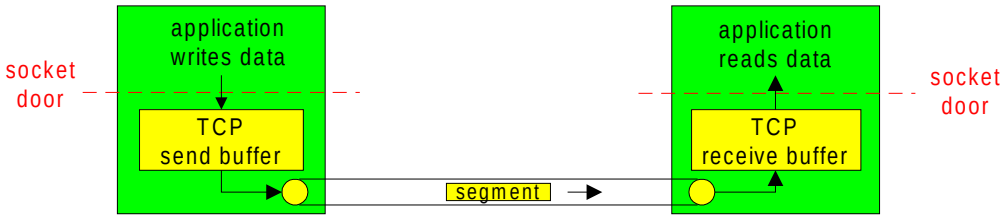
TCP

TCP: Overview

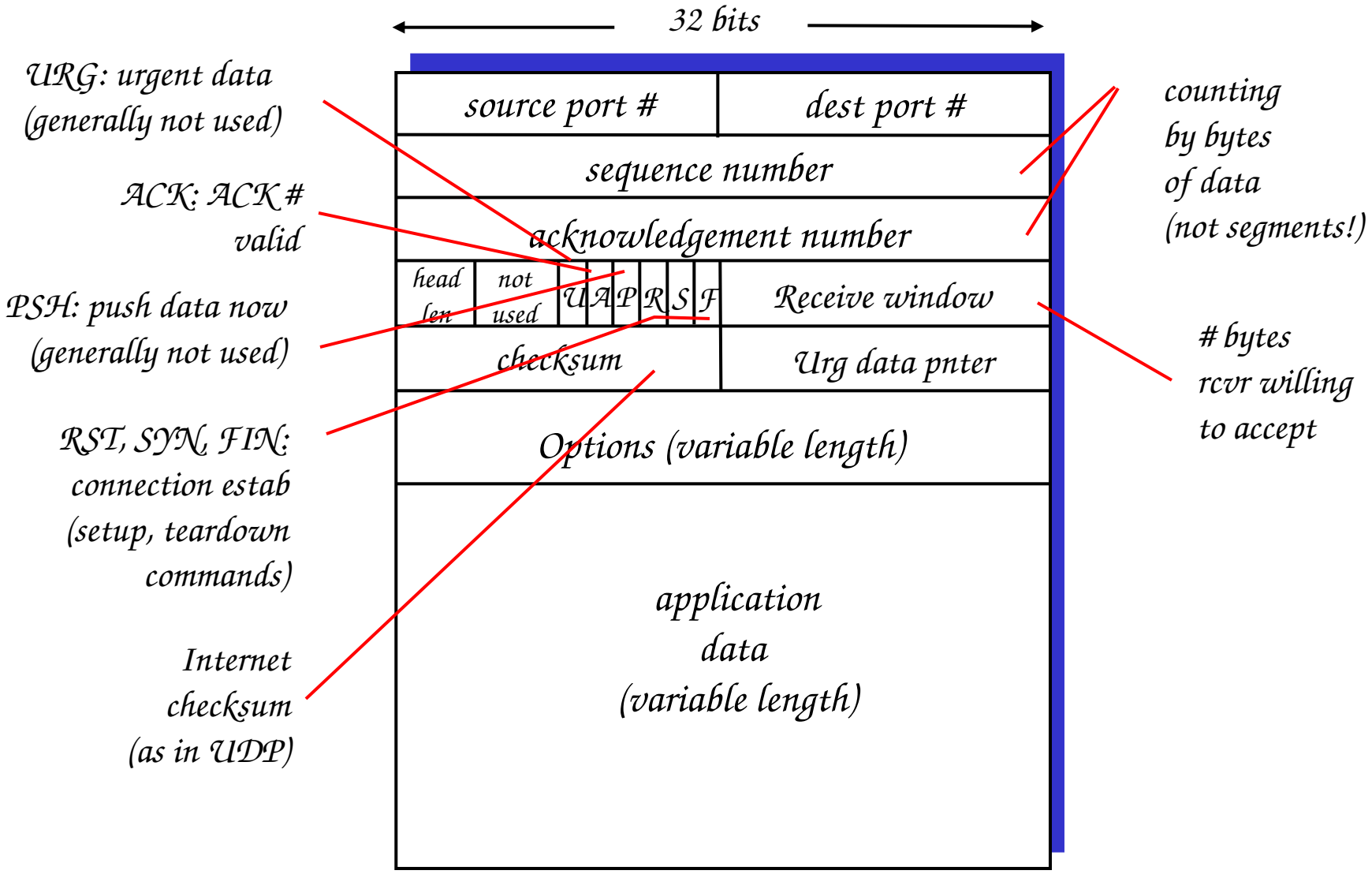
RFCs: 793, 1122, 1323, 2018, 2581

- *point-to-point:*
  - *one sender, one receiver*
- *reliable, in-order byte steam:*
  - *no "message boundaries"*
- *pipelined:*
  - *TCP congestion and flow control set window size*
- *send & receive buffers*



- *full duplex data:*
  - *bi-directional data flow in same connection*
  - *MSS: maximum segment size*
- *connection-oriented:*
  - *handshaking (exchange of control msgs) init's sender, receiver state before data exchange*
- *flow controlled:*
  - *sender will not overwhelm receiver*

TCP segment structure



TCP seq. #'s and ACKs

Seq. #'s:

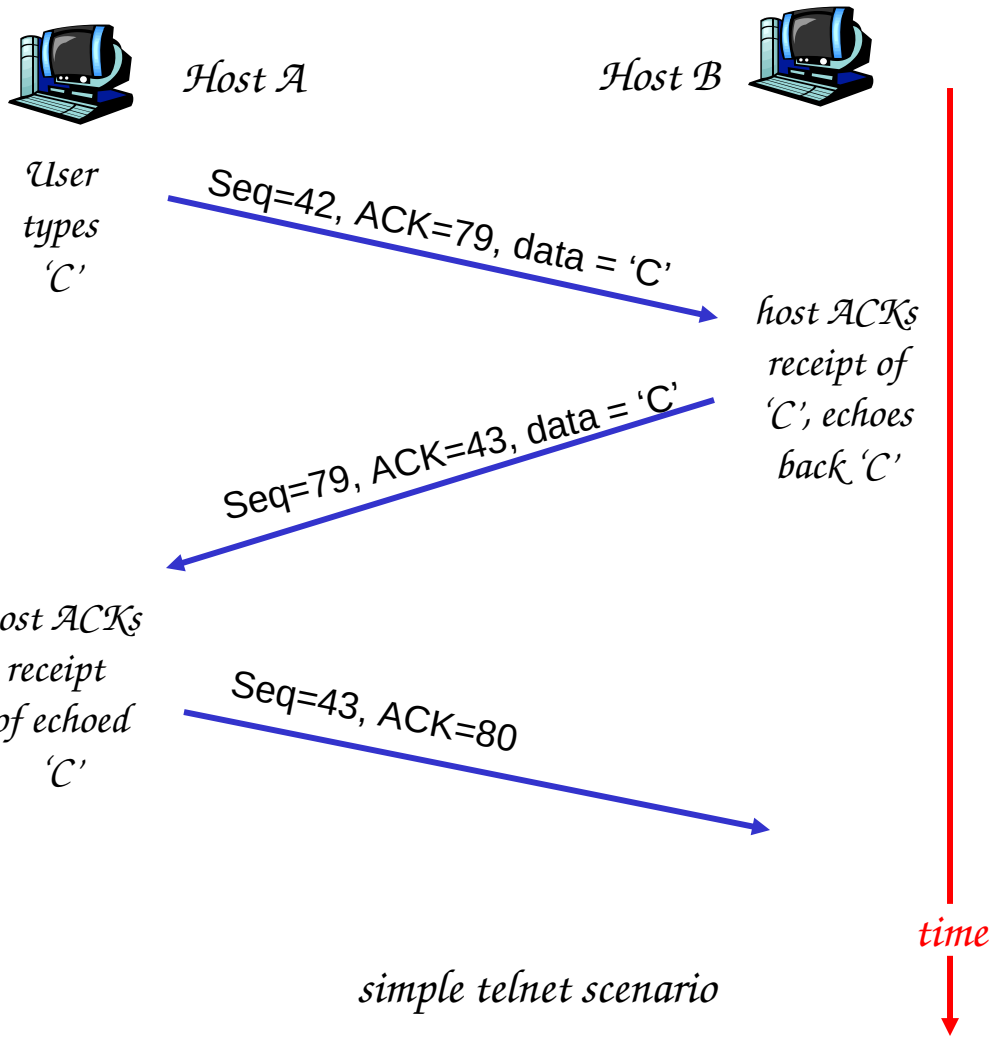
- *byte stream "number" of first byte in segment's data*

ACKs:

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

Q: how receiver handles out-of-order segments

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



simple telnet scenario

TCP Connection Management

Recall: TCP sender, receiver establish "connection" before exchanging data segments

- initialize TCP variables:
  - seq. #s
  - buffers, flow control info (e.g. **`RcvWindow`**)
- client: connection initiator
- server: contacted by client

TCP Connection Management

Three way handshake:

Step 1: client host sends TCP SYN segment to server

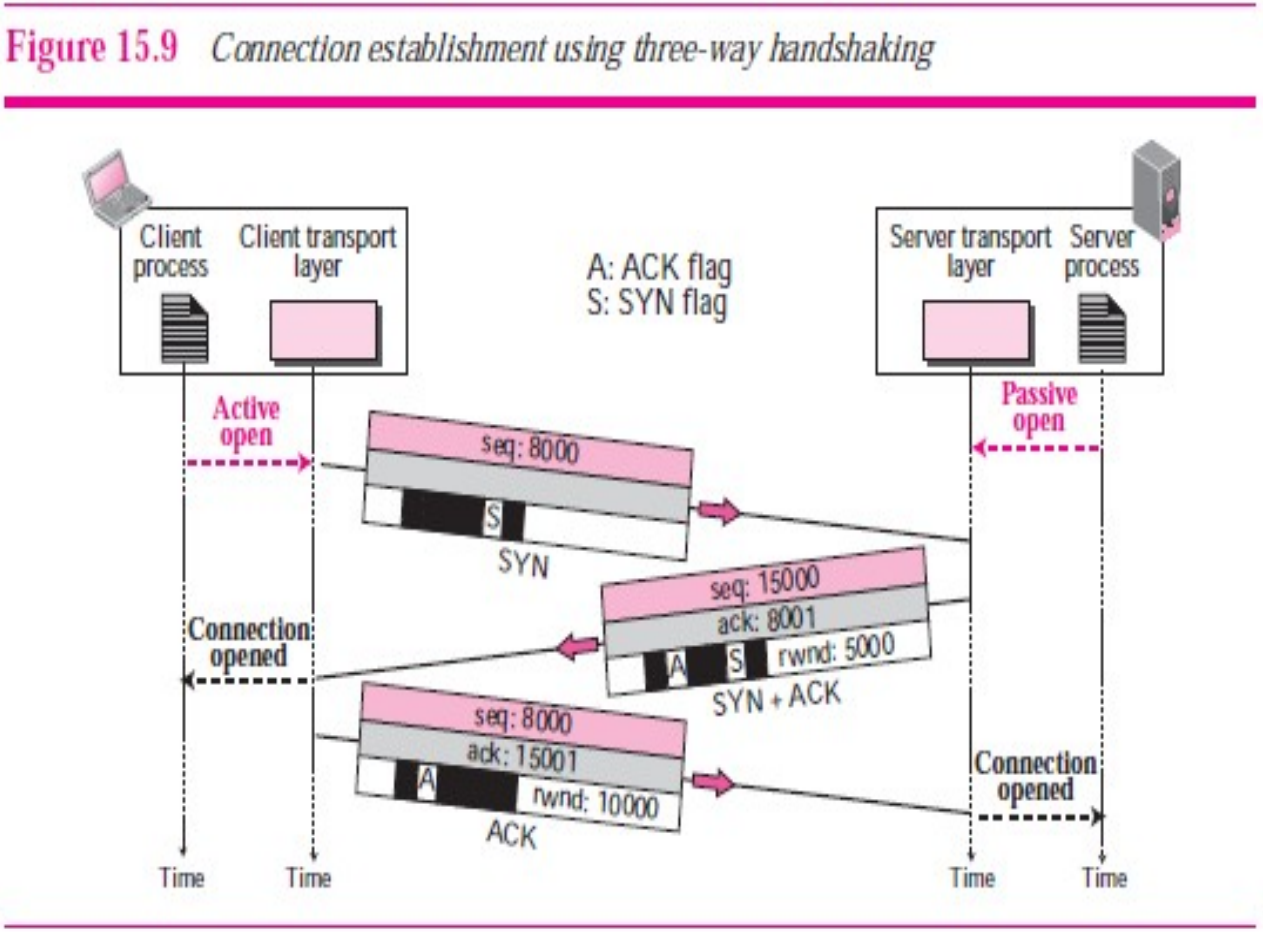
- specifies initial seq #
- no data

Step 2: server host receives SYN, replies with SYNACK segment

- server allocates buffers
- specifies server initial seq. #

Step 3: client receives SYNACK, replies with ACK segment, which may contain data

Figure 3.47: Connection establishment using three-way handshaking



Note

A SYN segment cannot carry data, but it consumes one sequence number.

A SYN + ACK segment cannot carry data, but does consume one sequence number.

An ACK segment, if carrying no data, consumes no sequence number.

Syn Flood Attack

- One or more malicious attackers send large no of SYN Segments.
- TCP allocates resources to each connection. (Transfer control block, timers…)
- If no of SYN segments large server eventually runs out of resources & shuts down.
- Limit the no of connection, filter unwanted address, postpone the allocation of resources.

Figure 3.48: Data transfer

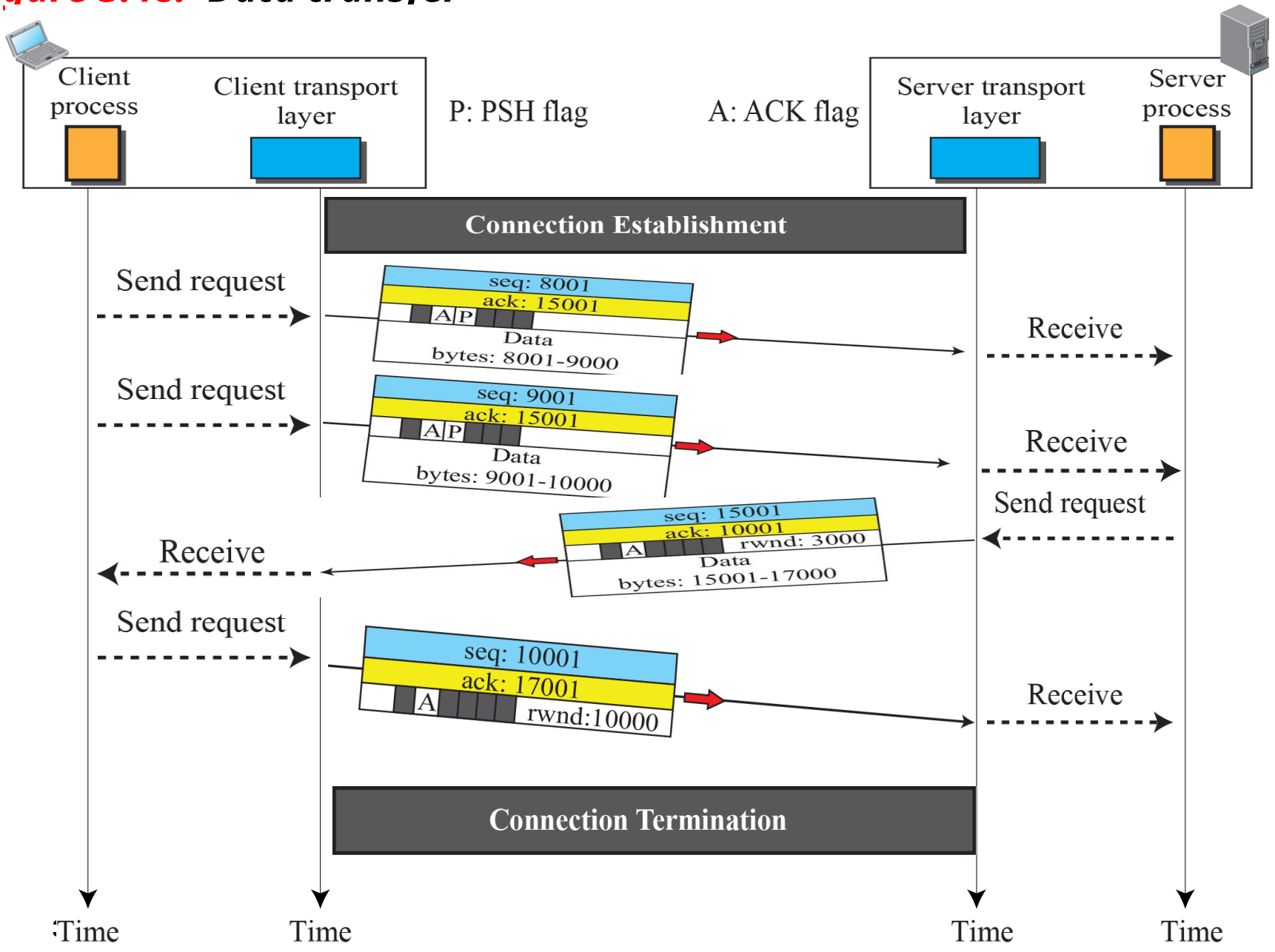
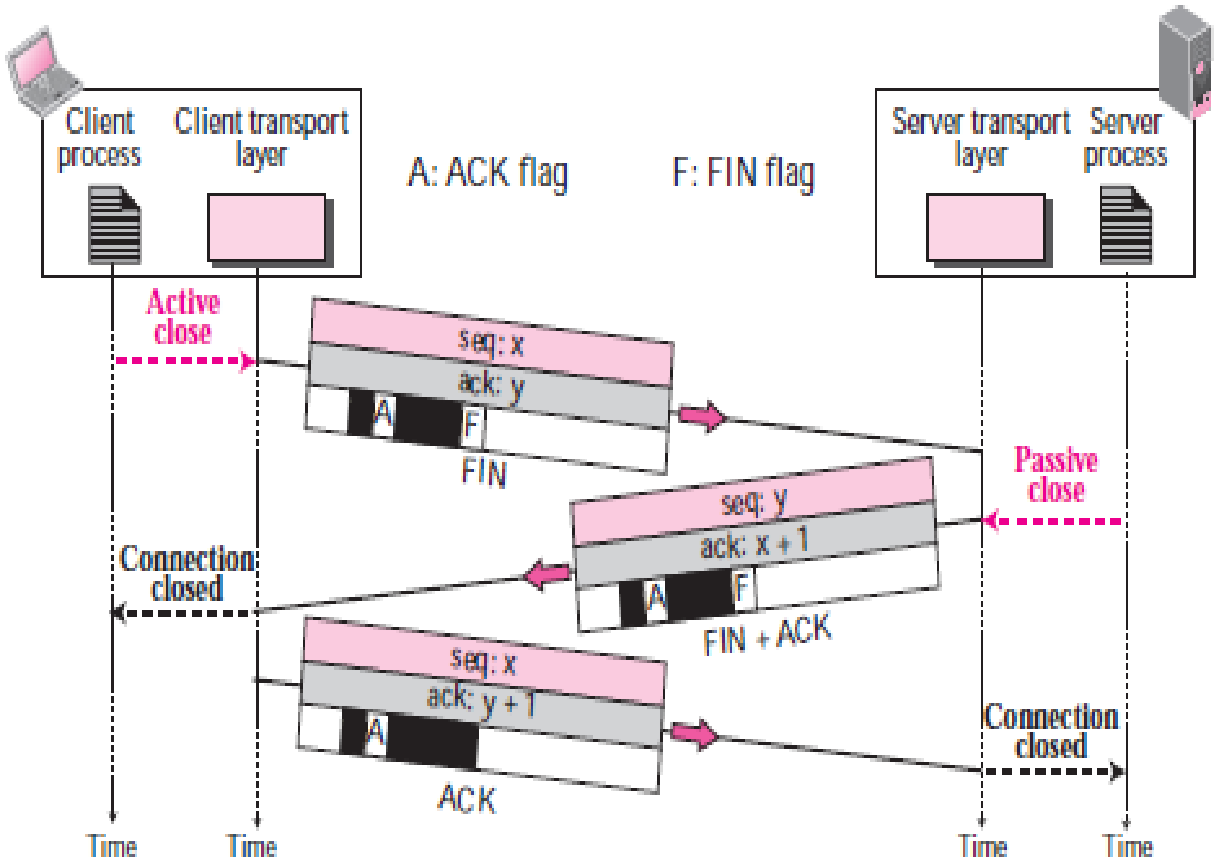


Figure 3.49: Connection termination using three-way handshaking

Figure 15.11 *Connection termination using three-way handshaking*

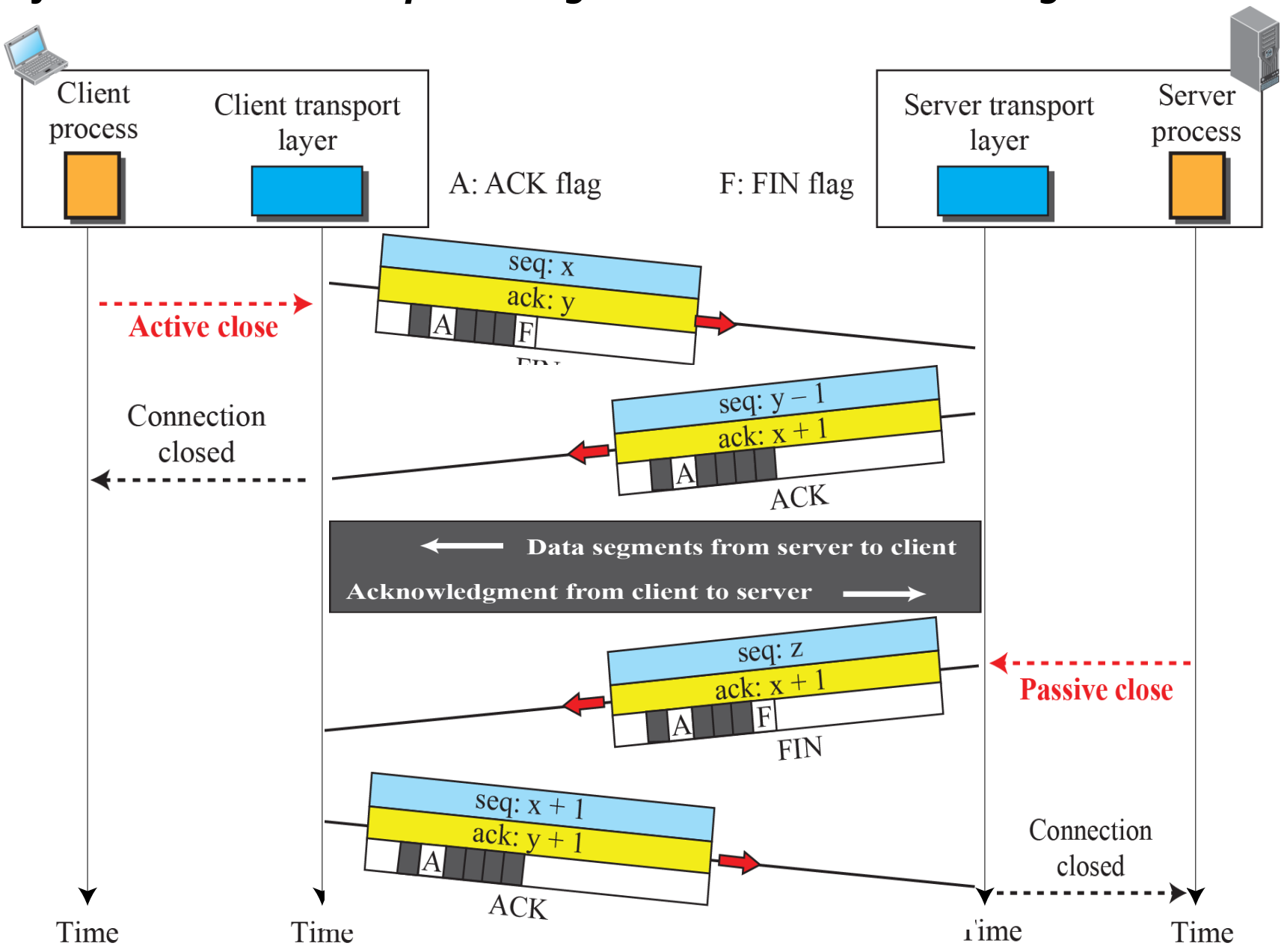


Note

The FIN segment consumes one sequence number if it does not carry data.

The FIN + ACK segment consumes one sequence number if it does not carry data.

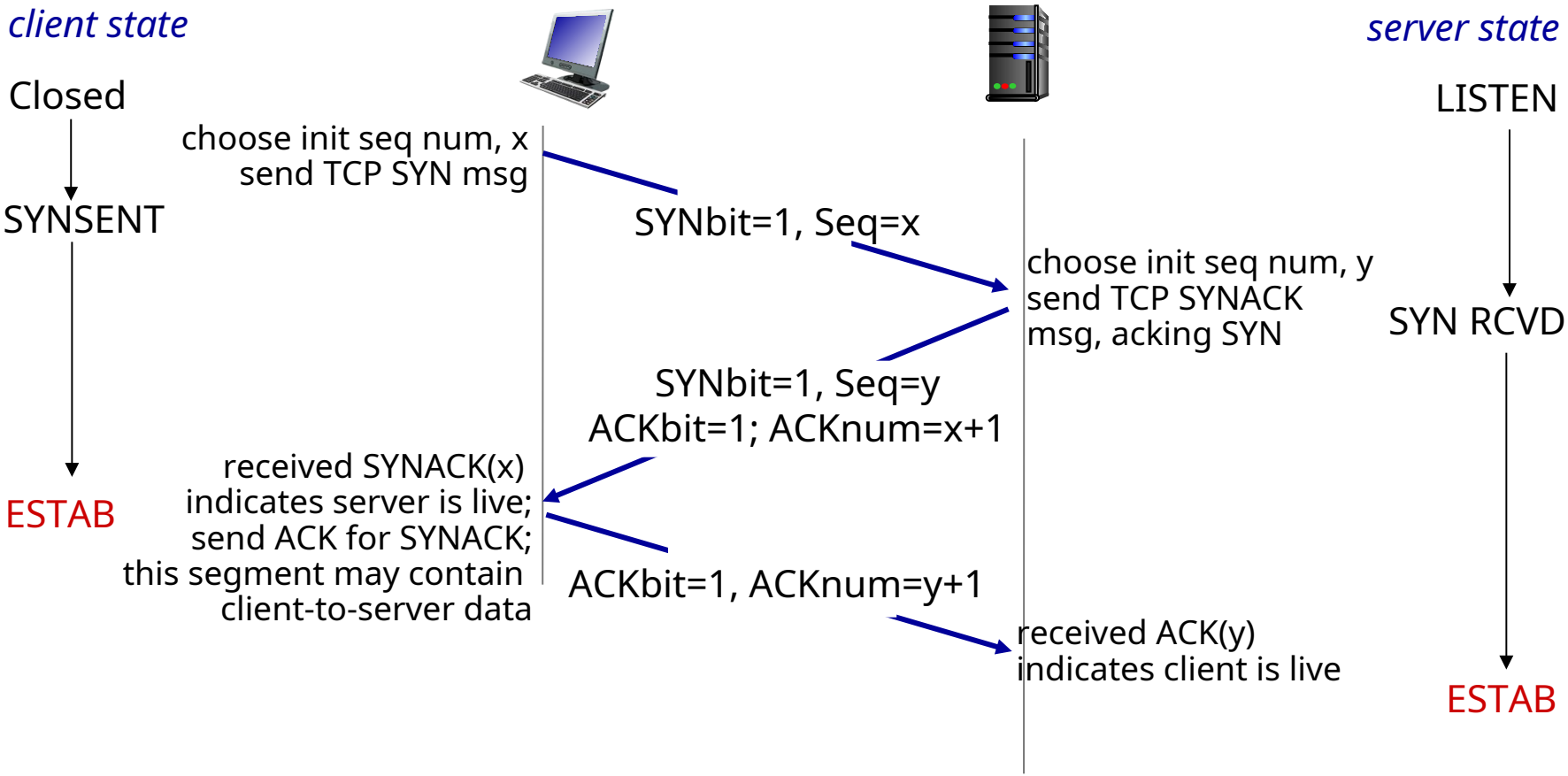
Half-close: One end stop sending data while still receiving data



State Transmission Diagram

To keep track of all the different events happening during connection establishment, connection termination, and data transfer, TCP is specified as the finite state machine (FSM).

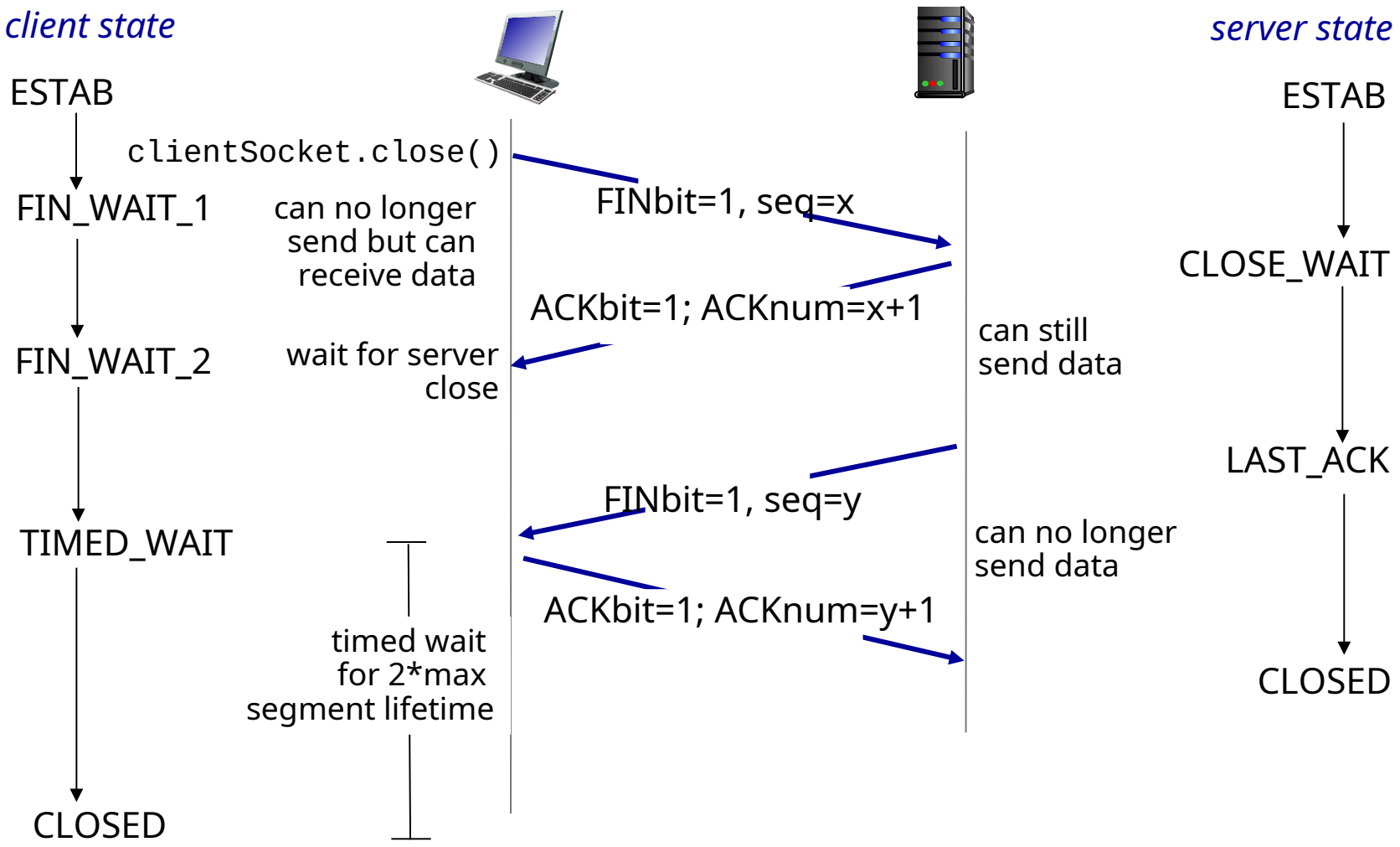
TCP 3-way handshake



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



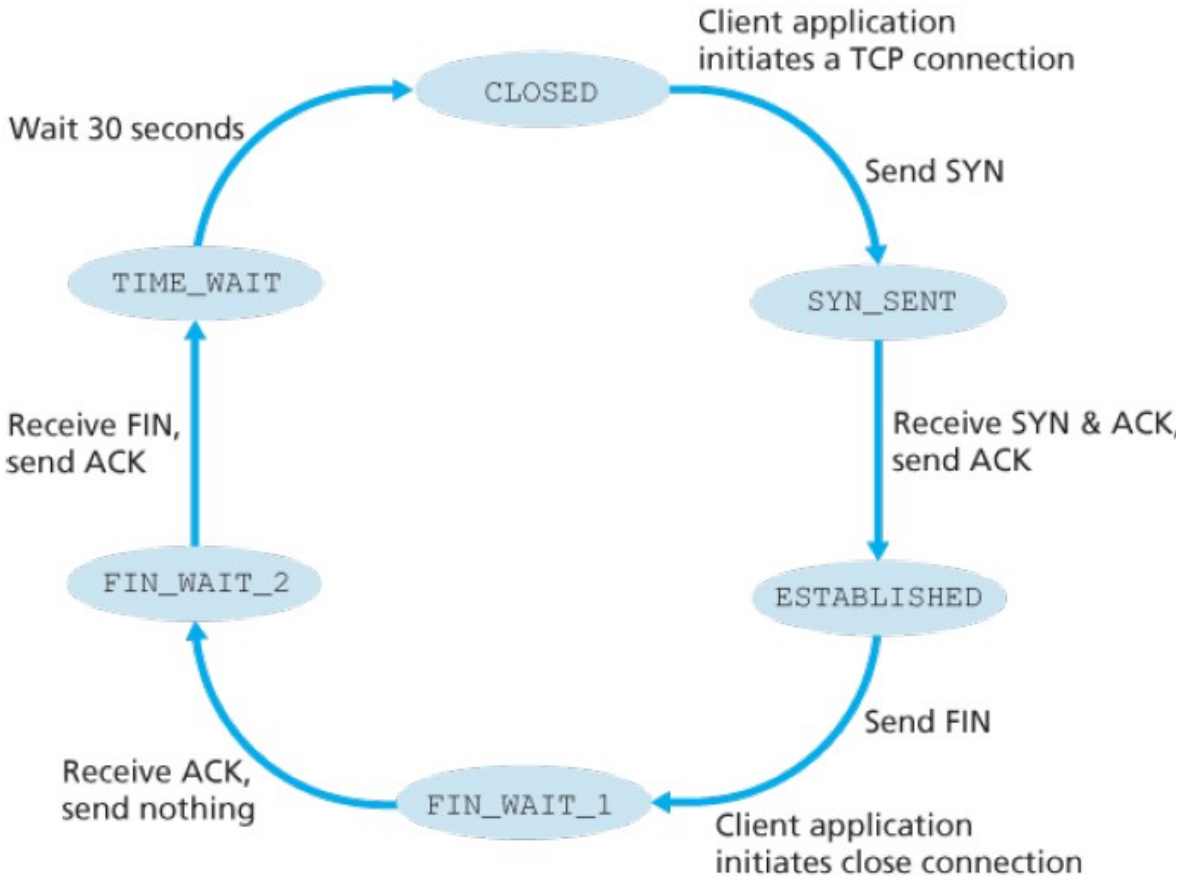


Figure 3.41 A typical sequence of TCP states visited by a client TCP

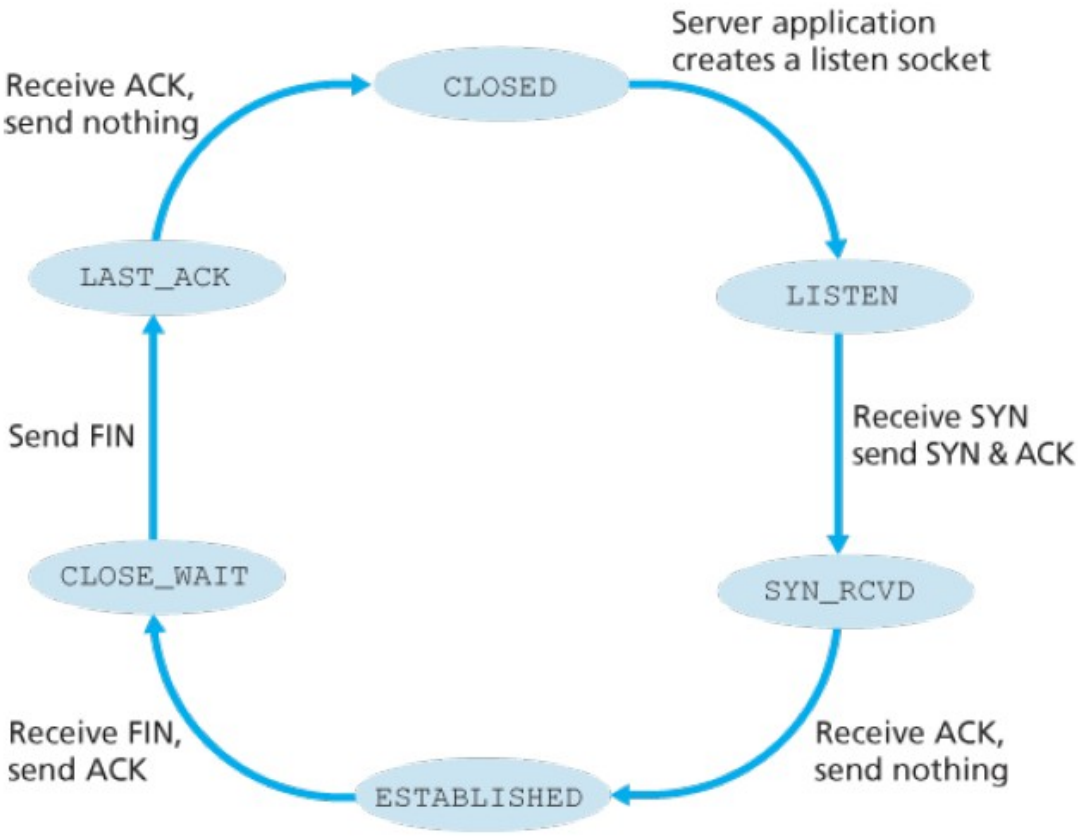


Figure 3.42 A typical sequence of TCP states visited by a server-side TCP

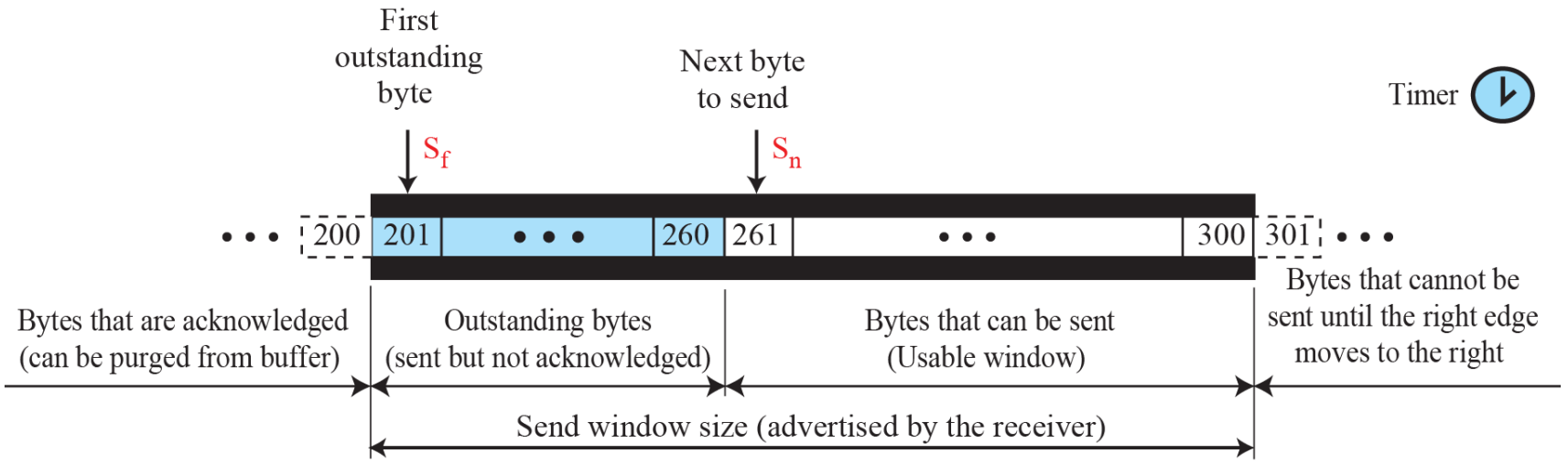
Windows in TCP

TCP uses two windows (send window and receive window) for each direction of data transfer, which means four windows for a bidirectional communication. To make the discussion simple, we make an unrealistic unidirectional; the bidirectional communication can be inferred using two unidirectional communications with piggybacking.

- Send Window

- Receive Window

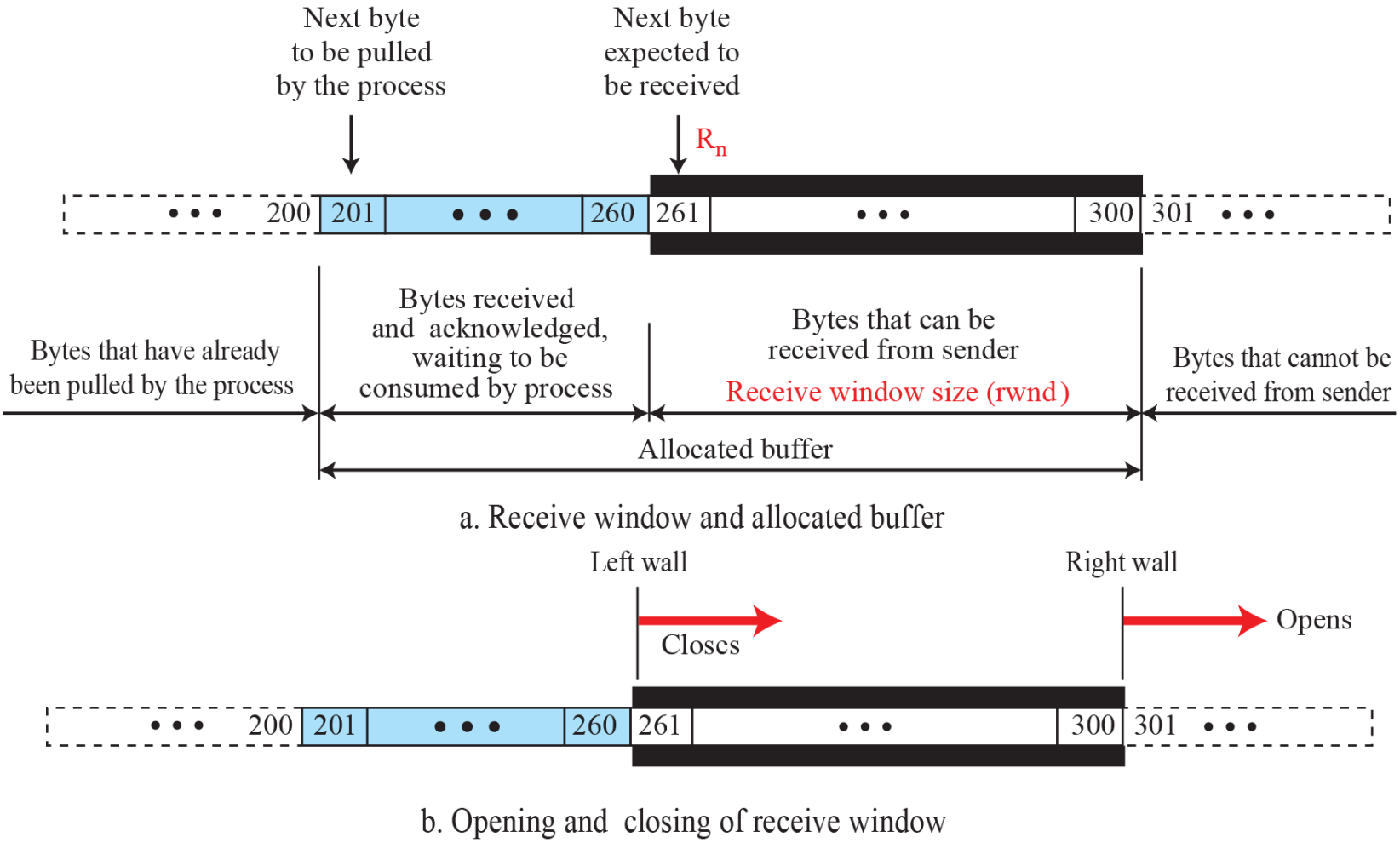
Send window in TCP



a. Send window

b. Opening, closing, and shrinking send window

Receive window in TCP

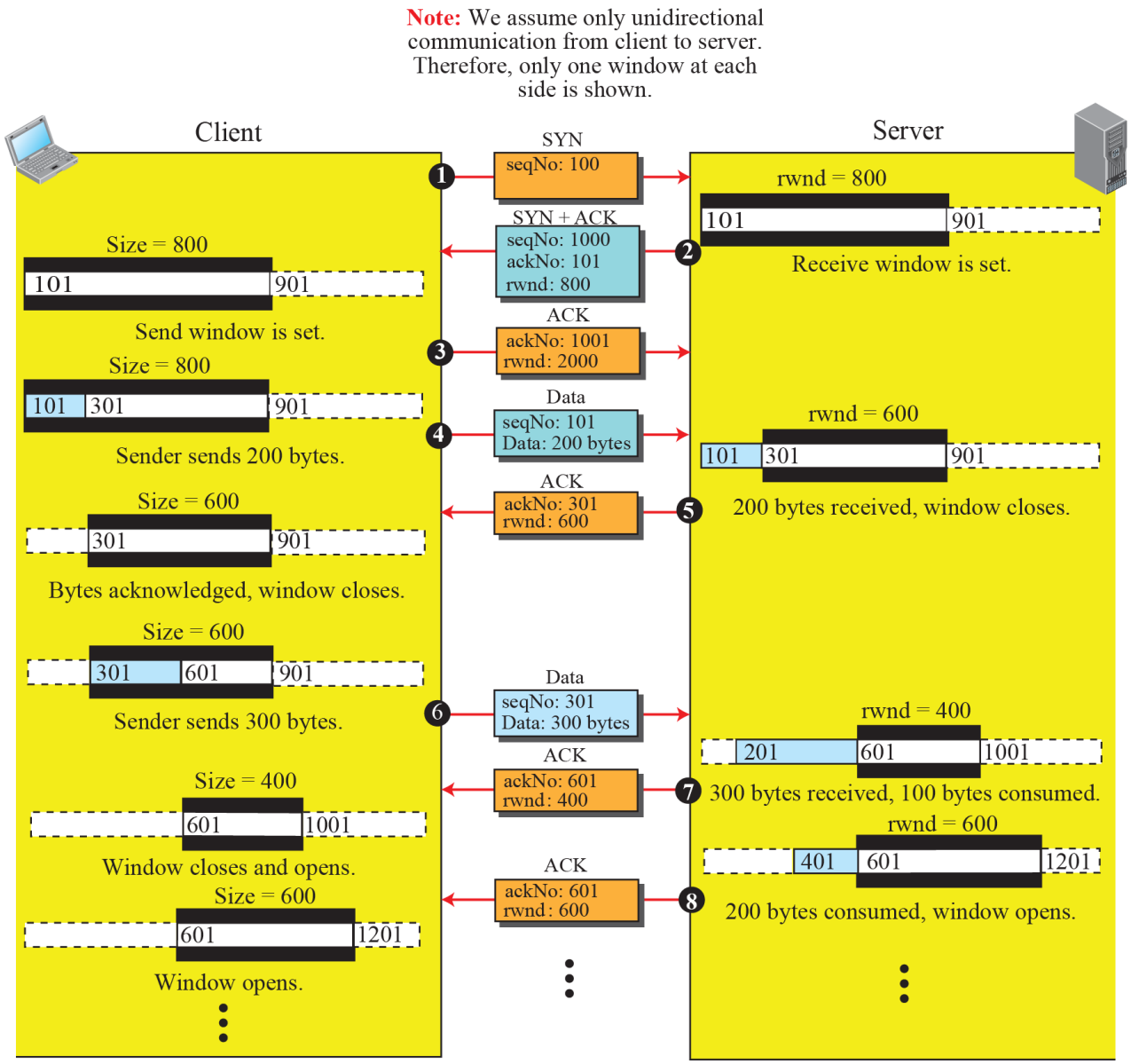


The receive window opens and closes; in practice,the window should never shrink. **rwnd = buffer size − number of waiting bytes to be pulled**

Opening and Closing Windows

- To achieve flow control, TCP forces the sender and the receiver to adjust their window sizes, although the size of the buffer for both parties is fixed when the connection is established.
- The receive window closes (moves its left wall to the right) when more bytes arrive from the sender; it opens (moves its right wall to the right) when more bytes are pulled by the process. Rx window can not shrink.
- The opening, closing, and shrinking of the send window is controlled by the receiver.
- The send window closes (moves its left wall to the right) when a new acknowledgement allows it to do so. The send window opens (its right wall moves to the right) when the receive window size (rwnd) advertised by the receiver allows it to do so (new ackNo+new rwnd > last ackNo+ last rwnd). The send window shrinks in the event this situation does not occur.

An example of flow control



- *Shrinking send window is discouraged. But one exception: rxer can temporarily shut down the send window by sending rwnd 0. In this case sender does not actually shrink, but stops sending data until new rwnd. Also sender can send a 1 byte probe segment to prevent deadlock.*
- ***Silly Window Syndrome***
- *A serious problem can arise in the sliding window operation when either the* ***sending application program creates data slowly*** *or the* ***receiving application program consumes data slowly, or both****.*
- ***Syndrome Created by the Sender***
- ***Nagle's Algorithm*** *is simple:*

1. *The sending TCP sends the first piece of data it receives from the sending application program even if it is only 1 byte.*

2. *After sending the first segment, the sending TCP accumulates data in the output buffer and waits until either the receiving TCP sends an acknowledgment or until enough data has accumulated to fill a maximum-size segment. At this time, the sending TCP can send the segment.*

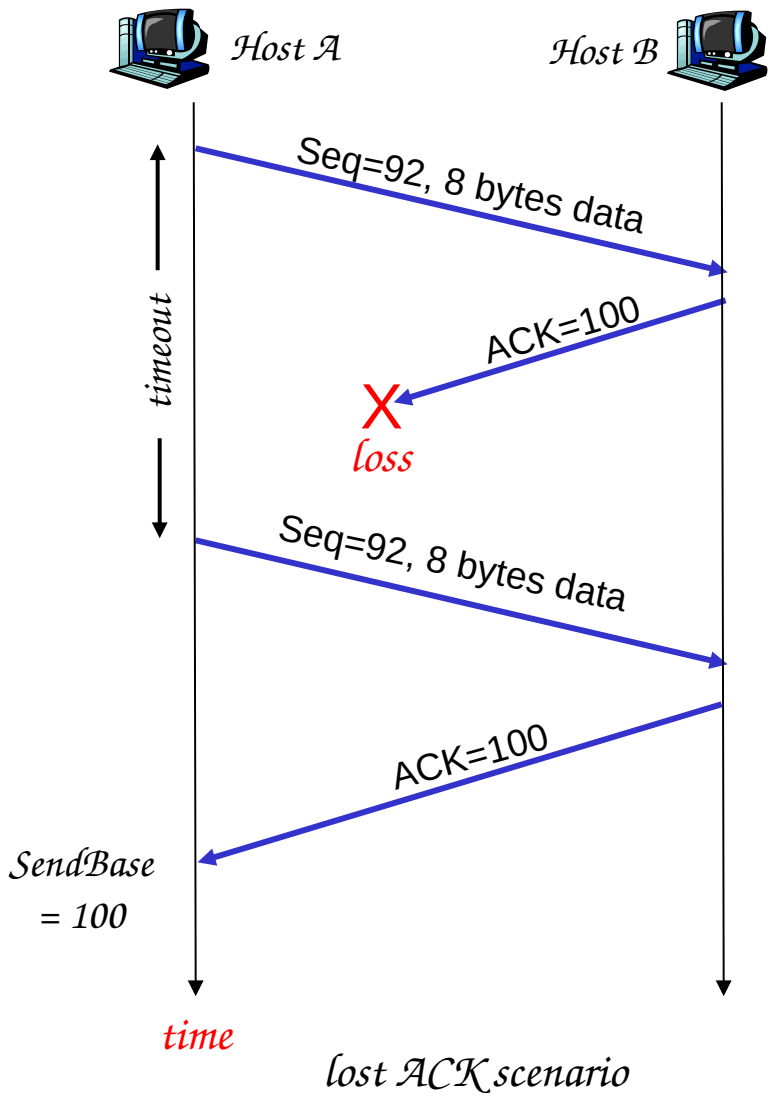
3. *Step 2 is repeated for the rest of the transmission. Segment 3 is sent immediately if an acknowledgment is received for segment 2, or if enough data have accumulated to fill a maximum-size segment.*

This takes into account the application speed & network speed.

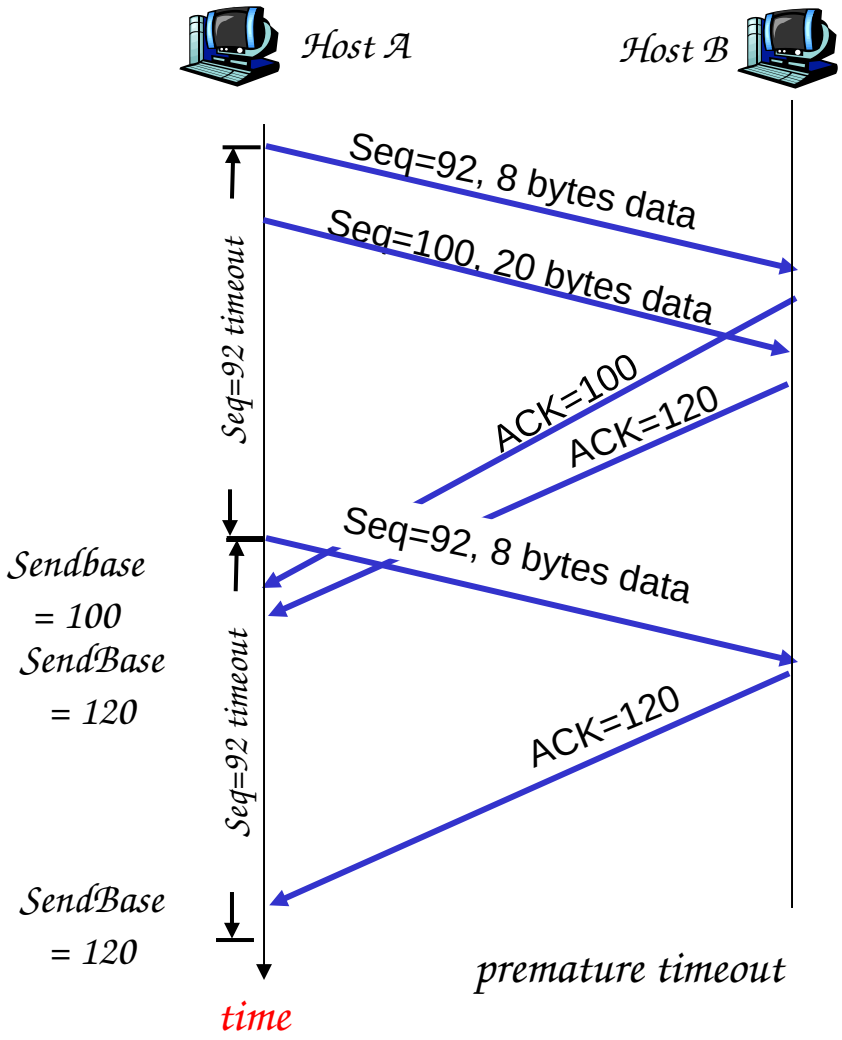
Syndrome Created by the Receiver

- The receiving TCP may create a silly window syndrome if it is serving an application program that consumes data slowly, for example, 1 byte at a time.
- One byte of data is consumed and a segment carrying 1 byte of data is sent results in silly window syndrome.
- **Clark's Solution :Clark's solution** is to send an acknowledgment as soon as the data arrive, but to announce a window size of zero until either there is enough space to accommodate a segment of maximum size or until at least half of the receive buffer is empty.
- **Delayed Acknowledgment** The second solution is to delay sending the acknowledgment. This means that when a segment arrives, it is not acknowledged immediately. The receiver waits until there is a decent amount of space in its incoming buffer before acknowledging the arrived segments. The delayed acknowledgment prevents the sending TCP from sliding its window.
- Delayed acknowledgment also has another advantage: it reduces traffic. The receiver does not have to acknowledge each segment. However, there also is a disadvantage in that the delayed acknowledgment may result in the sender unnecessarily retransmission

TCP: retransmission scenarios

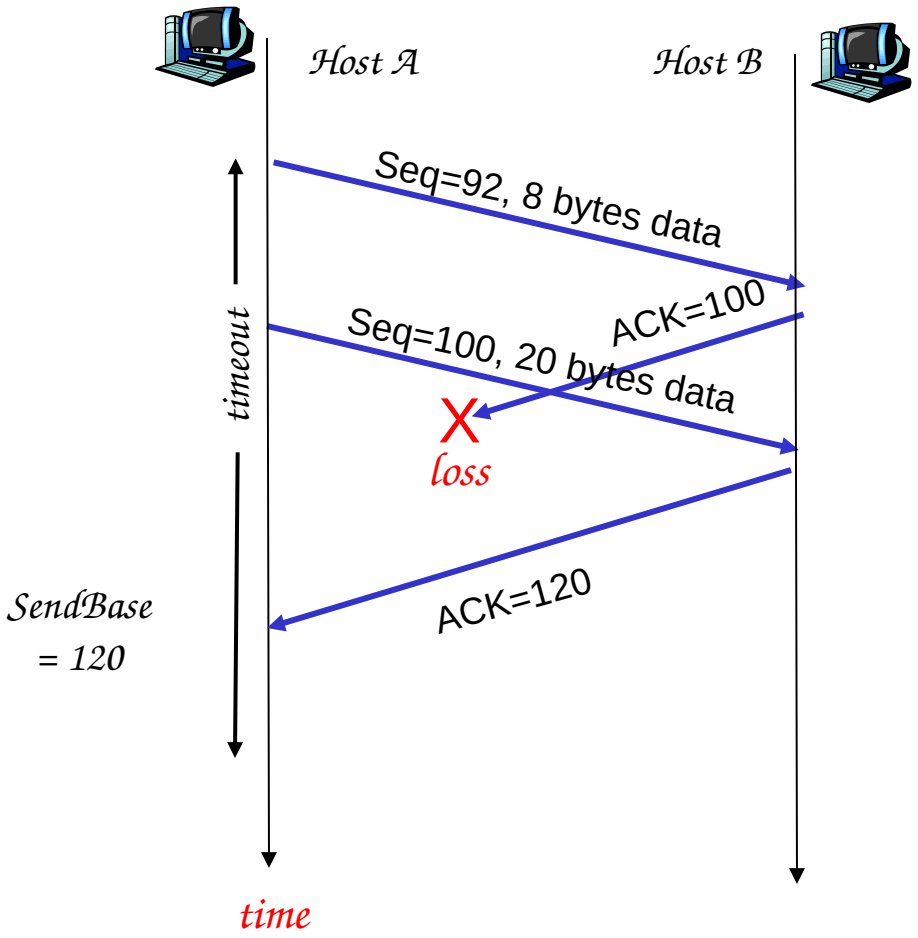


lost ACK scenario



premature timeout

TCP retransmission scenarios (more)



Cumulative ACK scenario

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 startsat lower end of gap

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 3 ACKs for the same data, it supposes that segment after ACKed data was lost:
  - fast retransmit: resend segment before timer expires

TCP Round Trip Time and Timeout

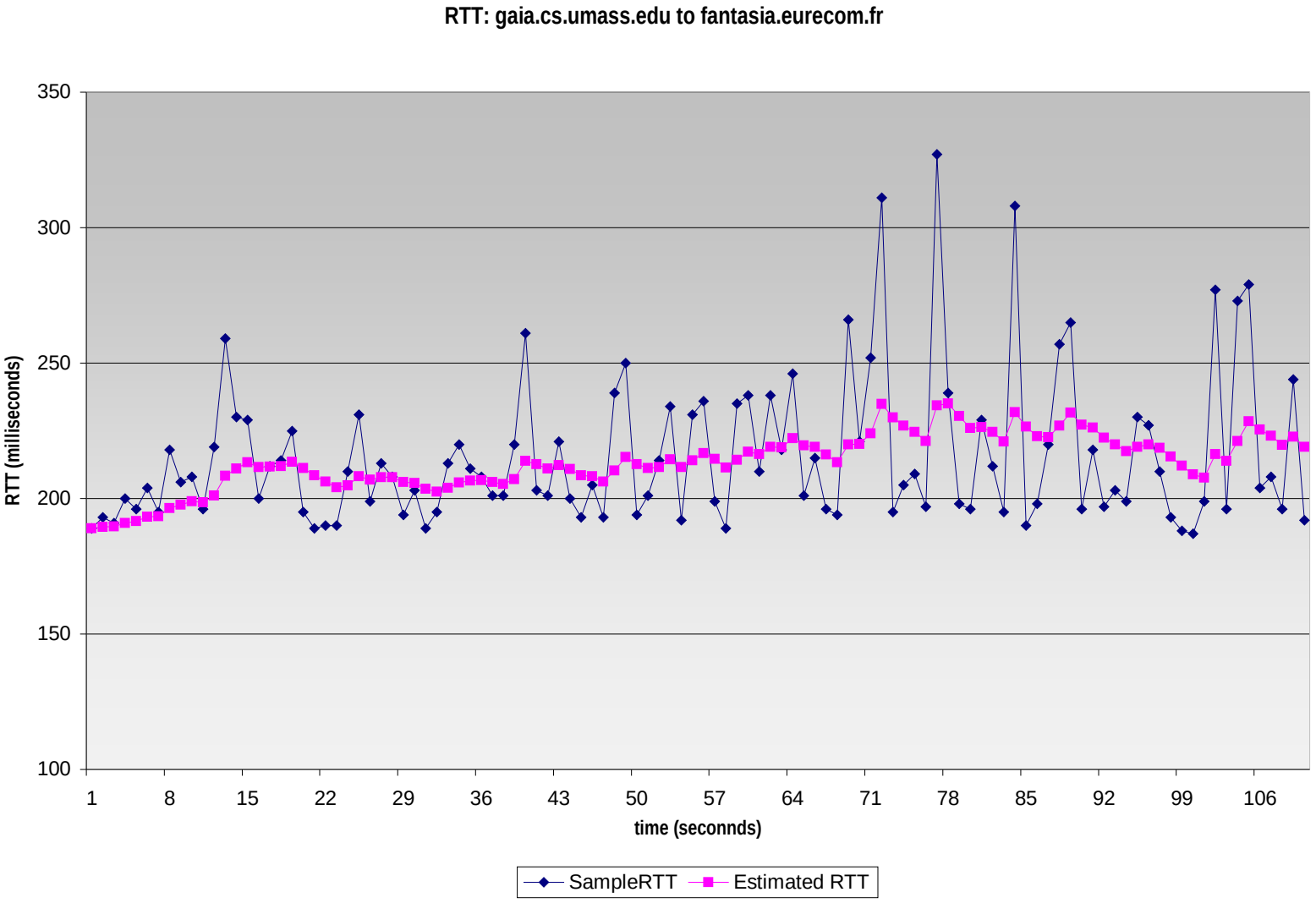
Q: how to set TCP timeout value?

- longer than RTT
  - but RTT varies
- too short: premature timeout
  - unnecessary retransmissions
- too long: slow reaction to segment loss

Q: how to estimate RTT?

- **SampleRTT**: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- **SampleRTT** will vary, want estimated RTT "smoother"
  - average several recent measurements, not just current **SampleRTT**

Example RTT estimation:



TCP sender events:

data rcvd from app:

- Create segment with seq #
- seq # is byte-stream number of first data byte in segment
- start timer if not already running (think of timer as for oldest unacked segment)
- expiration interval: `TimeOutInterval`

timeout:

- retransmit segment that caused timeout
- restart timer

Ack rcvd:

- If acknowledges previously unacked segments
  - update what is known to be acked
  - start timer if there are outstanding segments