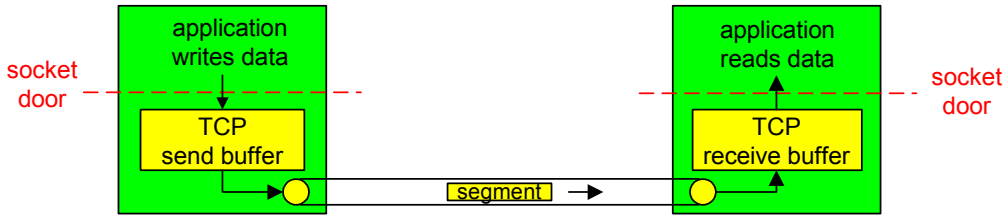
TCP: Overview

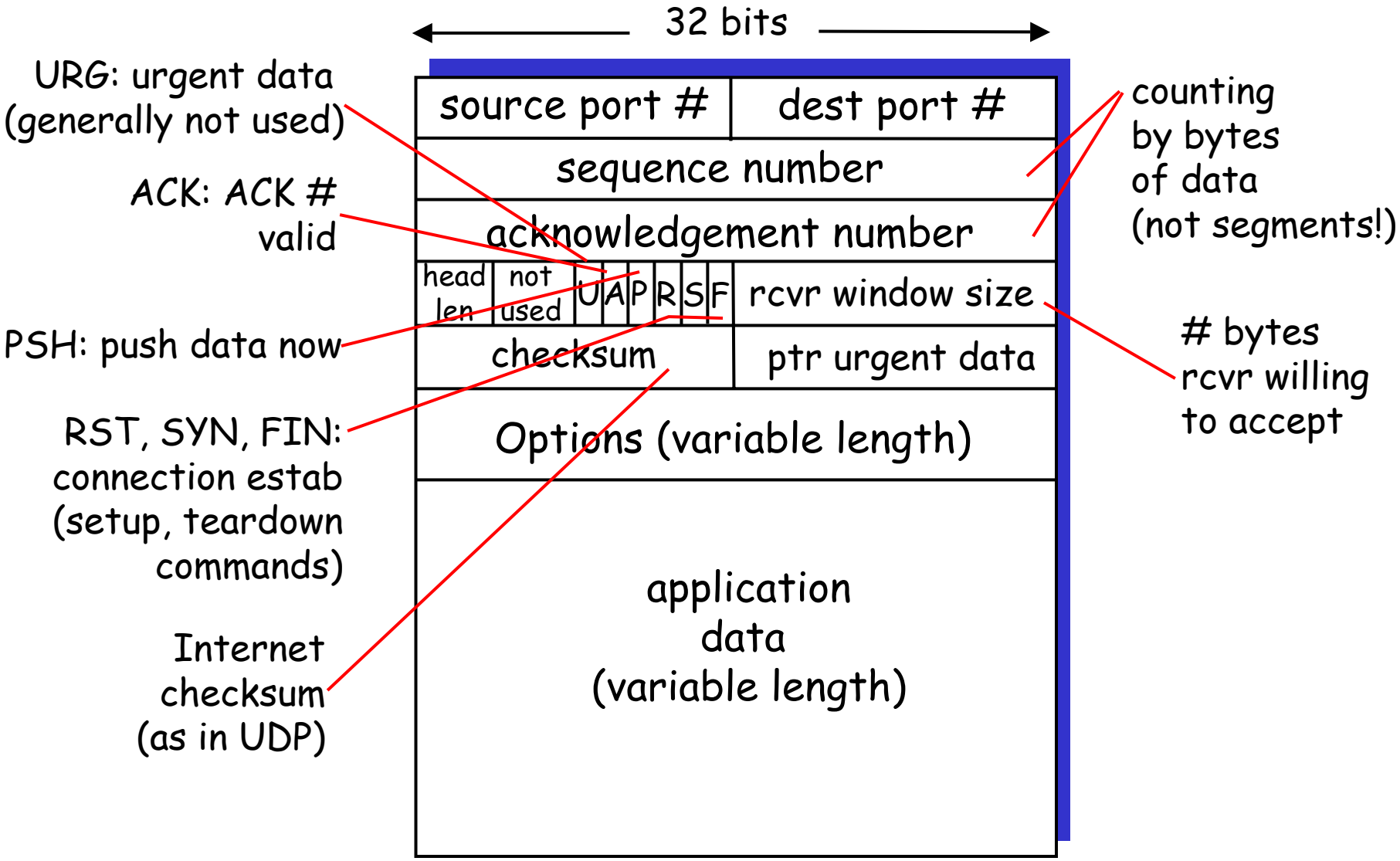
RFCs: 793, 1122, 1323, 2018, 2581

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte stream:*
  - 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:

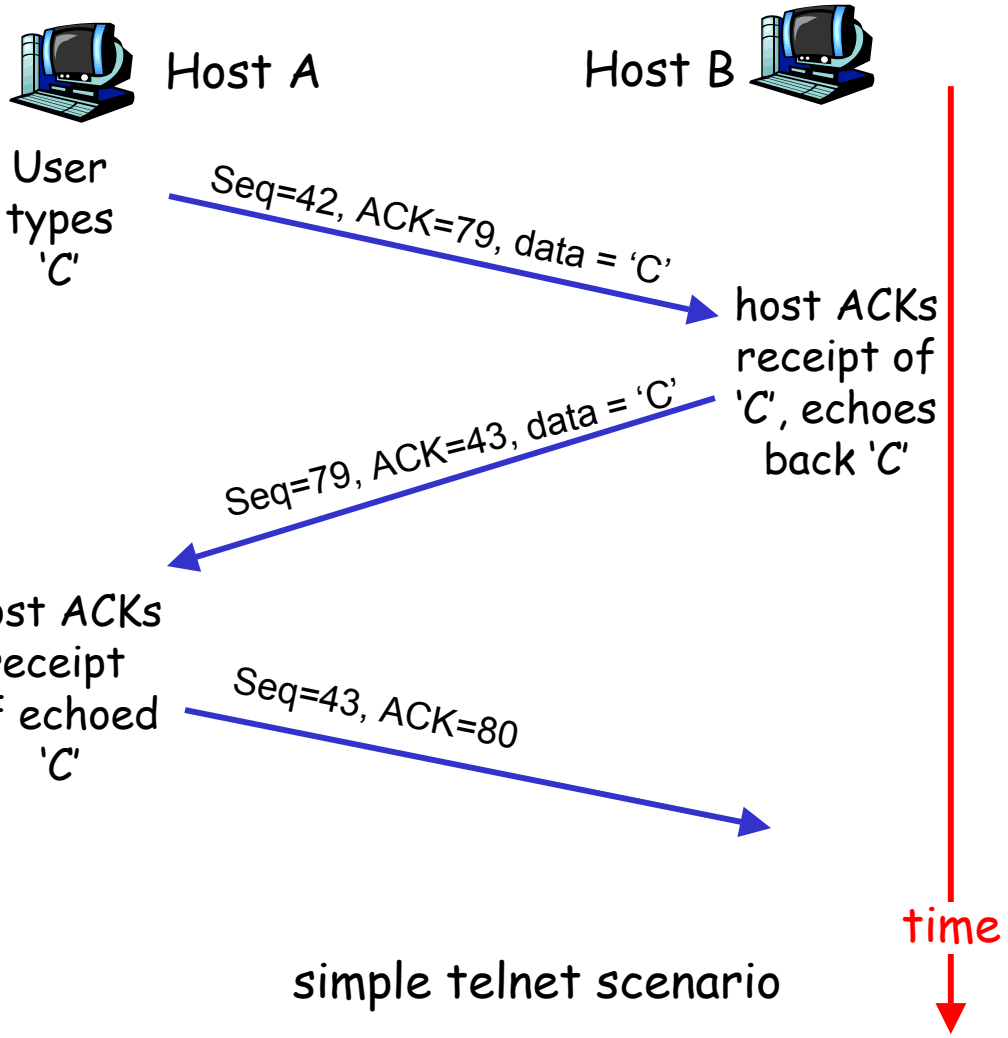
- byte stream "number" of first byte in segment's data

ACKs:

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

Q: how receiver handles out-of-order segments

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



simple telnet scenario

TCP: reliable data transfer

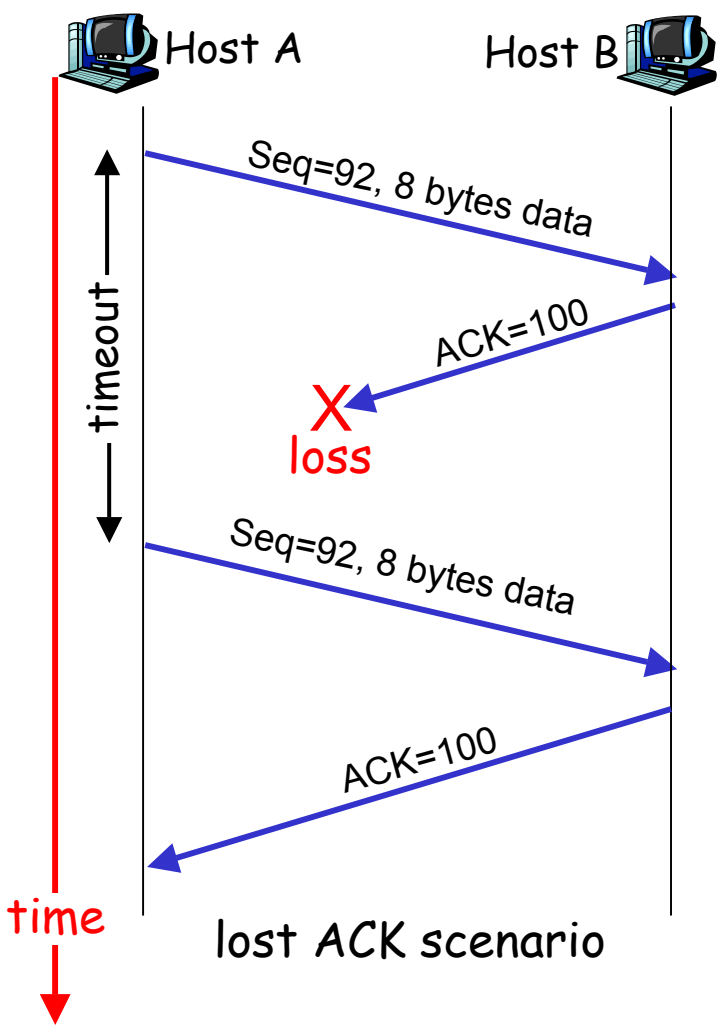
Simplified TCP sender

```
00  sendbase = initial_sequence number
01   nextseqnum = initial_sequence number
02
03    loop (forever) {
04      switch(event)
05      event: data received from application above
06           create TCP segment with sequence number nextseqnum
07           start timer for segment nextseqnum
08           pass segment to IP
09           nextseqnum = nextseqnum + length(data)
10       event: timer timeout for segment with sequence number y
11           retransmit segment with sequence number y
12           compute new timeout interval for segment y
13           restart timer for sequence number y
14       event: ACK received, with ACK field value of y
15           if (y > sendbase) { /* cumulative ACK of all data up to y */
16               cancel all timers for segments with sequence numbers < y
17                sendbase = y
18                }
19           else { /* a duplicate ACK for already ACKed segment */
20                increment number of duplicate ACKs received for y
21                if (number of duplicate ACKS received for y == 3) {
22                   /* TCP fast retransmit */
23                   resend segment with sequence number y
24                   restart timer for segment y
25               }
26      }  /* end of loop forever */
```

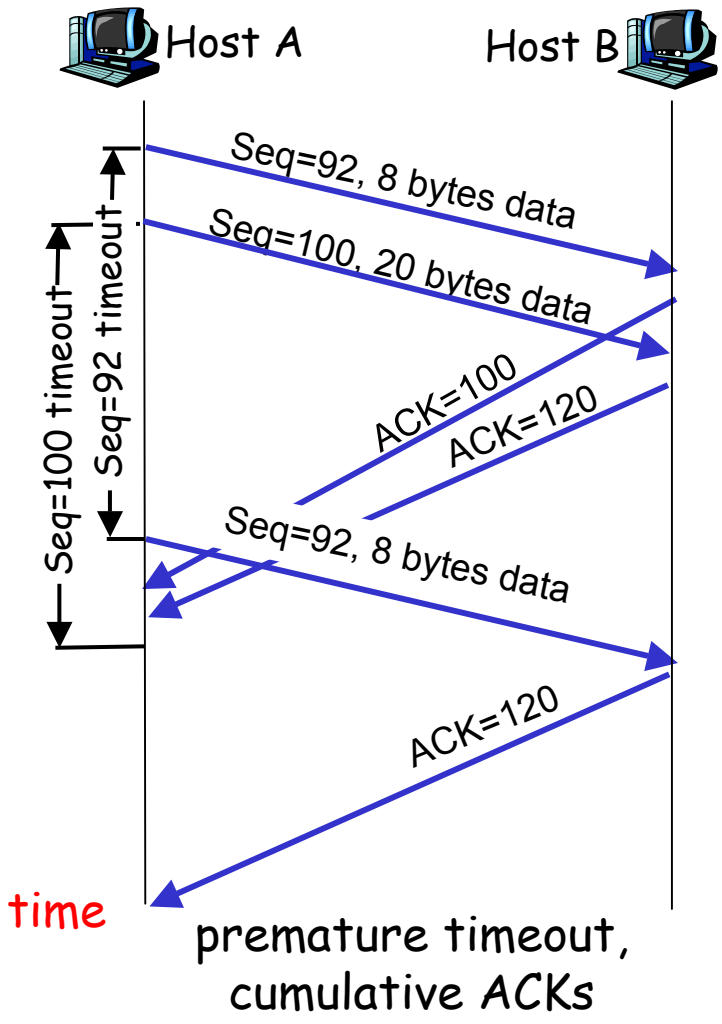
TCP ACK generation [RFC 1122, RFC 2581]

Event	TCP Receiver action
in-order segment arrival, no gaps, everything else already ACKed	delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
in-order segment arrival, no gaps, one delayed ACK pending	immediately send single cumulative ACK
out-of-order segment arrival higher-than-expect seq. # gap detected	send duplicate ACK, indicating seq. # of next expected byte
arrival of segment that partially or completely fills gap	immediate ACK if segment starts at lower end of gap

TCP: retransmission scenarios



lost ACK scenario



premature timeout, cumulative ACKs

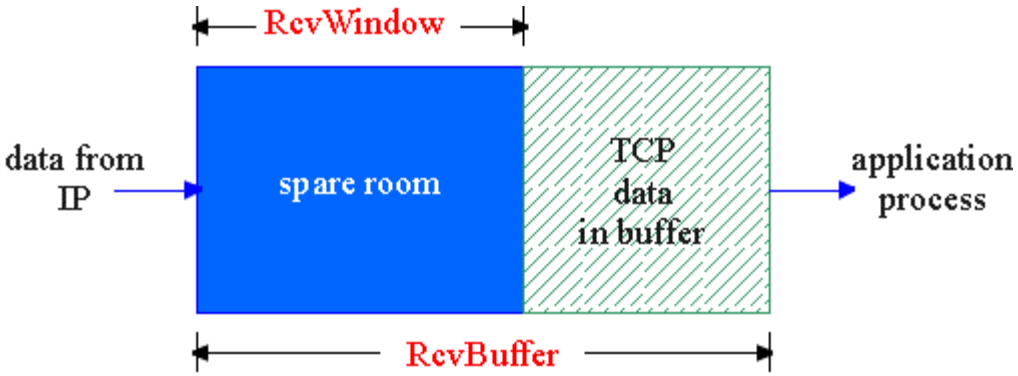
TCP Flow Control

flow control

sender won't overrun receiver's buffers by transmitting too much, too fast

RcvBuffer = size or TCP Receive Buffer

RcvWindow = amount of spare room in Buffer



receiver buffering

receiver: explicitly informs sender of (dynamically changing) amount of free buffer space

- **RcvWindow field** in TCP segment

sender: keeps the amount of transmitted, unACKed data less than most recently received **RcvWindow**

TCP Round Trip Time and Timeout

Q: how to set TCP timeout value?

- longer than RTT
  - note: RTT will vary
- 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, cumulatively ACKed segments
- **SampleRTT** will vary, want estimated RTT "smoother"
  - use several recent measurements, not just current **SampleRTT**

TCP Round Trip Time and Timeout

EstimatedRTT = (1-x)*EstimatedRTT + x*SampleRTT

- Exponential weighted moving average
- influence of given sample decreases exponentially fast
- typical value of x: 0.1

Setting the timeout

- **EstimtedRTT** plus "safety margin"
- large variation in **EstimatedRTT ->** larger safety margin

Timeout = EstimatedRTT + 4*Deviation

Deviation = (1-x)*Deviation +
x*|SampleRTT-EstimatedRTT|

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

```
Socket clientSocket = new
Socket("hostname","port
number");
```

- *server:* contacted by client

```
Socket connectionSocket =
welcomeSocket.accept();
```

Three way handshake:

Step 1: client end system sends TCP SYN control segment to server
- specifies initial seq #

Step 2: server end system receives SYN, replies with SYNACK control segment
- ACKs received SYN
- allocates buffers
- specifies server-> receiver initial seq. #

TCP Connection Management (cont.)

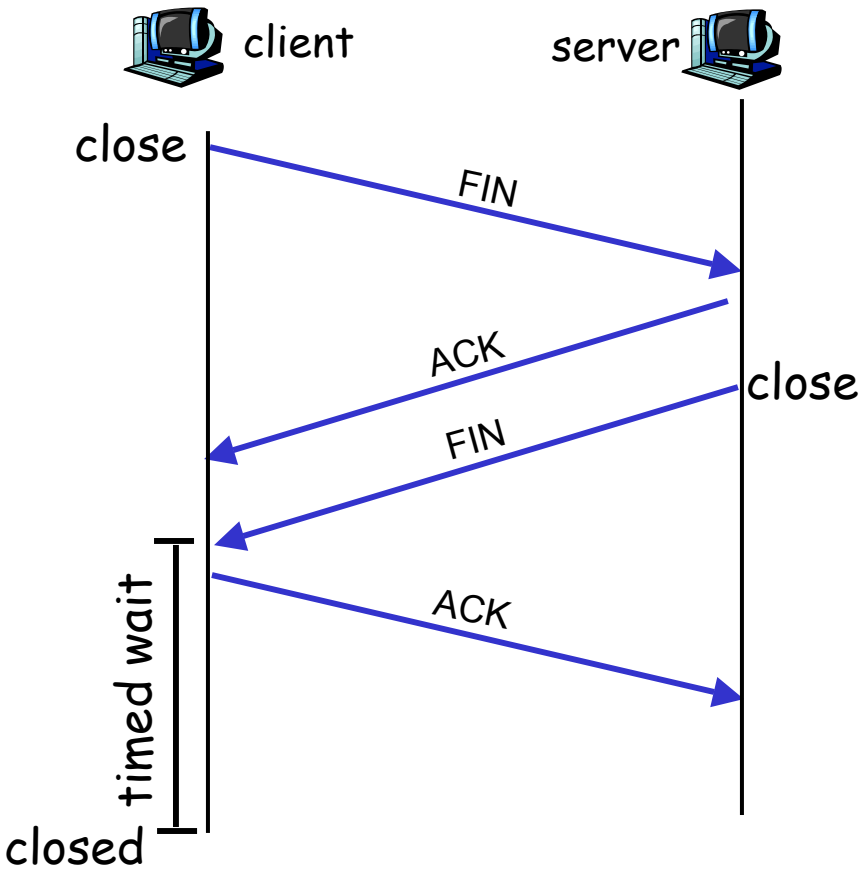
Closing a connection:

client closes socket:

```
clientSocket.close();
```

Step 1: client end system sends TCP FIN control segment to server

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



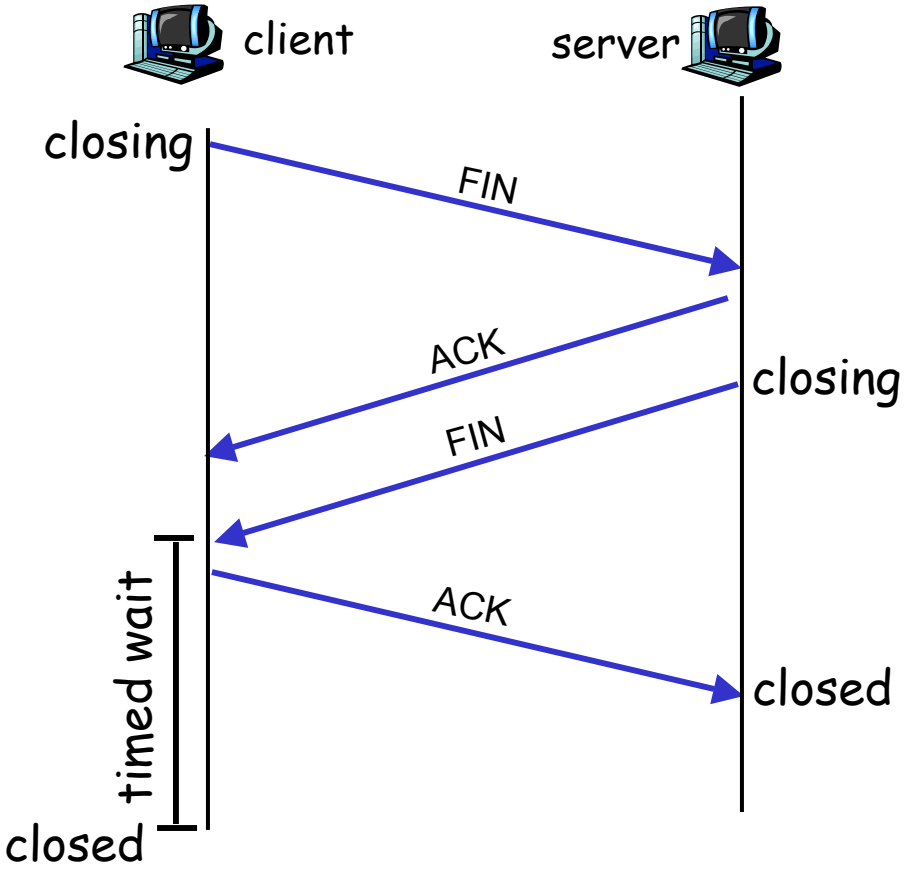
TCP Connection Management (cont.)

Step 3: client receives FIN, replies with ACK.

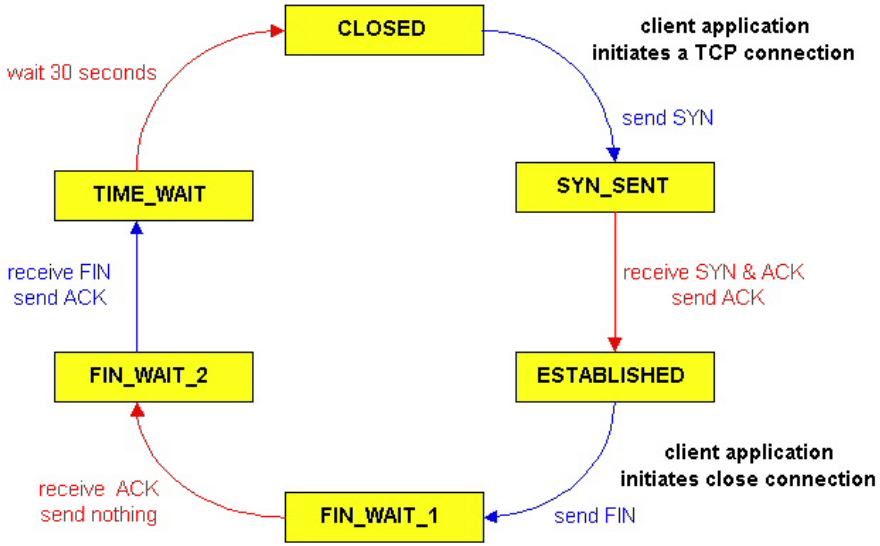
- Enters "timed wait" - will respond with ACK to received FINs

Step 4: server, receives ACK. Connection closed.

Note: with small modification, can handle simultaneous FINs.



TCP Connection Management (cont)



TCP client lifecycle

TCP server lifecycle

