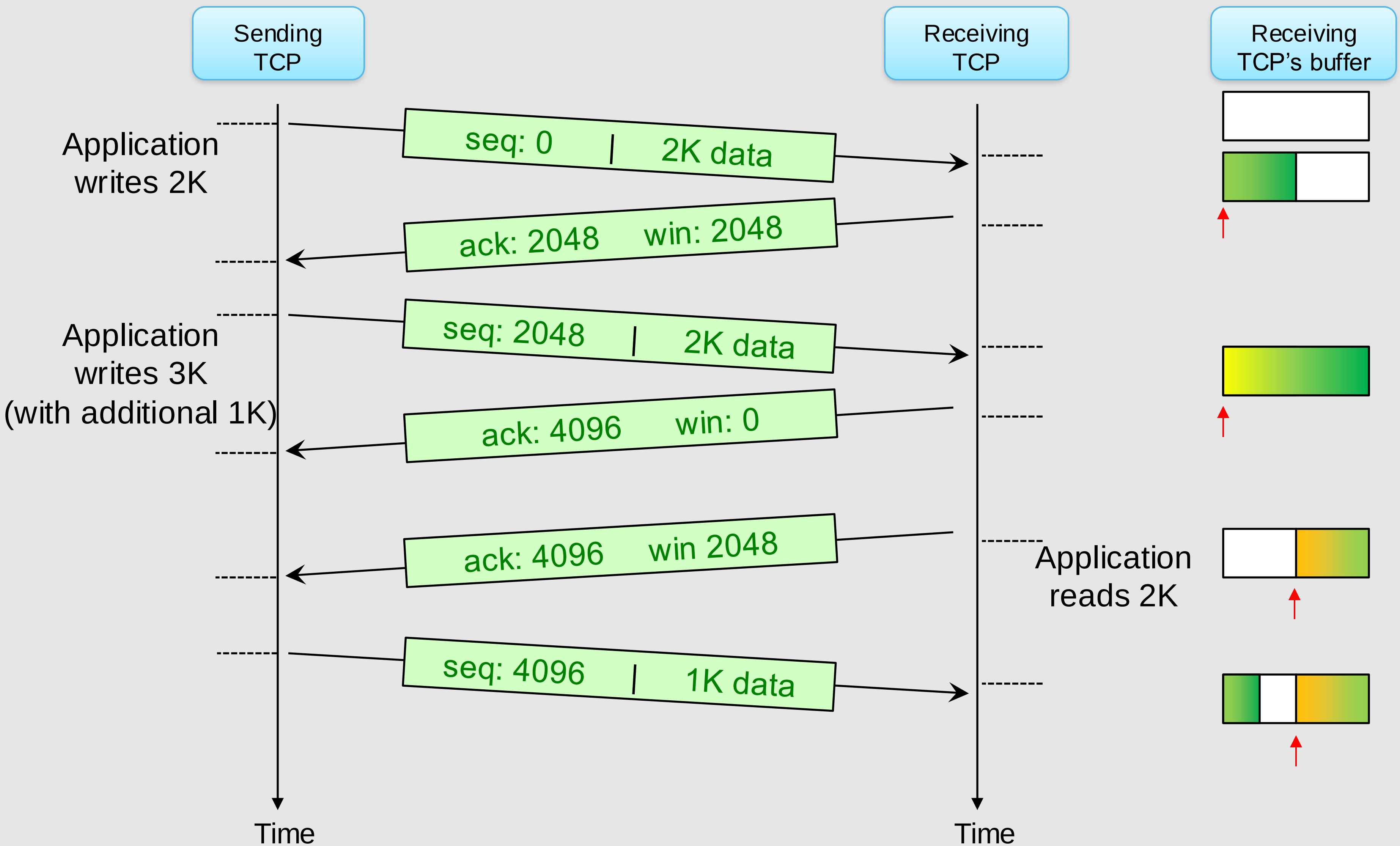
Bytes 1 – 8 will be sent by sending TCP, since they can be

16	17	18	4	5	6	7	8	9	10	11	12	13	14	15

Case 3: Memory reallocation (shrinking window)

An ACK comes in, acknowledging upto byte 3, and window update of 3
(Receiving TCP has received 3 bytes.
Application has likely NOT consumed them, and receiving TCP's buffer allocation has been reduced.)
Bytes 1 – 3 can be purged, making room for application to write more bytes
Nothing more can be sent by sending TCP at this time, AND:
Bytes 7 – 8 are retroactively considered "un-sent"
(Will be sent again when window slides forward over those bytes again.)

TCP Window Management Example



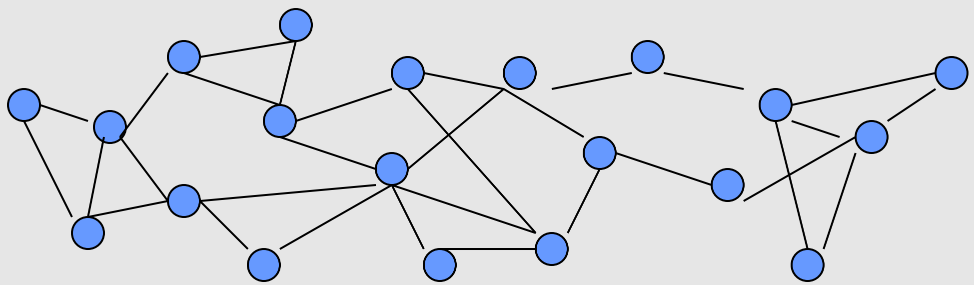
Performance Issues

- Small packets and small window create efficiency problems
- These can solved by
  - Delaying sending of data
    - sender “voluntarily” consolidates multiple small packets into a single larger packet
  - Delaying sending of ACKs/window advertisements
    - sender “strongly encouraged” to consolidate multiple small packets into a single larger packet
- TCP “probes” the network for bandwidth (later)
  - Attempts to adjust throughput to available b/w
  - This involves latency, and embeds assumptions

TCP Error Control

1. Error *detection*
   - *checksum*: to check for corrupted segments at destination
   - *ACK*: to confirm receipt of segment by destination
   - *time-out*: one retransmission timer for each segment sent
2. Error *correction*
   - source retransmits segments for which retransmission timer expired

Congestion Control



- Congestion - overloaded network
  - Applies to part of the network - the “pain point” or “bottleneck”
    - Some link(s) in the network being used by a disproportionately large amount of traffic
    - Some node(s) in the network at the head of many highly loaded link
- Congestion is undesirable - wastes network resources for no gain, causes oscillations
  - Can try adaptively re-routing traffic, or notifying sources to slow down

TCP Implicit Feedback for CC

- TCP relies on implicit feedback from the network to detect congestion
  - Timeout caused by a lost packet (used by TCP-Tahoe, TCP-Reno)
  - Duplicate ACK (used by TCP-Reno)
- Assumption: packet loss is always due to congested routers, not transmission errors
  - As wired transmission technology had grown more sophisticated, reasonable assumption
- When congestion is detected, slow down rate
  - Mechanism - make window smaller
  - smaller window = lower rate, larger window = higher rate

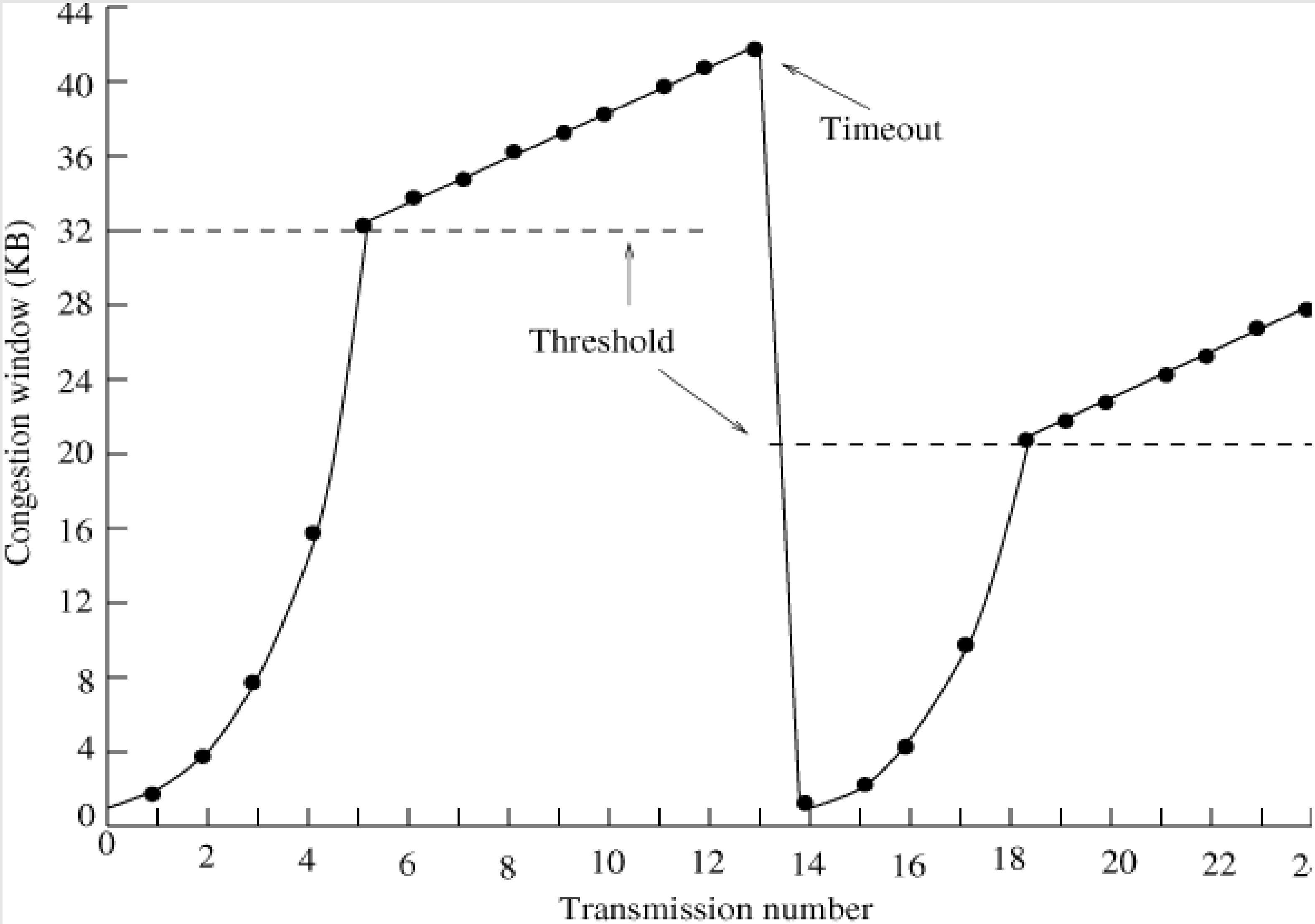
TCP Dynamic Windows

- Window #1: advertised by receiver
  - purpose: avoid overrunning a slow receiver (i.e., for flow control)
  - Called the receiver window, or `rwnd`
- Window #2: maintained by sender
  - purpose: avoid network overload (i.e. for congestion control)
  - Called the congestion window, or `cwnd`
  - the sending TCP dynamically manipulates `cwnd`
- “The” window size = MIN(`rwnd, cwnd`)

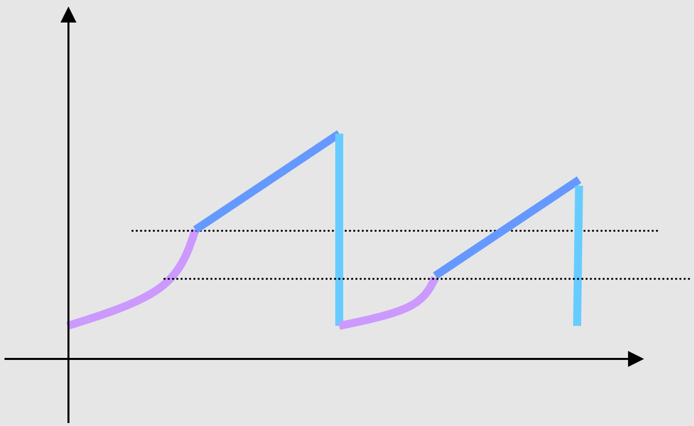
TCP Congestion Control Overview

- Probe for available bandwidth by increasing `cwnd`
  - "slow" start = initially window is small
    - But every ACK adds 1 MSS to window
  - *exponential* increase phase - every RTT doubles window
- Congestion avoidance = *linear* increase phase
  - Slow start threshold (= `ssthresh`)
  - controls transition from exponential to linear phase
- Upon packet loss (timeout, duplicate acknowledgment), assume congestion
  - retransmit the packet
  - reduce the window size - reduce rate
  - Also re-start probing

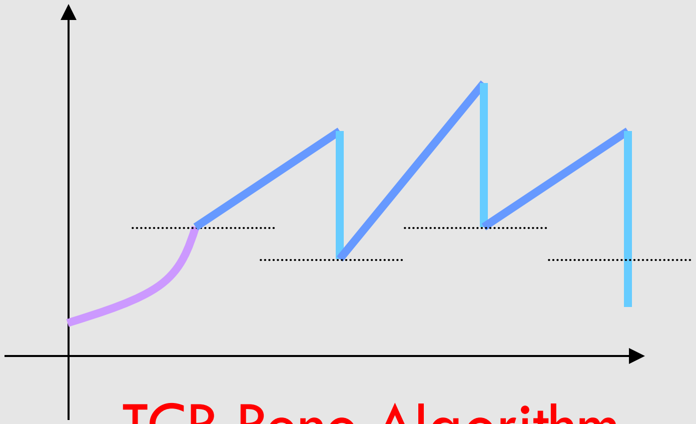
Evolution of TCP's Congestion Window (Taho)



Congestion Control



Alternative: Fall to W/2 (+n*MSS) and start congestion avoidance directly



TCP Reno Algorithm

- Slow Start
  - Start with W=1 (TCP seg)
  - For every ACK, W=W*2
- Congestion Avoidance (linear/additive increase)
  - For every ACK,
  - W = W+1
- Congestion Control (multiplicative decrease)
  - ssthresh = W/2
  - W = 1

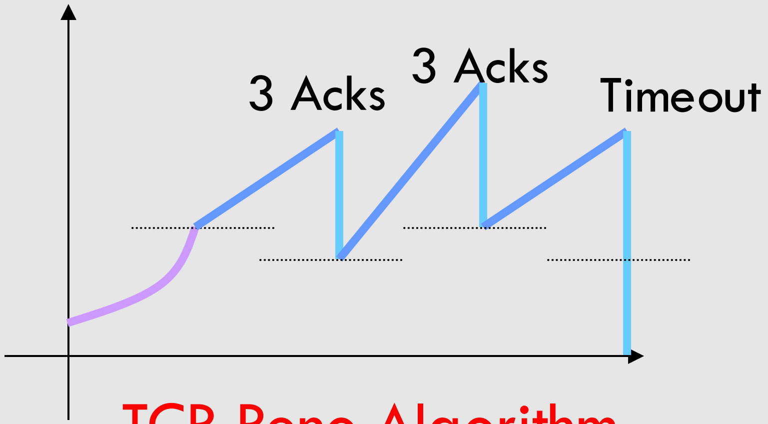
TCP Tahoe Algorithm

Congestion Control

Alternative: Fall to W/2 (+n*MSS)
and start congestion avoidance directly

If (Timeout)
W = 1
ssThresh = w/2
slow start …

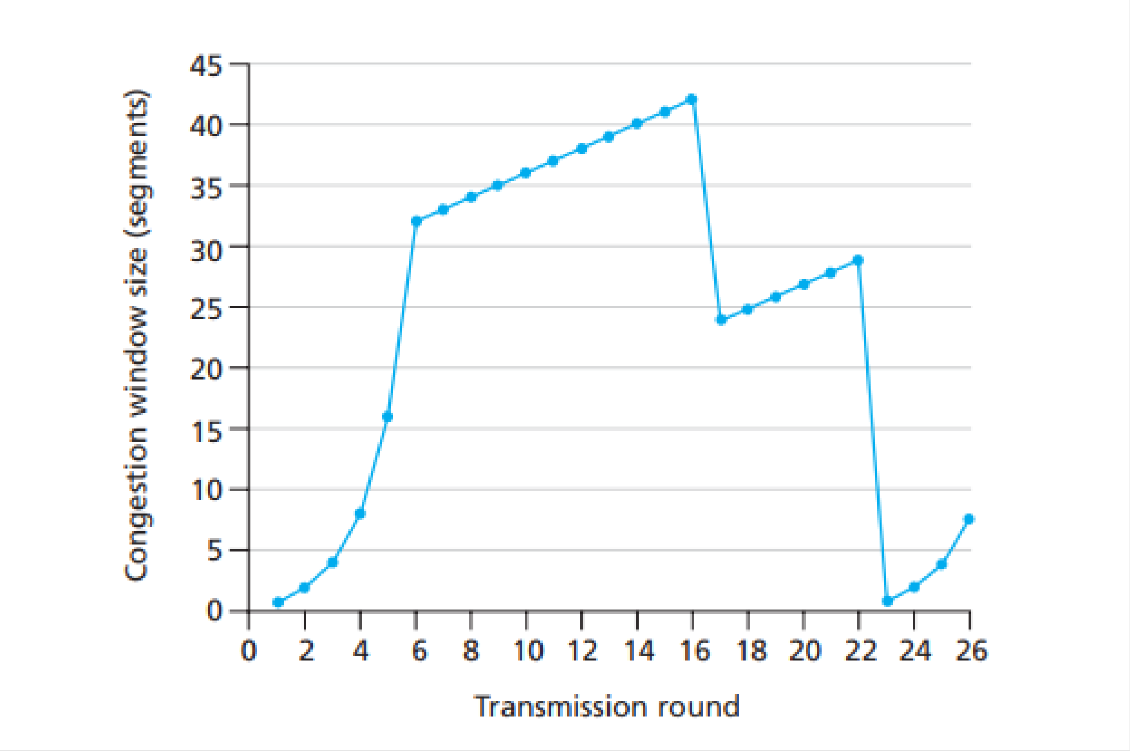
If (3 Acks)
W = w/2
linear/additive increase …



TCP Reno Algorithm

Congestion Control

Alternative: Fall to W/2 (+n*MSS)
and start congestion avoidance directly



History of TCP