COMP 3234B
Computer and Communication Networks

2nd semester 2023-2024

Transport Layer (III)

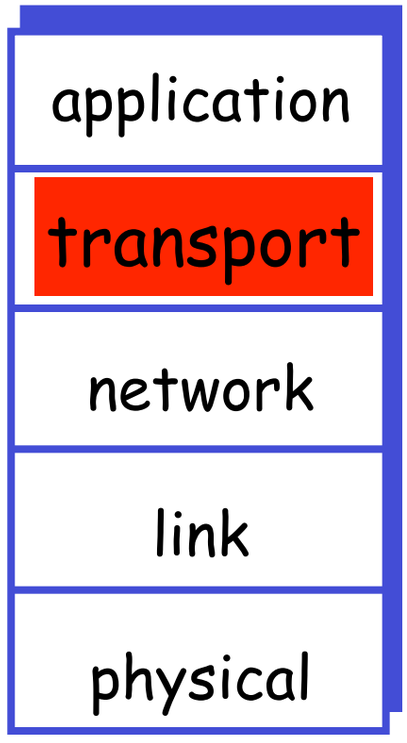
Prof. C Wu

Department of Computer Science
The University of Hong Kong

Roadmap

Transport layer

- Principles behind transport-layer services
  - multiplexing/demultiplexing (ILO1, 2)
  - reliable data transfer:(ILO 2, 3)
    - rdt 1.0
    - rdt 2.0, 2.1, 2.2
    - rdt 3.0
    - GBN
    - selective repeat
- Transport protocols in the Internet (ILO 2, 3)
  - TCP: RDT, flow control, congestion control
  - UDP



TCP: overview

RFCs: 793, 1122, 1323, 2018, 2581

- Point-to-point

  one sender, one receiver
- Reliable data transfer

  in-order byte steam
- Pipelined transfer

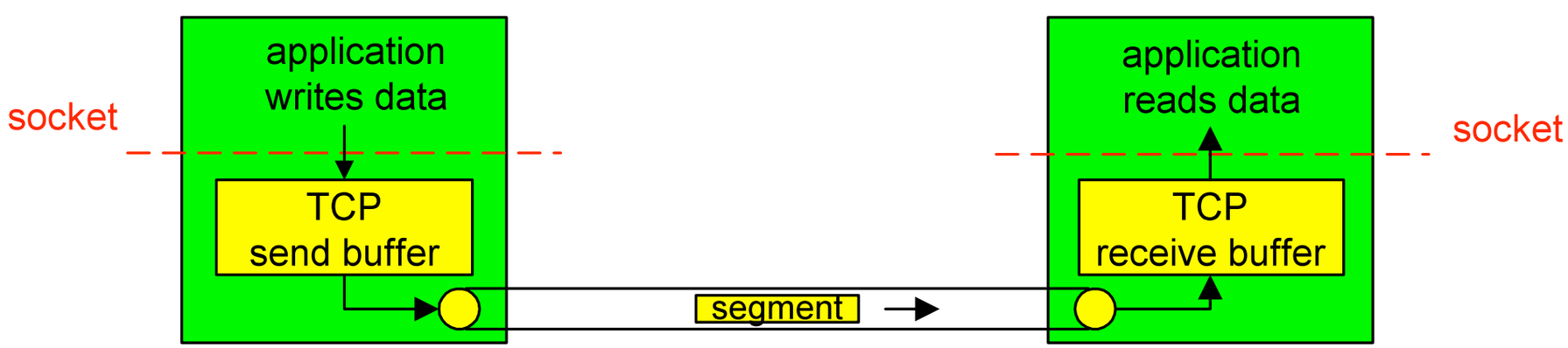
  TCP congestion control and flow control (set window size)
- Full duplex service

  bi-directional data flow in same connection
- Connection-oriented

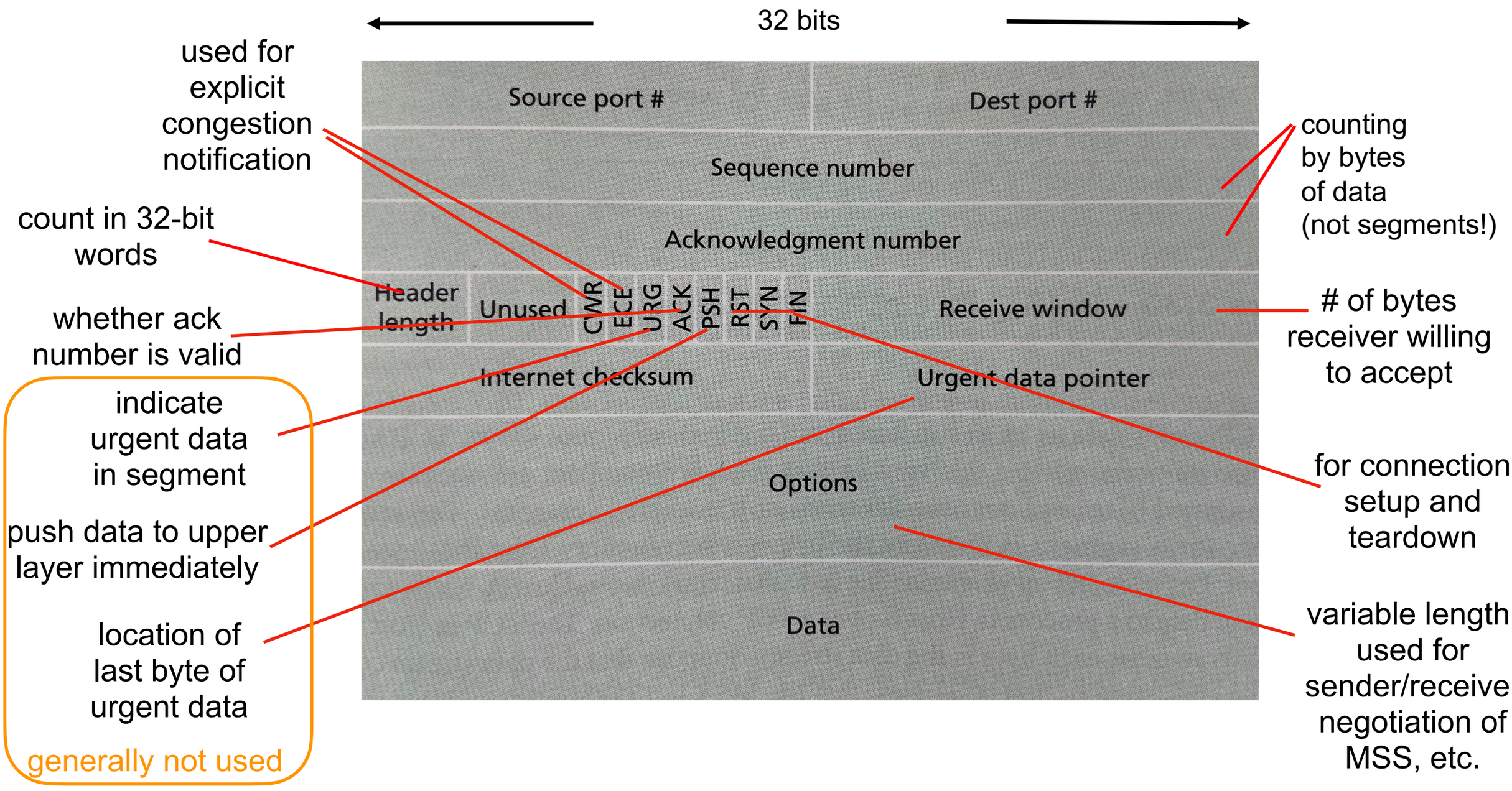
  handshaking (exchange of control msgs) before data exchange
- Flow and congestion controlled

  sender will not overwhelm receiver

  sender adjusts sending rate upon detecting congestion



TCP: segment structure



MSS (maximum segment size)

The maximum amount of application-layer data in a segment

TCP: seq. #'s and ACKs

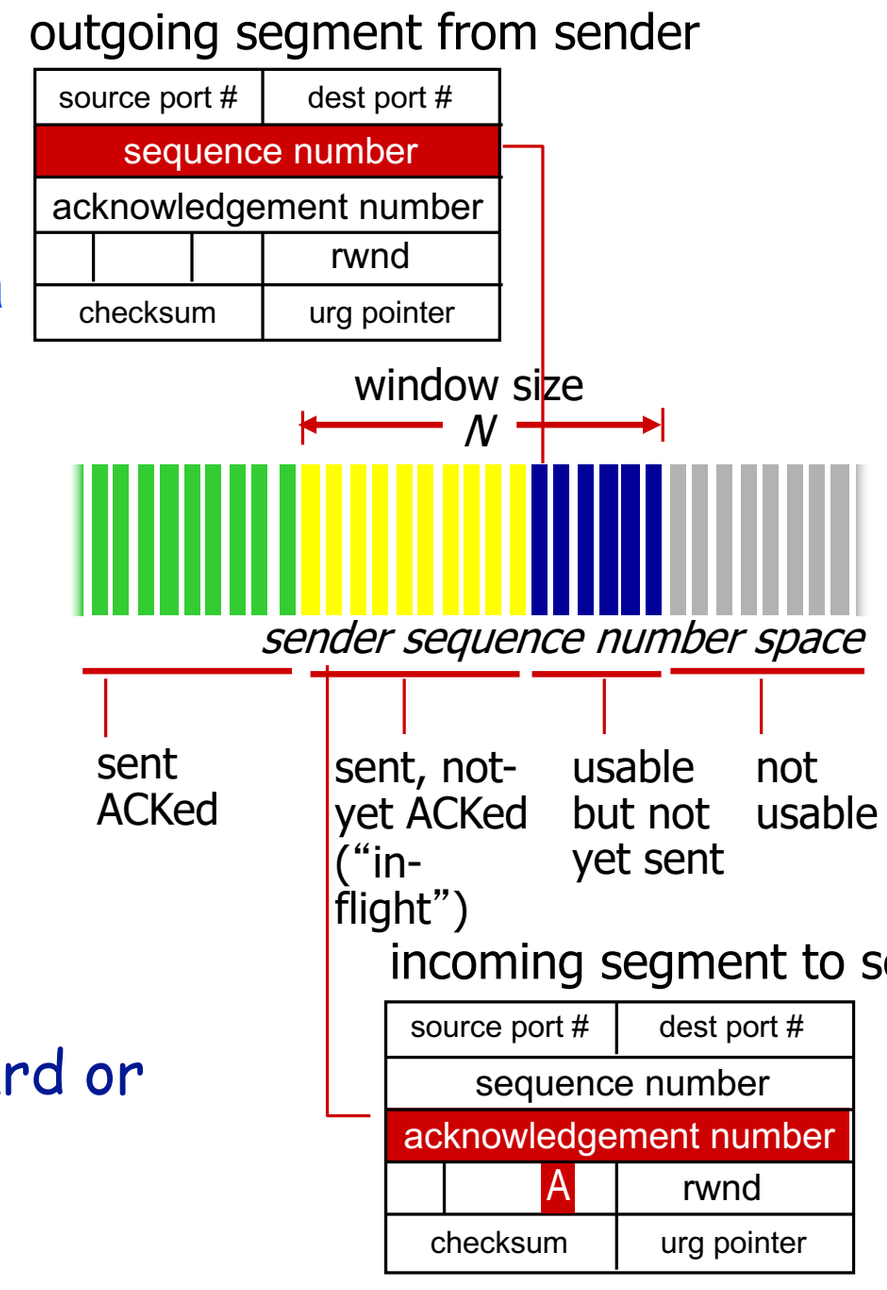
- TCP views data as ordered stream of bytes
- Seq. #
  - byte-stream number of first byte in segment's data

    not number of segments
  - initial seq. # is randomly chosen
- ACK #
  - seq. # of next byte expected from the other side
  - cumulative ACK

    acknowledges bytes up to the first missing byte
  - ACK piggybacked in data segments
- How does receiver handle out-of-order bytes: discard or buffer?
  - TCP RFCs do not specify

    up to implementor

    in practice, buffer (as in Selective Repeat)



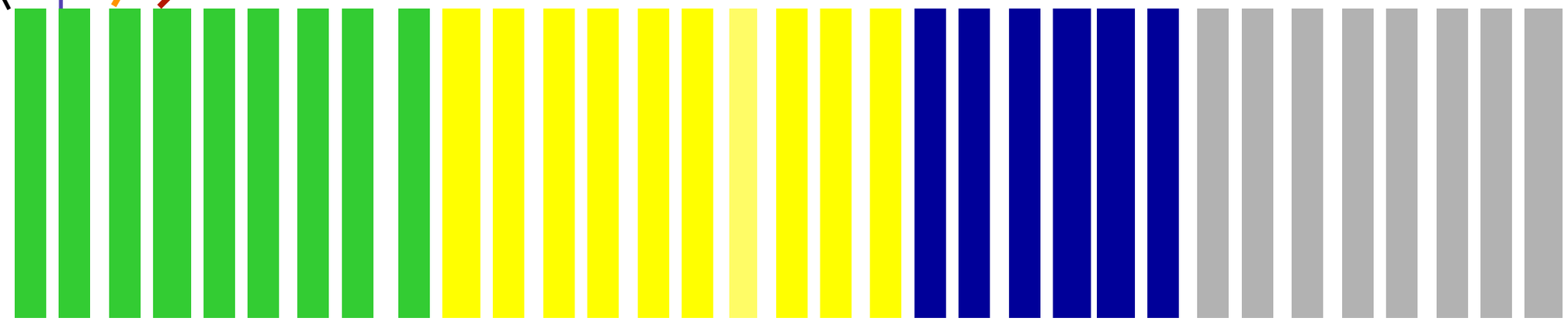
Sequence number

2126

2156

2191

2216



A TCP data stream consisting of multiple data chunks (each chunk to be enclosed in one TCP segment)

Chunk size

30 bytes

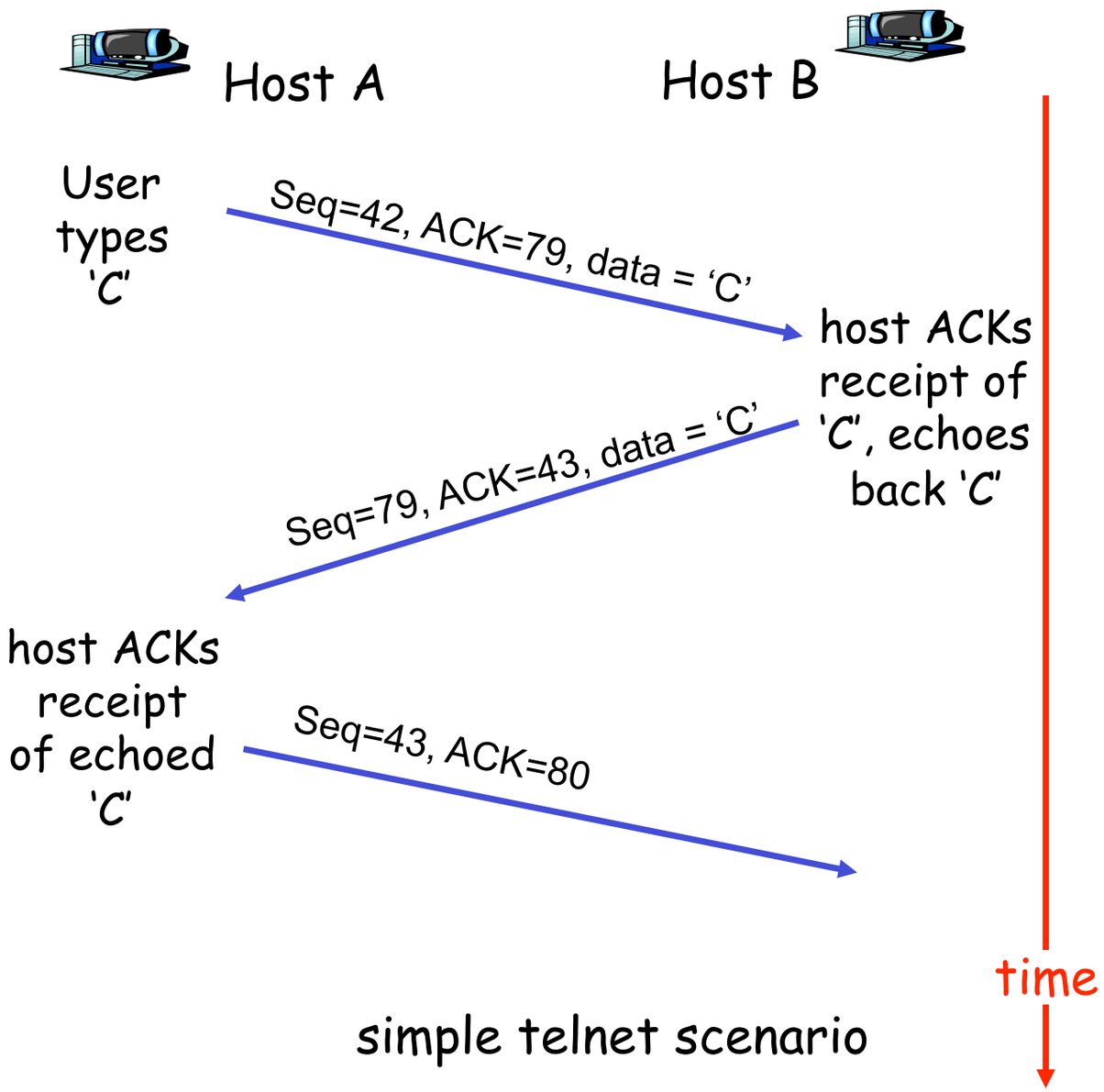
35 bytes

25 bytes

TCP: seq. #'s and ACKs (an example)

Telnet

- application-layer protocol for remote login
- runs over TCP



simple telnet scenario

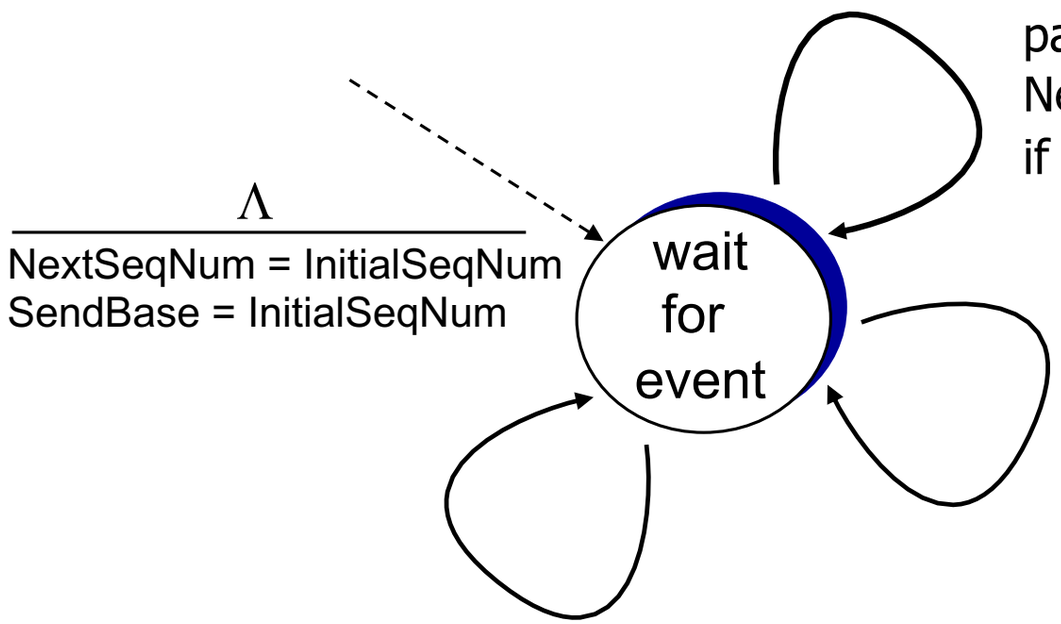
TCP reliable data transfer

- TCP creates reliable data transfer service on top of IP's unreliable best-effort service
- Selective repeat or go-back-N? A hybrid.
  - Pipelined segments
  - Cumulative acks
  - Uses single retransmission timer
  - Retransmissions are triggered by:

    timeout events

    duplicate acks

TCP RDT: simplified sender



data received from application above

create segment, seq. #: NextSeqNum
pass segment to IP (i.e., "send")
NextSeqNum = NextSeqNum + length(data)
if (timer currently not running)
    start timer

data received from app layer

- create segment with seq. #
- start timer if not already running

  timer is for oldest unacked segment

timeout

- retransmit oldest segment that is not acked
- restart timer

```
if (y > SendBase) {
    SendBase = y
    /* SendBase–1: last cumulatively ACKed byte */
    if (there are currently not-yet-acked segments)
        start timer
      else stop timer
     }
```

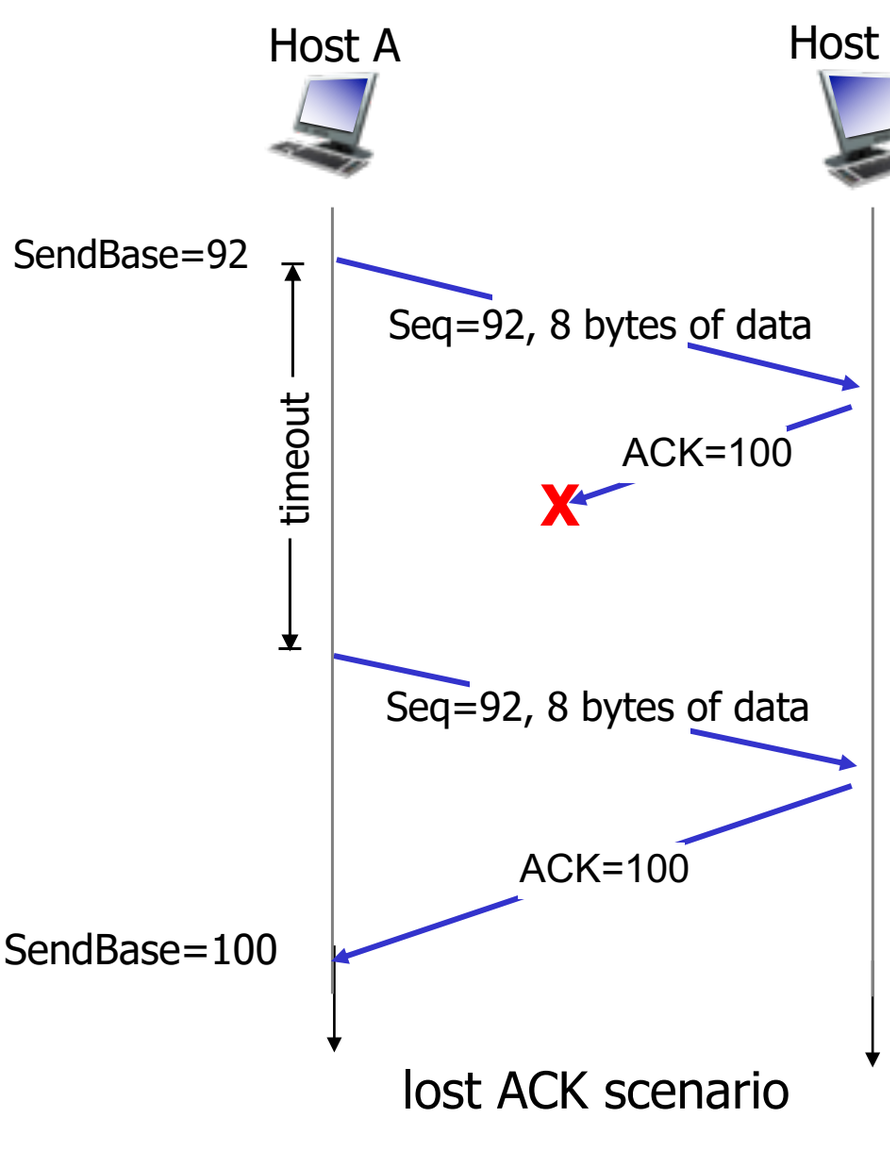
ack correctly received

- If acknowledging previously unacked segments

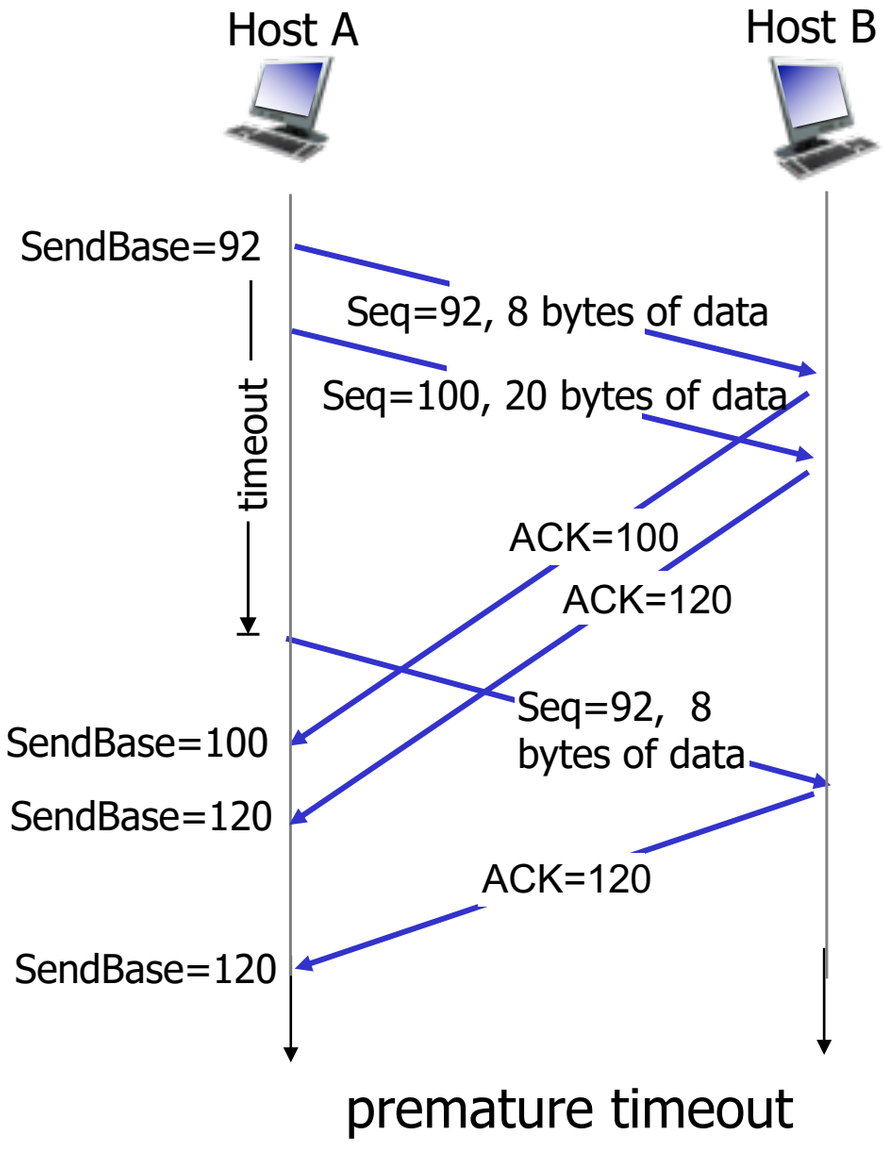
  advance send_base to next unacked seq. #

  start timer if there are unacked segments

TCP RDT: retransmission scenarios

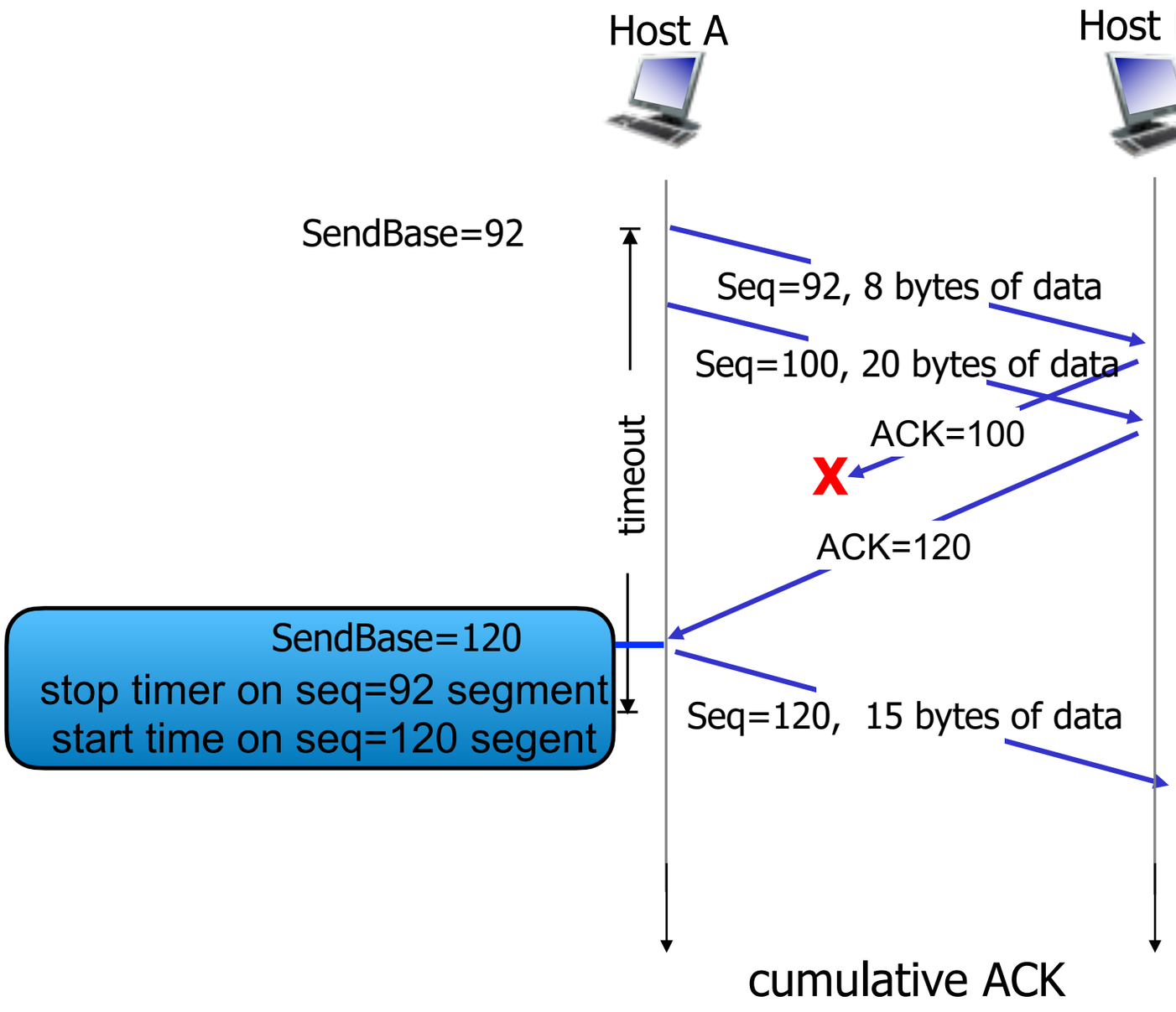


lost ACK scenario

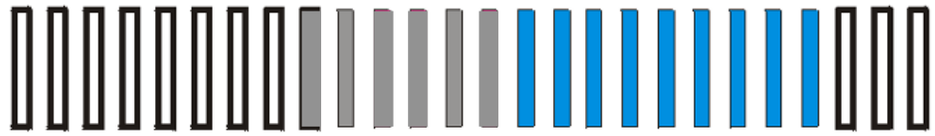


premature timeout

TCP RDT: retransmission scenarios



TCP RDT: receiver



TCP ACK Generation Recommendation [RFC 5681]

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 another in-order segment. If no next segment, send an 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

TCP RDT: fast retransmit

- Detect lost segments via timeouts
  - timeout 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.

- TCP fast retransmit: if sender receives 3 duplicate ACKs for the same data, it supposes that segment after ACKed data was lost, and resends segment before timeout

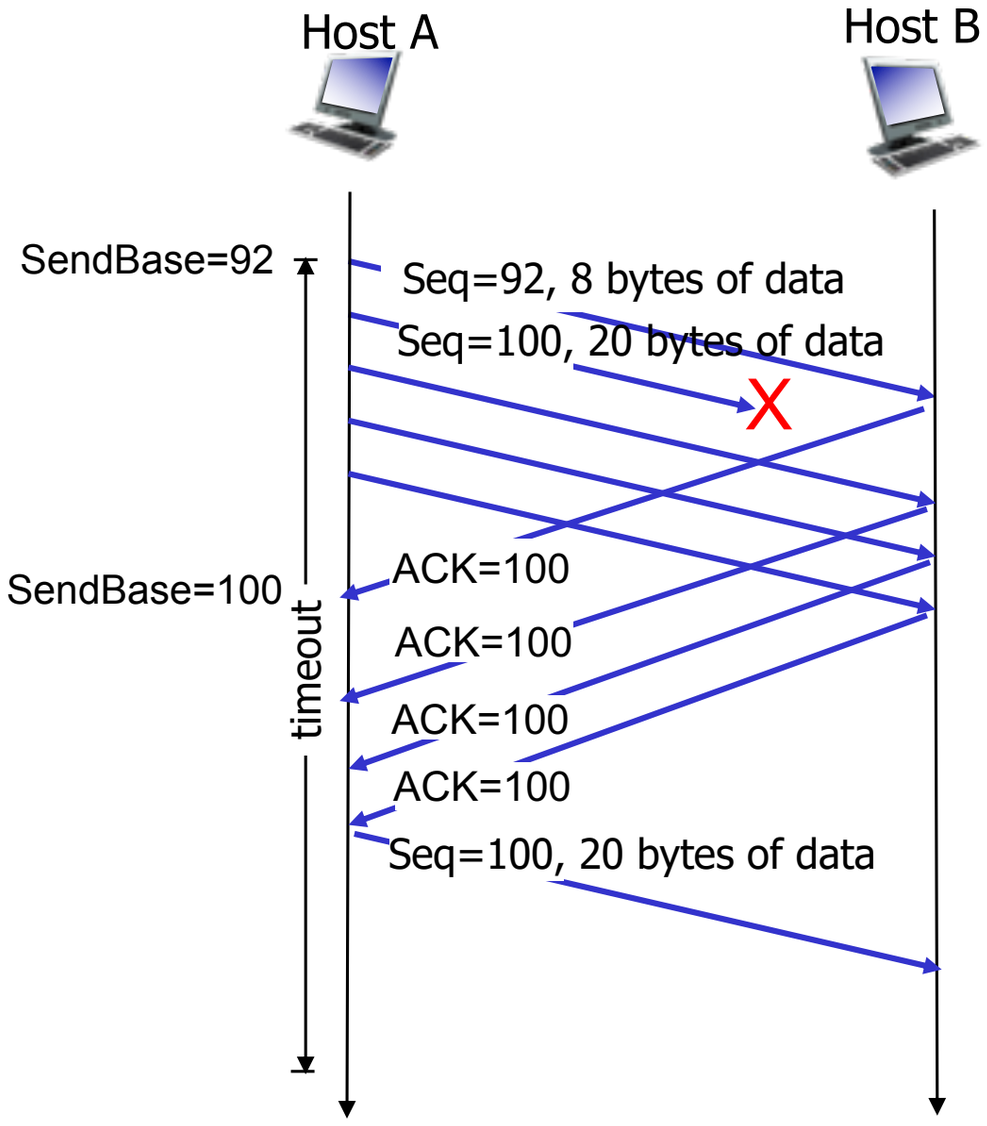
TCP RDT: fast retransmit (cont'd)

TCP sender with fast retransmit:
the following code replaces ACK received event on slide 9

```
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 stop time
                }
            else { //y=SendBase
                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

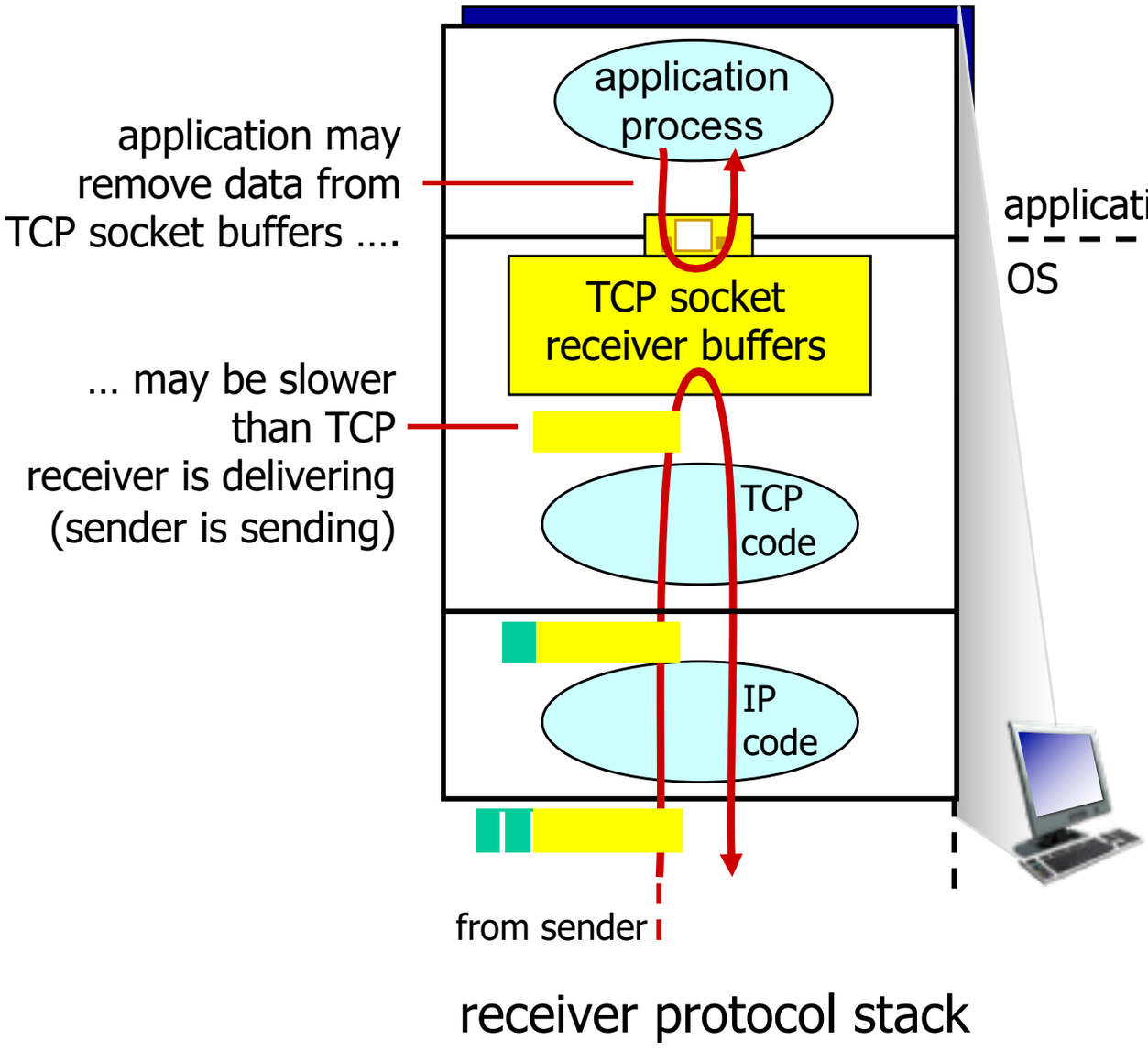


fast retransmit after sender receipt of triple duplicate ACK

TCP flow control

- Receiver side of TCP connection has a receive buffer:
- Application process may be slow at reading from buffer
- Flow control
  - sender won't overflow receiver's buffer by sending too much, too fast
  - speed-matching service

    matching the sending rate to the receiving app's drain rate



receiver protocol stack

TCP flow control (cont'd)

- How TCP flow control works
  - At TCP receiver

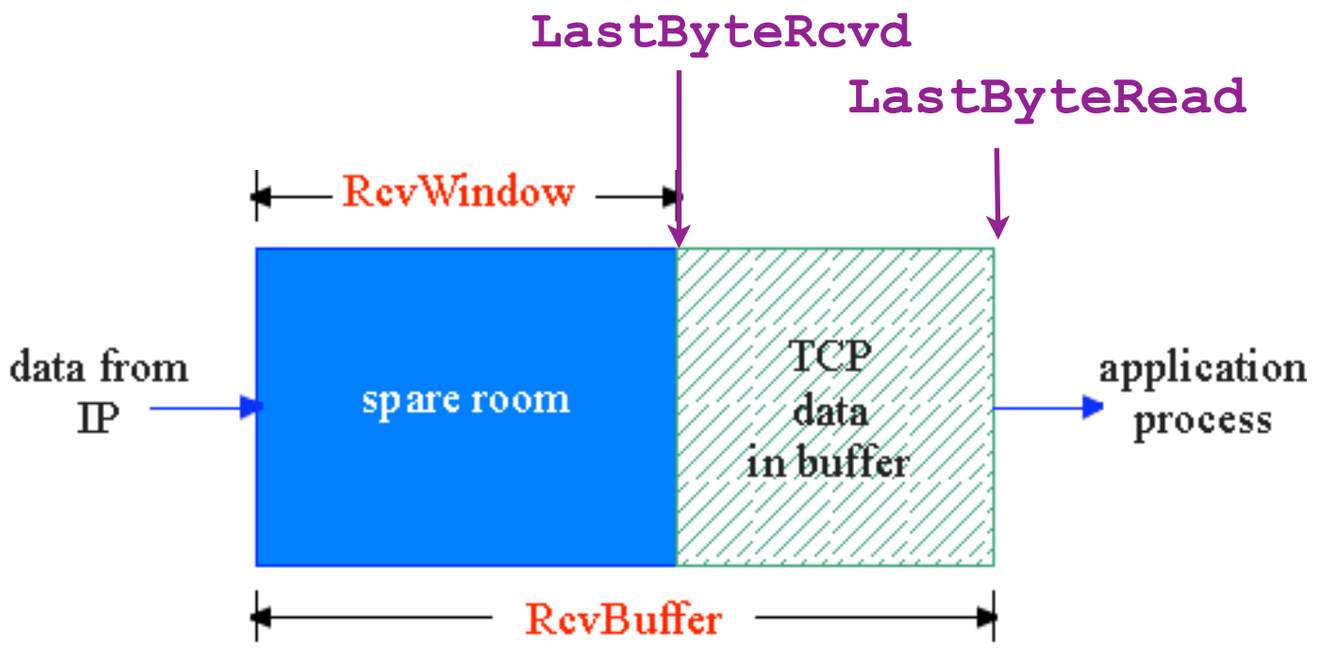
    spare room in buffer = **`RcvWindow = RcvBuffer-[LastByteRcvd - LastByteRead]`**

  - Receiver advertises spare room by including value of current **`RcvWindow`** in segments to sender

    RcvBuffer size can be set via socket options

  - Sender limits unACKed data to **`RcvWindow`**

    guarantees receive buffer does not overflow



TCP connection management: three-way handshake

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

How?

client: connection initiator
clientSocket.connect((serverName, serverPort))

server: contacted by client
connectionSocket, addr = serverSocket.accept()

Why?

initialize TCP variables:

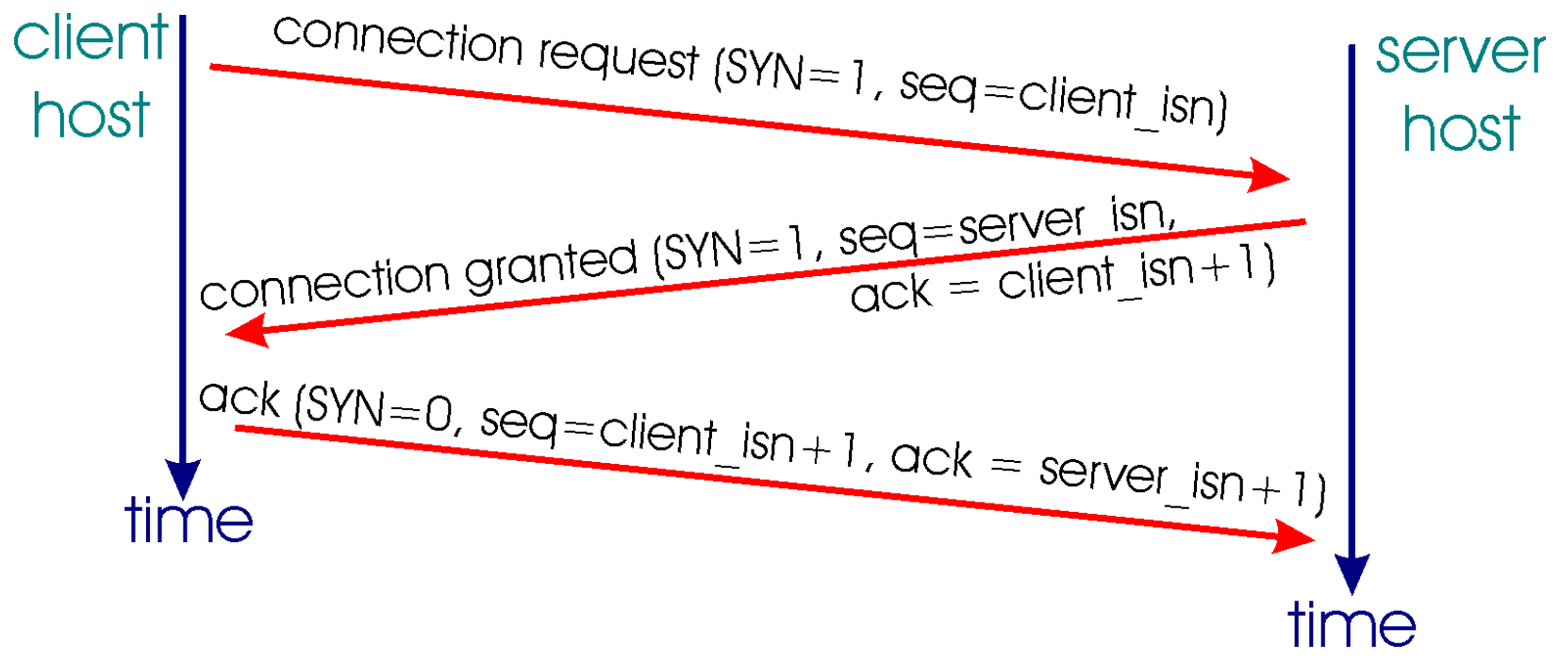
- seq. #s
- **RcvBuffer** size at client, server

Three-way handshake

Step 1: client host sends TCP SYN segment to server
- ★ specifies client 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



TCP connection management: closing a connection

Closing a connection (`socket.close()`)

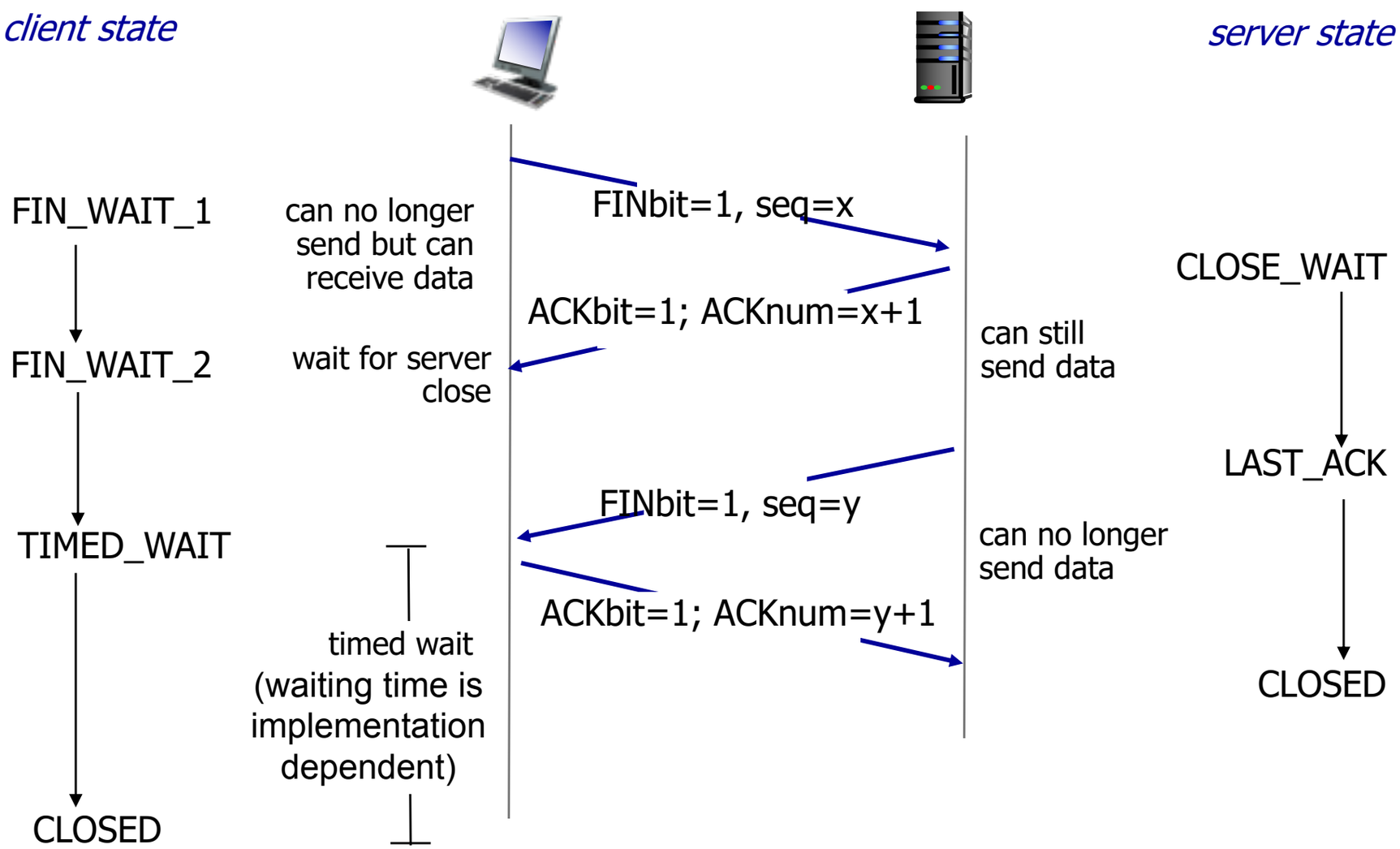
Step 1: client sends TCP FIN control segment (FIN bit is 1) to server

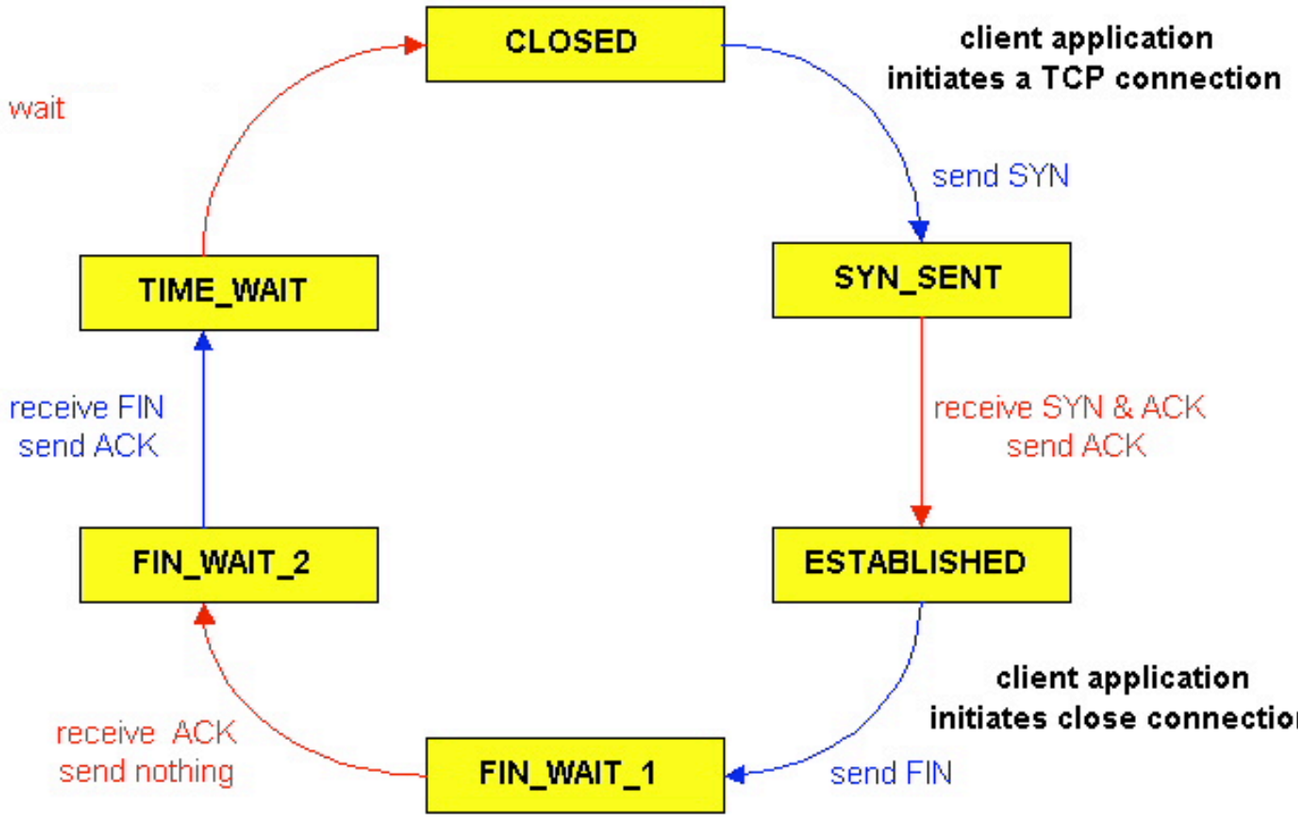
Step 2: server receives FIN, replies with ACK. Closing connection, sends FIN.

Step 3: client receives FIN, replies with ACK.

Enters “timed wait” - will respond with ACK to received FINs

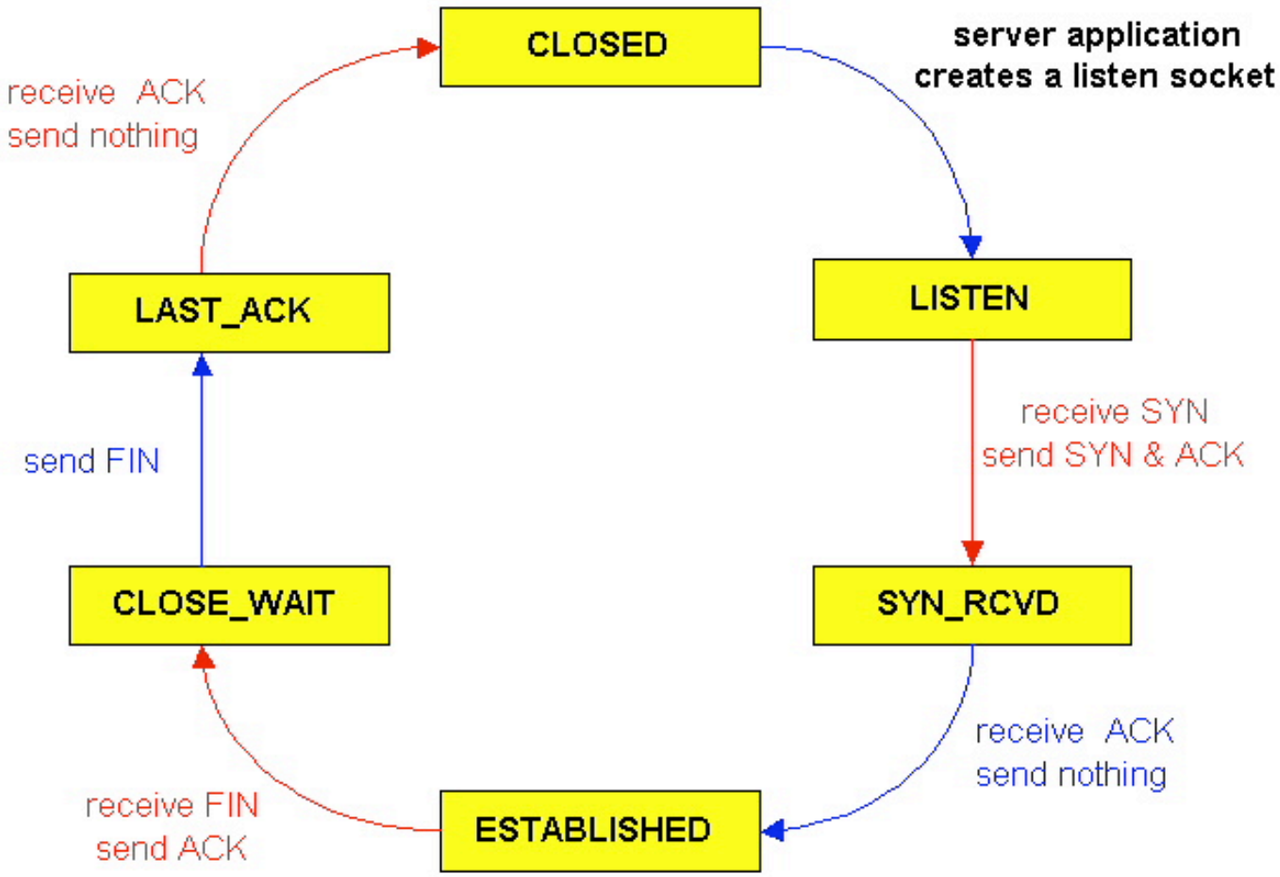
Step 4: server, receives ACK. Connection closed.





TCP client lifecycle

TCP server lifecycle



- Required Reading:
  - Computer Networking: A Top-Down Approach (8th Edition)
    Ch 3.5.1, 3.5.2, 3.5.4, 3.5.5, 3.5.6

- Acknowledgement:
  - Some materials are extracted from the slides created by Prof. Jim F. Kurose and Prof. Keith W. Ross for the textbook.