Computer Networks
Chapter 3.5

Jong-won Lee

Handong University

Chapter 3

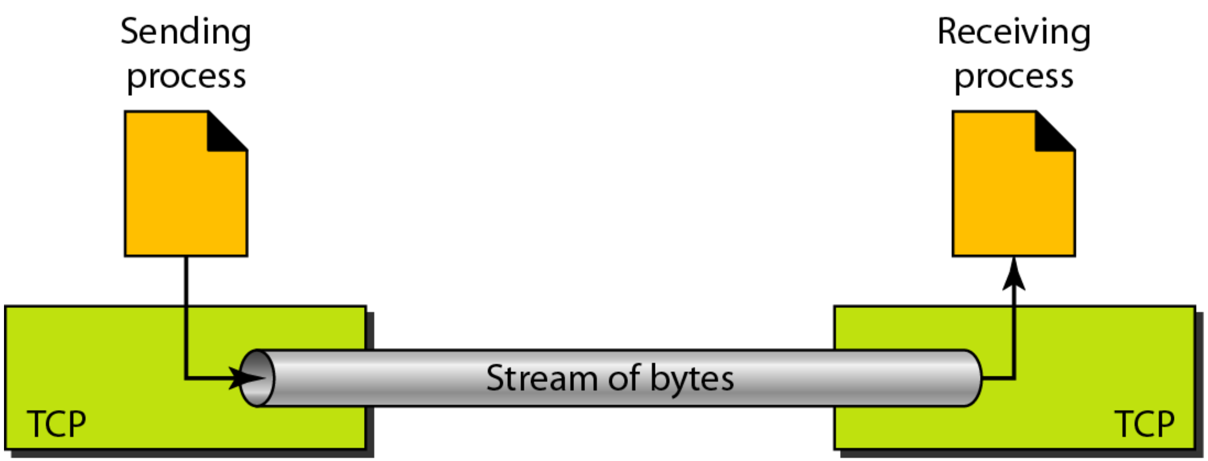
- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer (ARQ)
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - RTT estimation and timeout
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

TCP: Overview RFCs: 793, 1122, 1323, 2018, 2581

- connection-oriented:
  - (exchange of control msgs) init's sender, receiver state before data exchange
- point-to-point:
- full duplex data:
  - bi-directional data flow in same connection
  - MSS:
    - The max. amount of application data in a segment

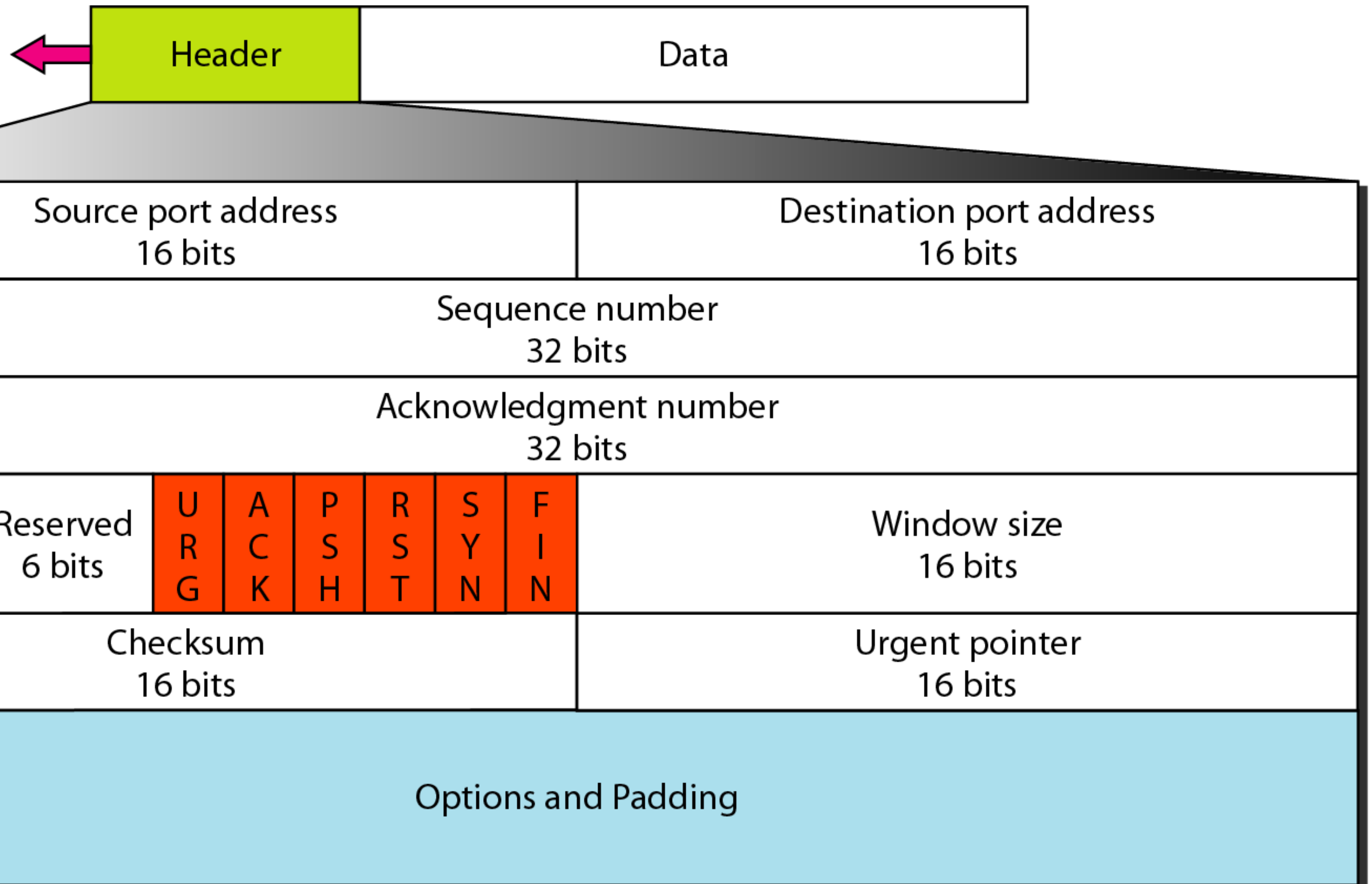
TCP: Overview RFCs: 793, 1122, 1323, 2018, 2581

-
  - Sliding window protocol
  -
  -
- Flow control
  - sender will not overwhelm receiver buffer
- Congestion control



TCP Segment Structure

- Header = 20 bytes + options



TCP Header

- *Source/destination port (16 bits)*
  - Identifies the application process
- *Sequence number (32 bits)*
  - Sequence number of the first byte in the segment.
    - Seq. number is advanced
  - If SYN is present, this is the initial sequence number (ISN) and the first data byte will be ISN+1.
  - during the connection establishment, each party uses a random number generator to create an *initial sequence number (ISN)*
- *Acknowledgement number (32 bits)*
  - Next expected sequence number, valid only when the ACK bit (reside in flag) is set
  - Cumulative ACK

TCP seq. #'s and ACKs

Seq. #'s:

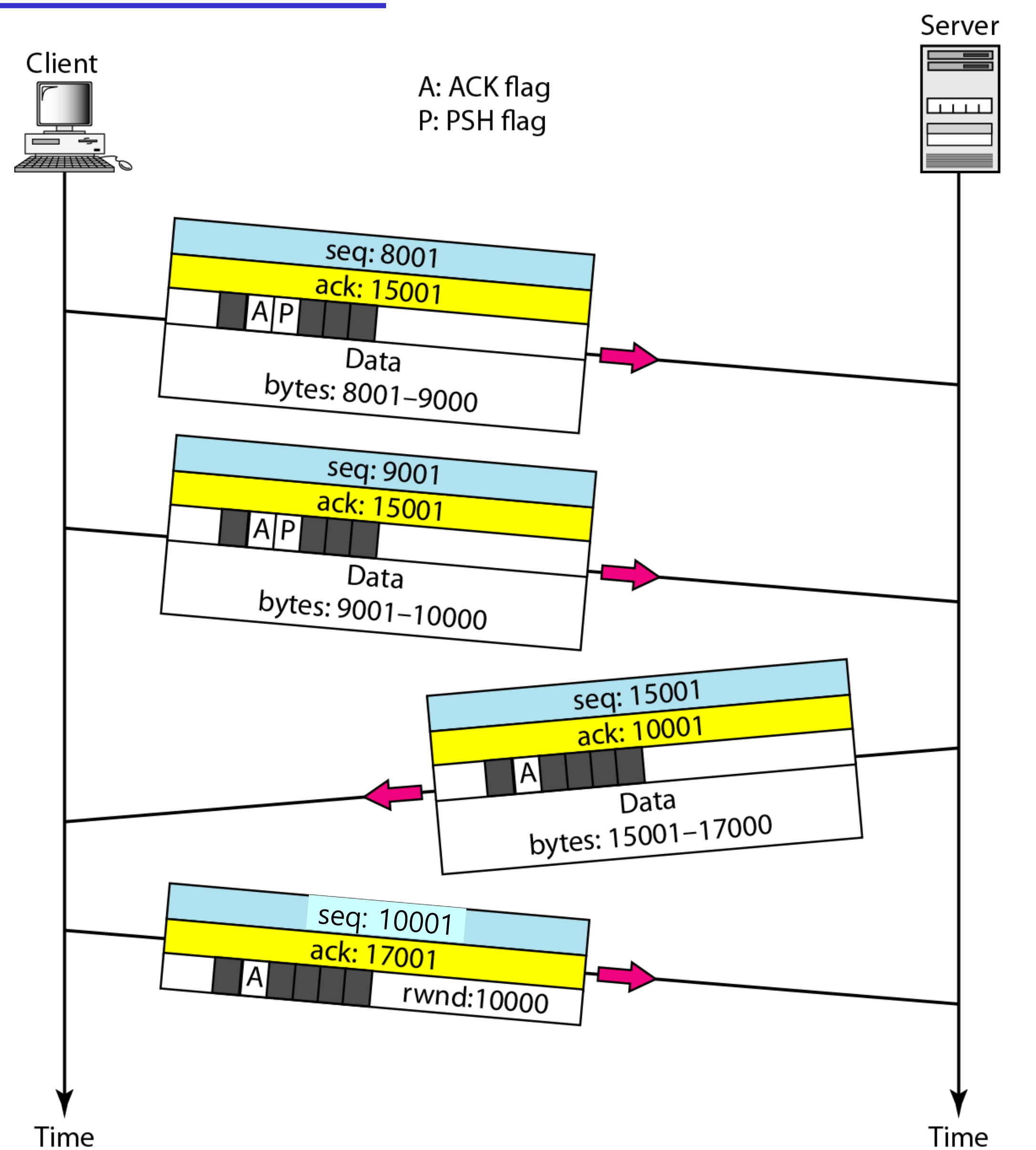
- byte stream "number" of first byte in segment's data

ACKs:

- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

- A: TCP spec doesn't say, - up to implementor

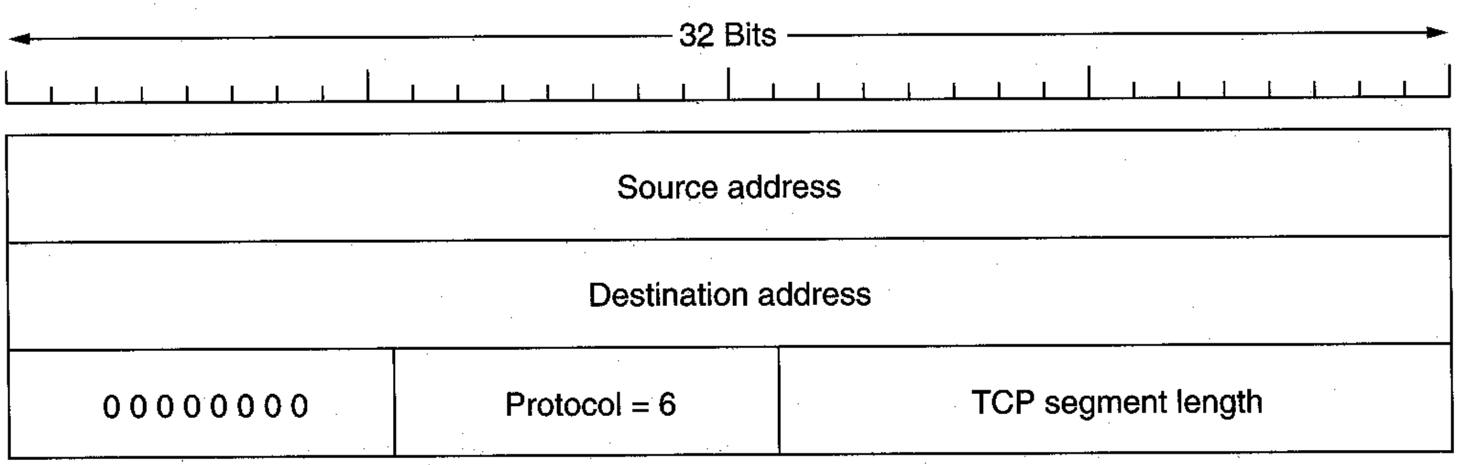


TCP Header

- 6 control flag bits
  - URG : set to 1 if *urgent pointer* is in use.
  - : set to 1 if *Acknowledgement number* is valid
  - PSH : set to 1 if the receiver should pass this data to the application as soon as possible.
  - RST : set to 1 to reset a erroneous connection
  - : synchronize sequence numbers to initiate a connection
    - If SYN=1 & ACK=0,
      then this segment is a ***CONNECTION-REQUEST***
    - If SYN=1 & ACK=1,
      then this segment is a ***CONNECTION-ACCEPTED***
  - : set to 1 to indicate no more data to send

TCP Header

- *TCP header length (4 bits)*
  - the length of the header in 32-bit words
    - 4 bits => max header size is 60 bytes
  - needed because the options field is of variable length
- *Window size (16 bits)*
  - will accept [Ack] to [Ack]+[window-1]
- *Checksum*
  - Internet checksum of *the header , the data, and the pseudo-header*
  - The pseudo-header is not transmitted.



TCP Header

- *Urgent pointer (16 bits)*
  - valid only if the *URG* flag is set
  - Lets receiver know how much data it should deliver right away.
  - Points to the last byte of the urgent data. (urgent data is at the beginning of the segment)

- *Options (variable)*
  - MSS (Maximum Segment Size), window scale factor, timestamp etc.

RTT Estimation and Timeout

- Q: How to set TCP timeout value?
  - longer than RTT
    - but large variance of RTT
  - too short: premature timeout
    - unnecessary retransmissions
  - too long: slow reaction to segment loss
- A highly dynamic algorithm is needed
  - adjusts the timeout interval based on continuous measurements of network performance

RTT Estimation and Timeout

- Original RTT estimation method (RFC 793)

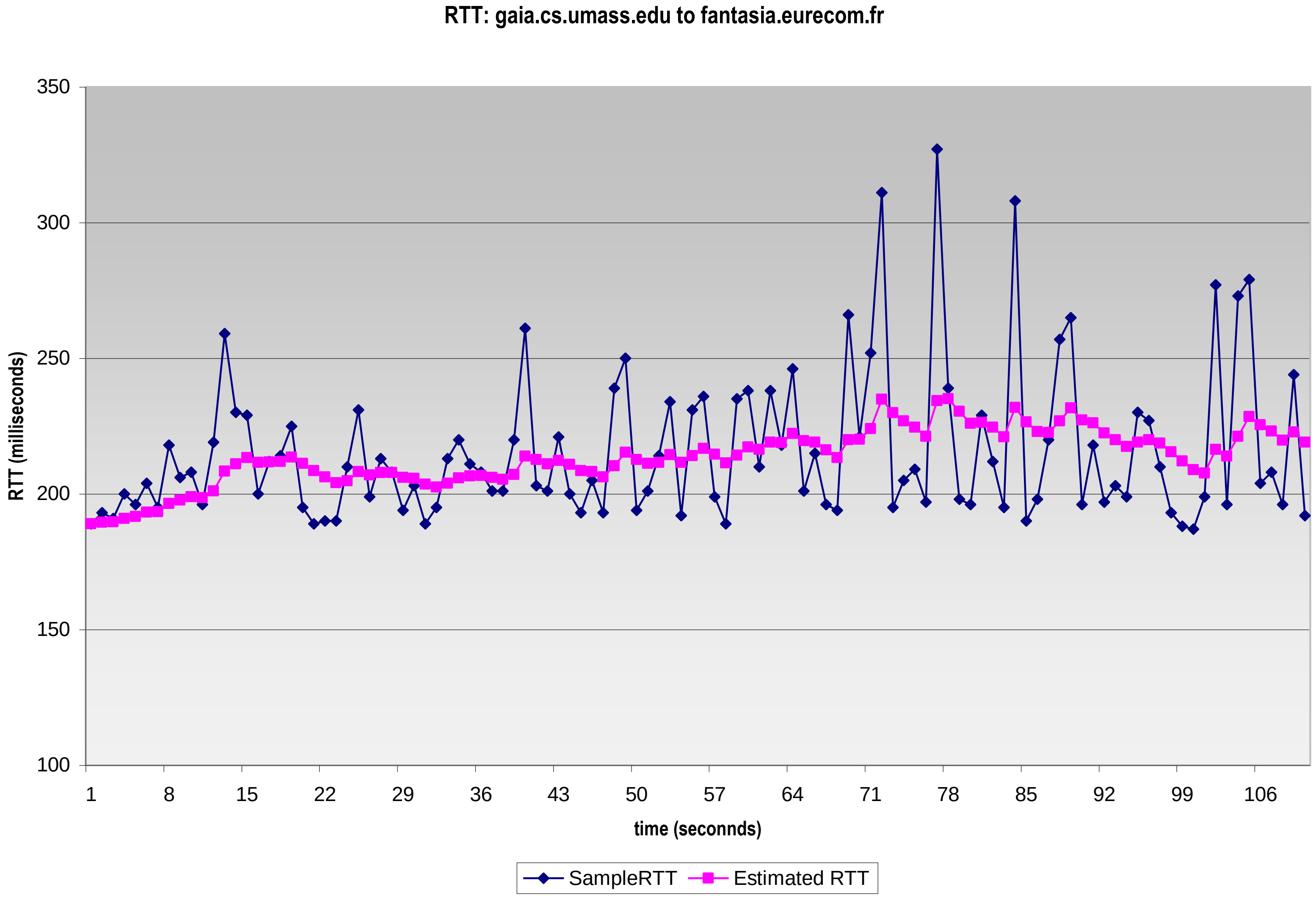
  (Exponential weighted moving average)
    - influence of past sample decreases exponentially fast

  - EstimatedRTT =
    - (typically α = 7/8)
    - **`SampleRTT`**: measured time from segment transmission until ACK receipt

  - RTO =

  - Too large RTO value

Example: RTT estimation:



RTT Estimation and Timeout

- Improved algorithm (Jacobson)
  - β is roughly proportional to the standard deviation
  - use a mean deviation D as an estimator of standard deviation

    $(\Sigma |M_i - a|)^2 >= \Sigma |M_i - a|^2$

  - Err = SampleRTT - EstimatedRTT

    EstimatedRTT = EstimatedRTT + g * Err

    (typically g =1/8(=1-α))

    DevRTT = DevRTT + h * (|Err| - DevRTT)

    (typically h =1/4)

    RTO = EstimatedRTT + 4 * DevRTT

RTT Estimation and Timeout

- Initial value for RTO:
  - Sender should set the initial value of RTO to

    RTO_0 = 3 seconds

- RTO calculation after first RTT measurements arrived

  $EstimatedRTT_1$ = RTT
  $DevRTT_1$ = RTT / 2
  $RTO_1 = EstimatedRTT_1 + 4 * DevRTT_1$

- When a timeout occurs, the RTO value is doubled

  $RTO_{n+1} = \min(q * RTO_n, RTO_{max})$ seconds
  (typically q = 2, RTO_{max} >= 60 sec.)

TCP reliable data transfer

- Based on sliding window protocol
  - Similar to SR ARQ.
  - Dynamic window size
    - Combined with flow control & congestion control
  - Use cumulative ACK.
  - The value of the acknowledgment field is the number of the next byte that the receiver expects to receive.
  - Retransmissions are triggered by:
    - timeout events
    - Three duplicate ACKs (fast retransmission)

Fast Retransmit

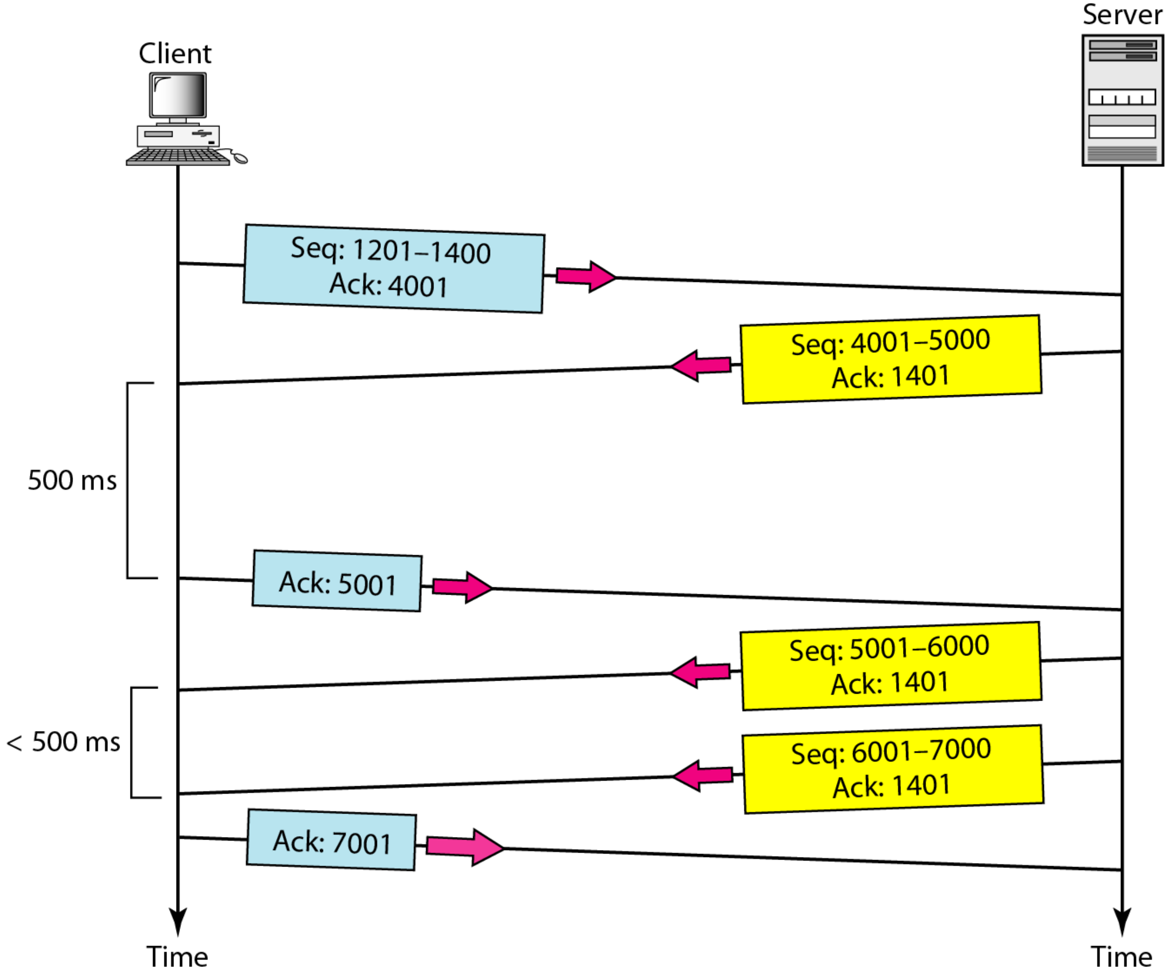
- Time-out period often relatively long:
  - long delay before resending lost packet
- Detect lost segments via duplicate ACKs.
  - Sender often sends many segments back-to-back
  - If segment is lost, there will likely be many duplicate ACKs.
- If sender receives for the same data, it supposes that segment was lost:
  - resend segment before timer expires

TCP ACK generation [RFC 1122, RFC 2581]

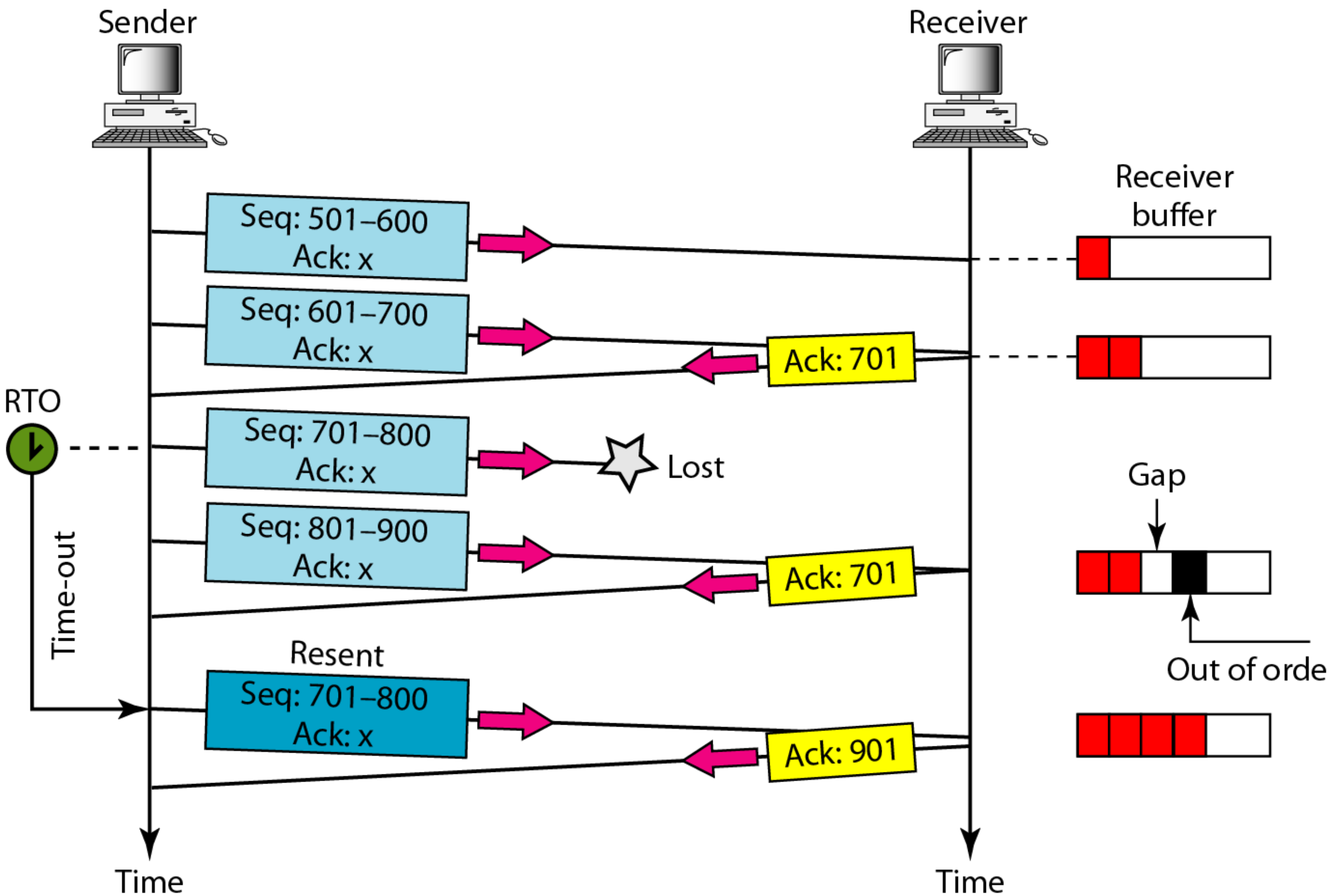
Event at Receiver	TCP Receiver action
Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	Delayed ACK. (within 500ms, typical value=100ms) If no next segment, send ACK
Arrival of in-order segment with expected seq #. One other segment has ACK pending	Immediately 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

Delayed ACK: piggyback an ACK on data

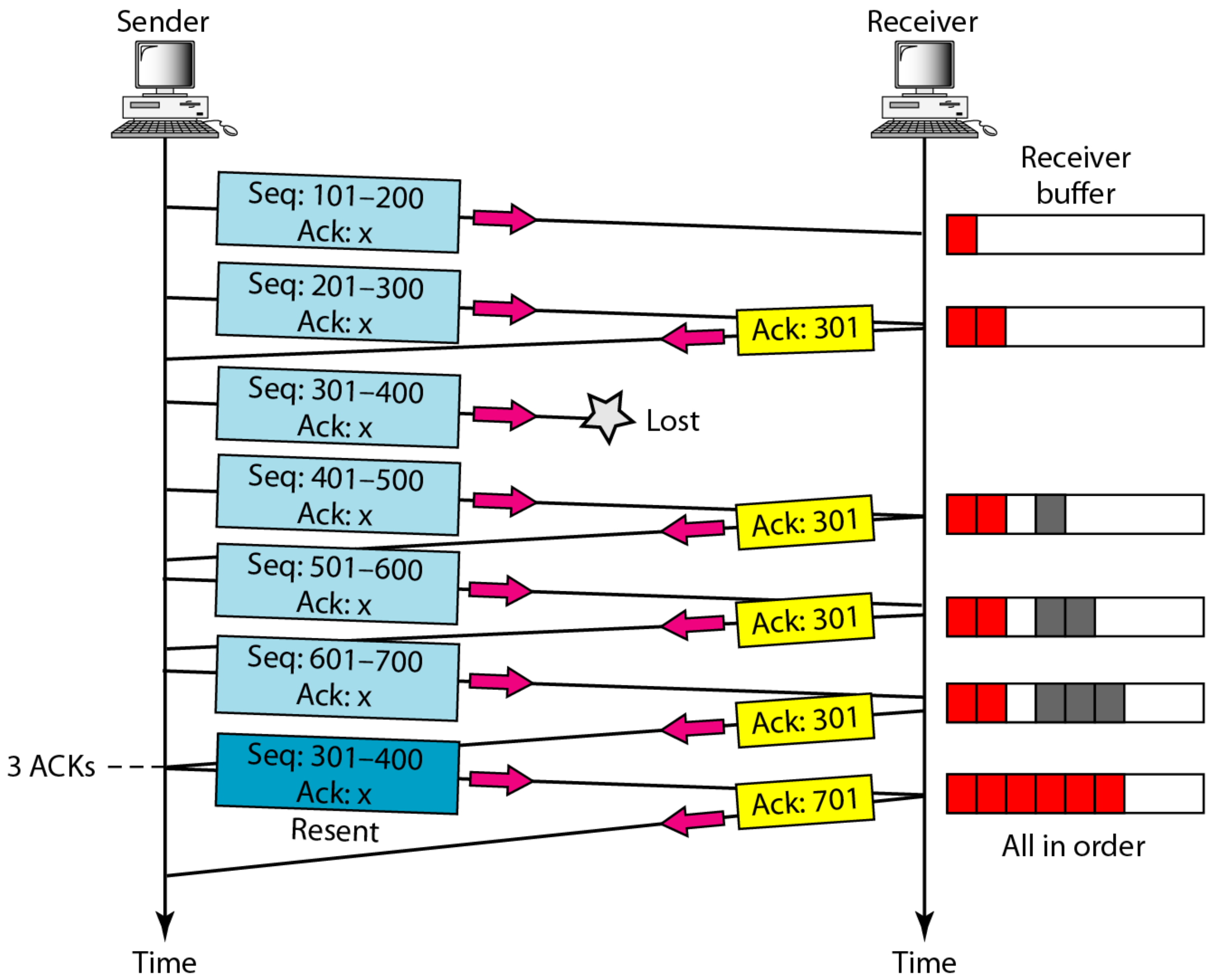
TCP Scenario: Normal case



TCP Scenario: Lost Segment



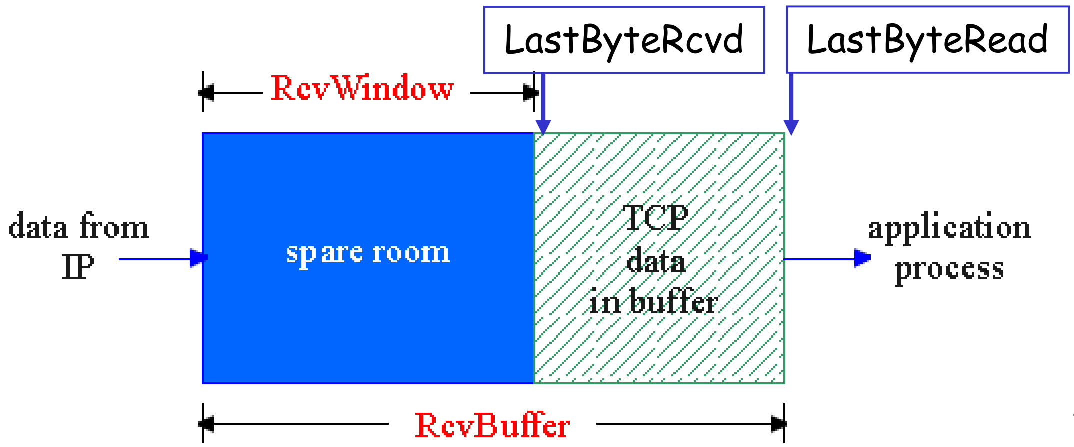
TCP Scenario: Fast Retransmission



TCP Flow Control

- To prevent the sender from overflowing the receiver's buffer.
- Dynamic window management
  - Receiver " advertises" it's available buffer size in the "window size" field of ACK segments.
  - RcvWindow(or AdvertisedWindow) =

    RcvBuffer - (LastByteRcvd - LastByteRead)
  - Sender limits unACKed data to **`RcvWindow`**
    - guarantees receive buffer doesn't overflow



TCP Flow Control

- If receiver buffer is totally full, then RcvWindow = 0
- Sender must entirely close its sender window and transmission may be halted forever!
  - Because TCP does not acknowledge ACK.
    - If the pcaket that opens the widow is lost, the deadlock occurs!
- How to escape from this deadlock ?

TCP: Transmission Policy

- Sender Buffering
  - 'Tinygram' wastes bandwidth
    - a keystroke in telnet session = 41 byte
      (40 byte header + 1 byte data)
  - be able to reduce header overhead by grouping many small data segments into one large TCP segment.

  - algorithm (RFC 896)

  when data come into the sender one byte, send the first byte. Then
    1) buffer all the rest until the outstanding byte is ACKed.
    2) Send a new packet if data fill the half the window or a maximum segment
    - better to be disabled if used on mouse movements.

TCP: Timers

- To perform its operation smoothly, TCP uses the 4 timers
  - Retransmission timer
  - Persistent timer
  - Keep-alive timer
  - 2MSL (time-waited) timer

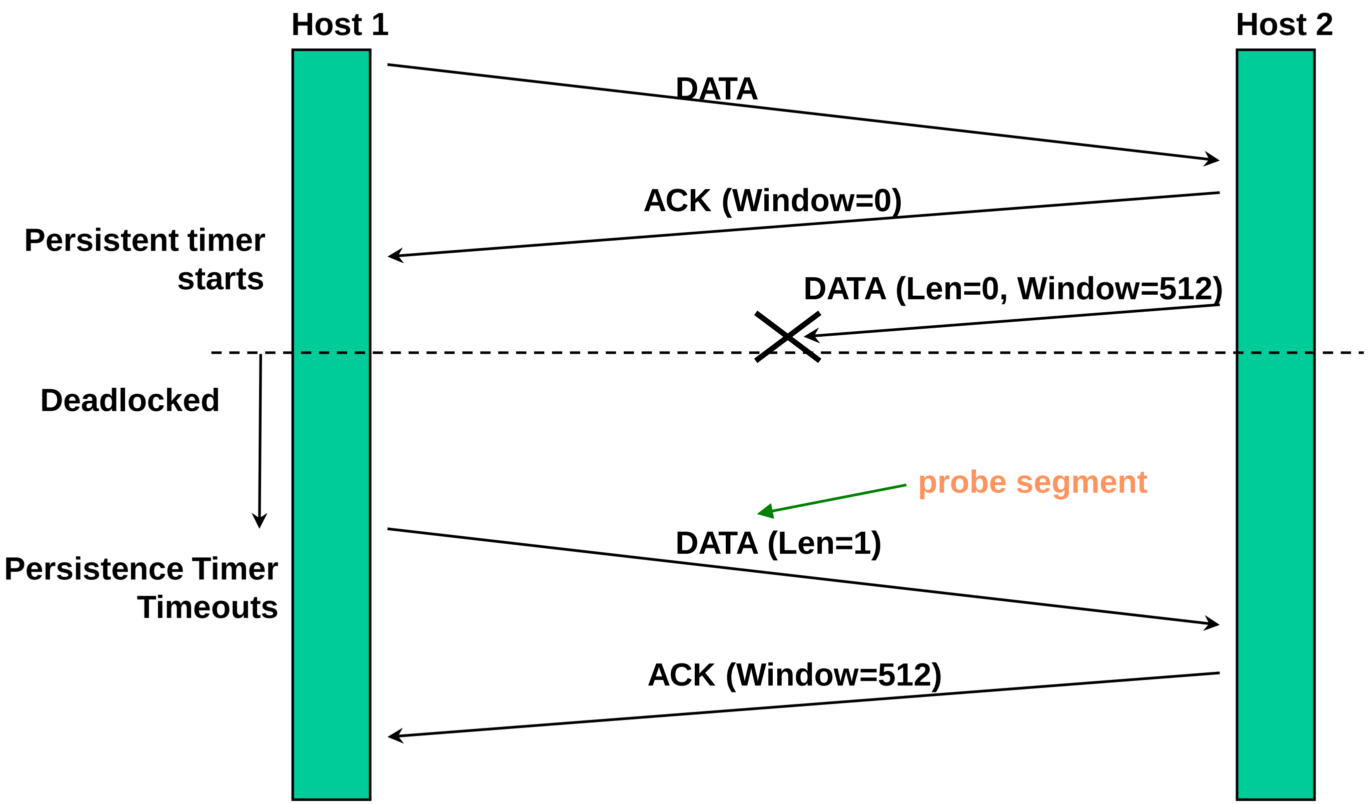
- Retransmission timer
  - Usually TCP sender maintains one retransmission timer for each connection

TCP: Timers

- Persistent timer
  - prevents deadlock from occurring when the packet with the update from the window size of 0 is lost.
  - When the sending TCP receives an acknowledgment with a window size of zero, the persistence timer is started
  - When persistence timer goes off, the sending TCP sends a special segment called a *probe segment*
  - The probe alerts the receiving TCP that the acknowledgment was lost and should be resent.
  - initially persistent timer=2*RTT;
  - If a response is not received, the sender continues sending the probe segments and doubling and resetting the value of the persistence timer until the value reaches a threshold (usually 60 seconds).

TCP: Timers

- Persistent timer

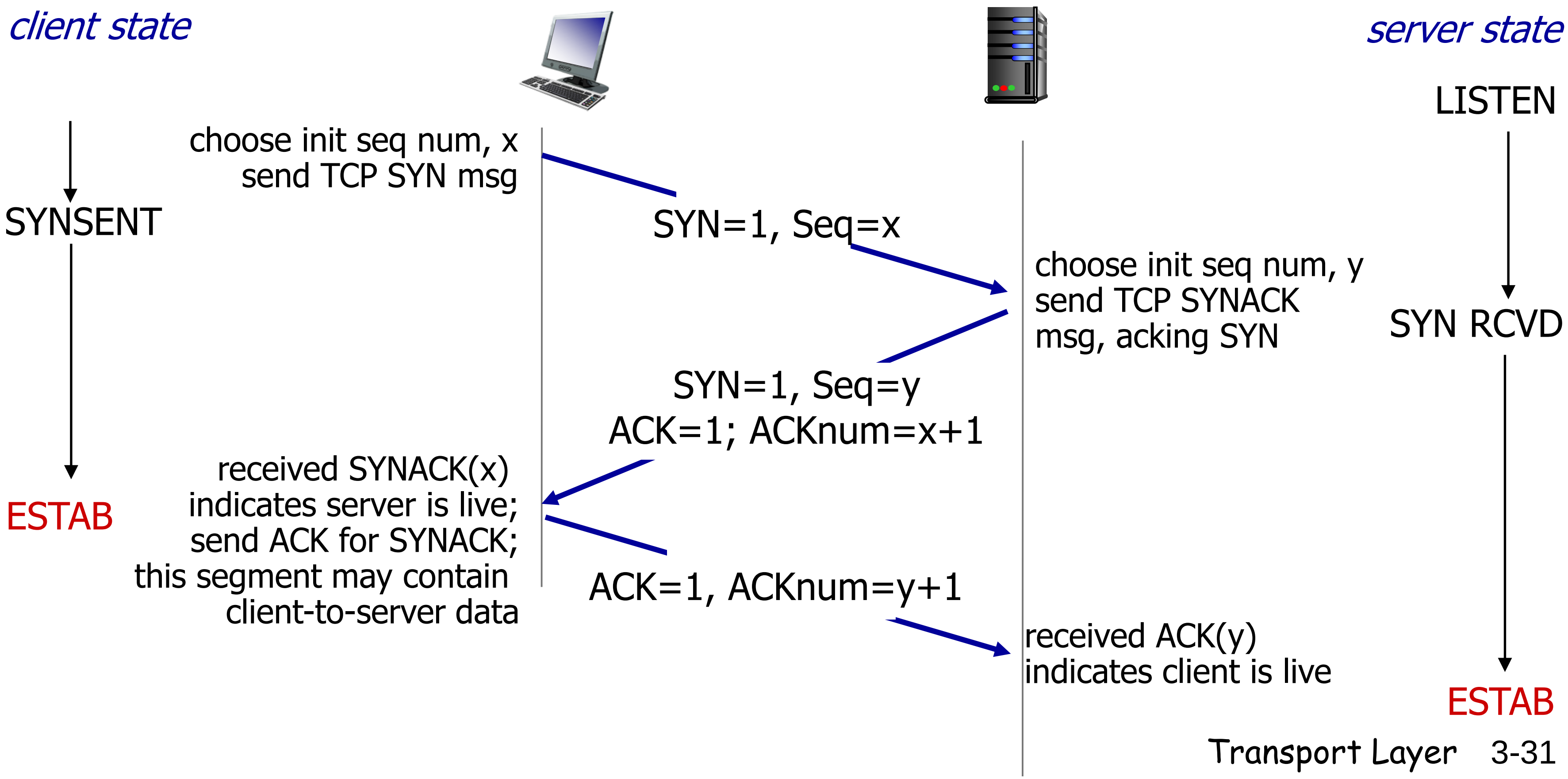


TCP: Timers

- Keep-alive timer
  - Used to prevent a long idle connection between two TCPs.
  - Each time the server hears from a client, it resets this timer.
  - When timeout, send a packet to peer
    - usually timeout value = 2 hours
    - when timeout occurs, probe packets are sent every 75 sec.
    - If there is no response to 10 probe packets, the connection is torn down.
- 2MSL(maximum segment lifetime) timer
  - Timer for the *TIMED WAIT* state while closing
    - (typically 120 sec.)
  - To make sure that all the packets created by this connection have died off.

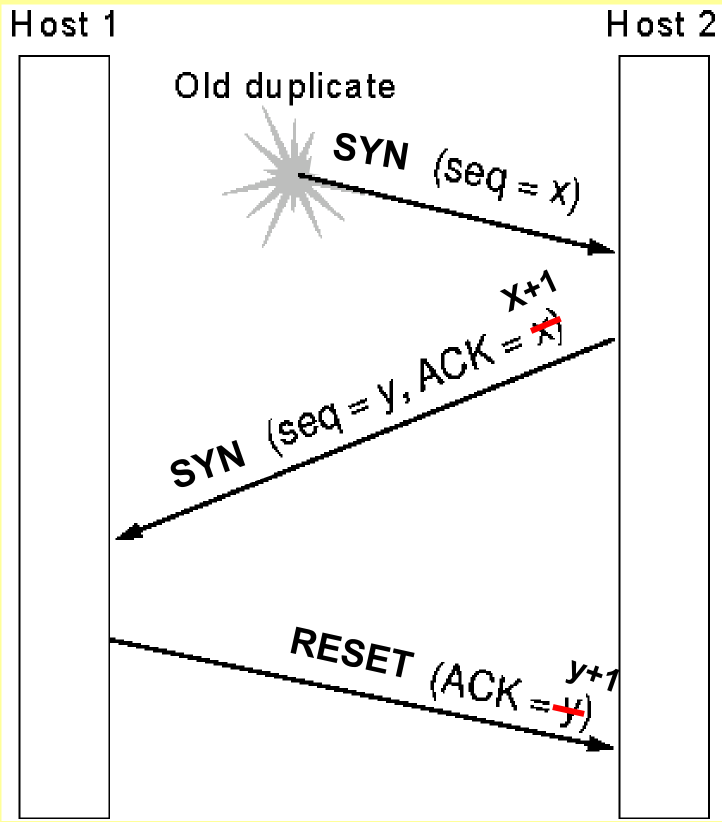
TCP Connection Management

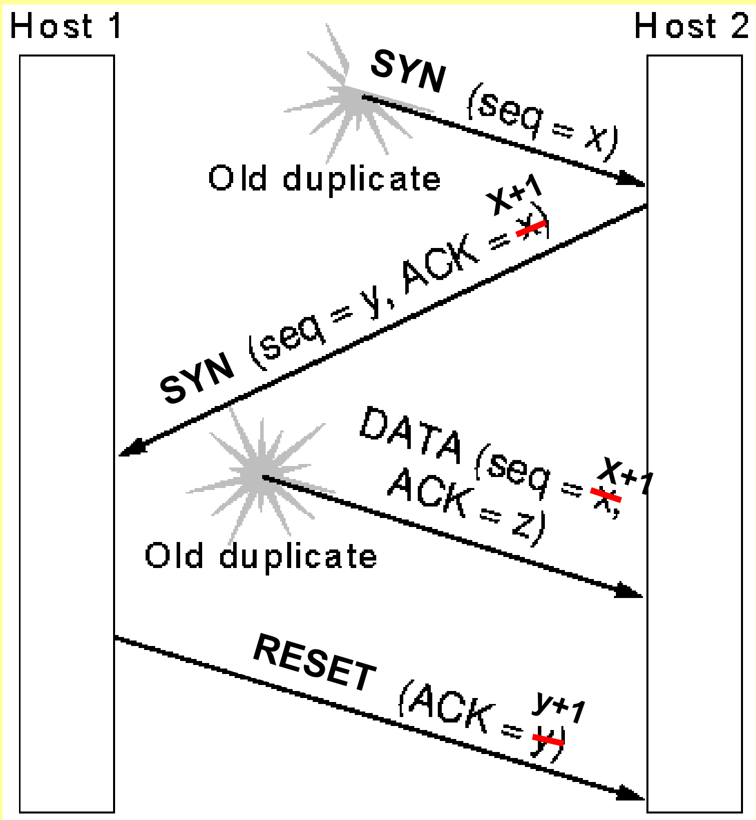
- Three-way handshake protocol
  - A connection-oriented protocol establishes a virtual path between the source and destination before sending data.



TCP Connection Management

- Three-way handshake : against abnormal cases



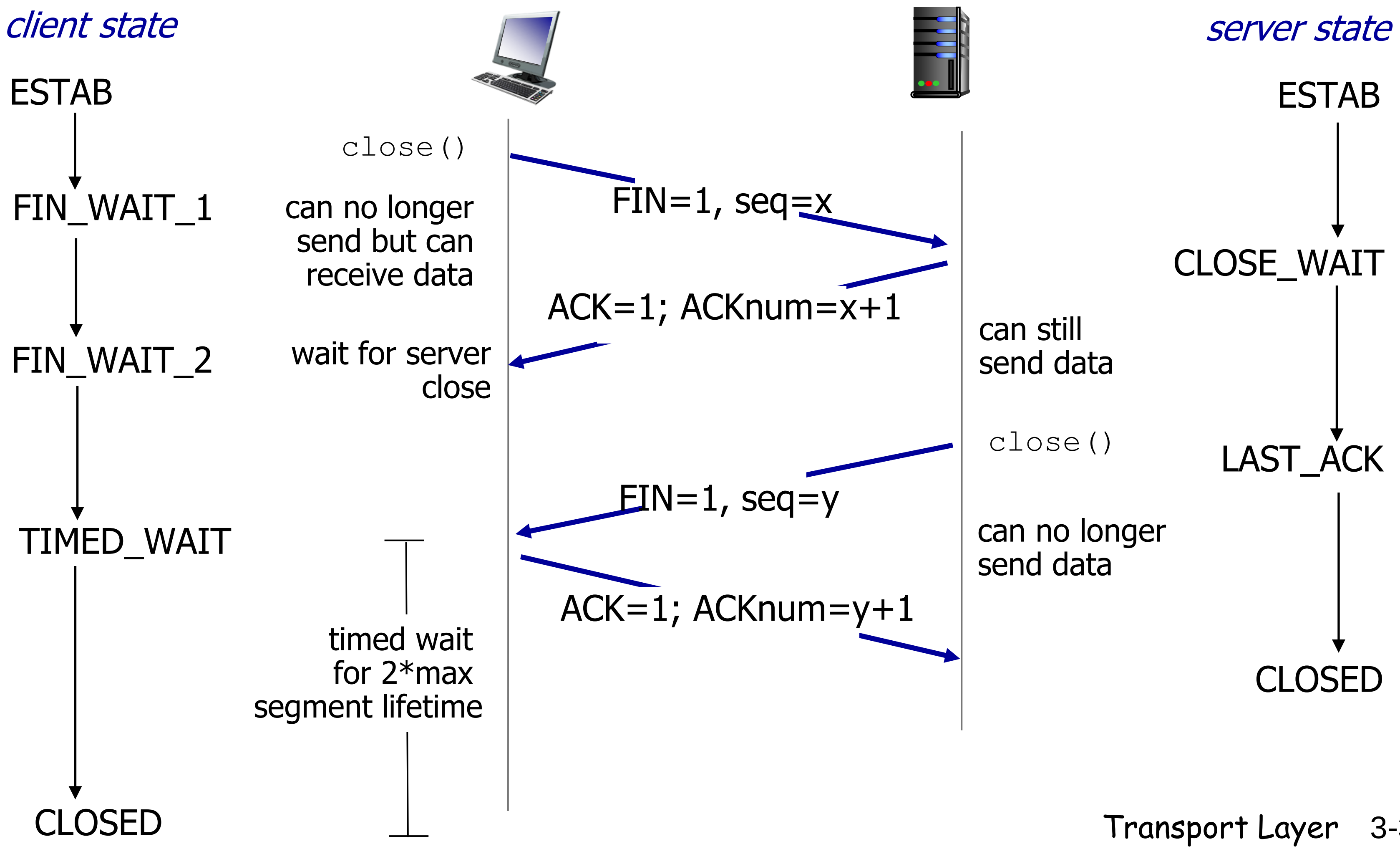


TCP: Closing a connection

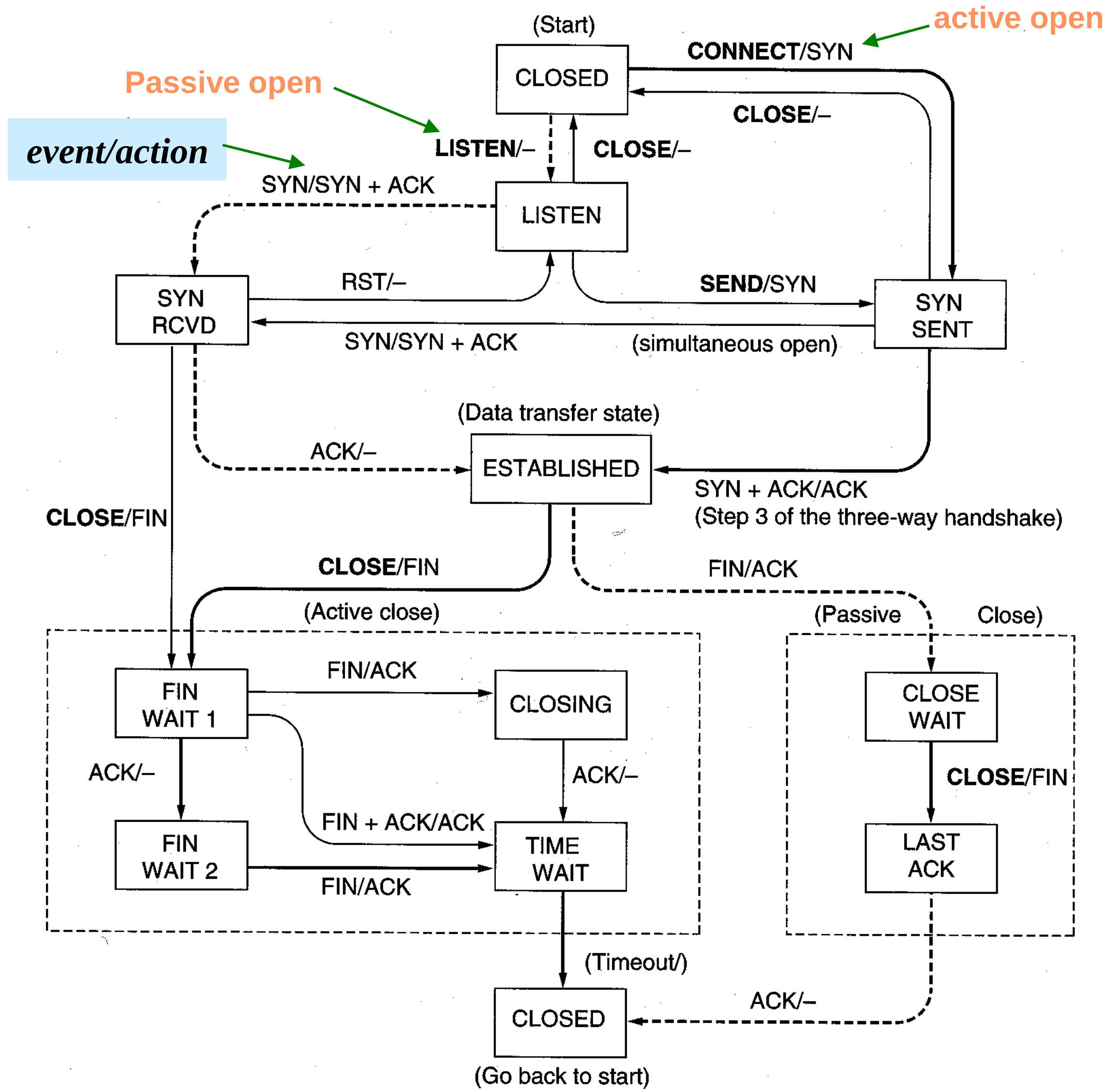
- client, server each close their side of connection
  - send TCP segment with FIN bit = 1
- respond to received FIN with ACK
  - on receiving FIN, ACK can be combined with own FIN
- simultaneous FIN exchanges can be handled

TCP: Closing a connection

- performed separately in each direction.



TCP: State Transition Diagram



dark line: client
dashed line: server

FIN WAIT 1: The client has said it is finished.
FIN WAIT 2: The server has agreed to release (half close)

TIME WAIT : Wait for all packets to die off.

TCP Connection Management

- 2MSL (Maximum Segment Lifetime) wait:
  - wait for final segment to be transmitted before releasing connection (typically 120 sec)
  - Socket *pair* cannot be reused during 2MSL
  - Delayed segments dropped
  - 2MSL effect
    - If you kill client and restart, it will *get* a different port
    - 2MSL wait protects against delayed segments from the previous "incarnation" of the connection.