

Chapter 3

Transport Layer

Chapter 3 outline

3.1 Transport-layer services

3.2 Multiplexing and demultiplexing

3.3 Connectionless transport: UDP

3.4 Principles of reliable data transfer

3.5 Connection-oriented transport: TCP

- ◆ segment structure
- ◆ reliable data transfer
- ◆ flow control
- ◆ connection management

3.6 Principles of congestion control

3.7 TCP congestion control

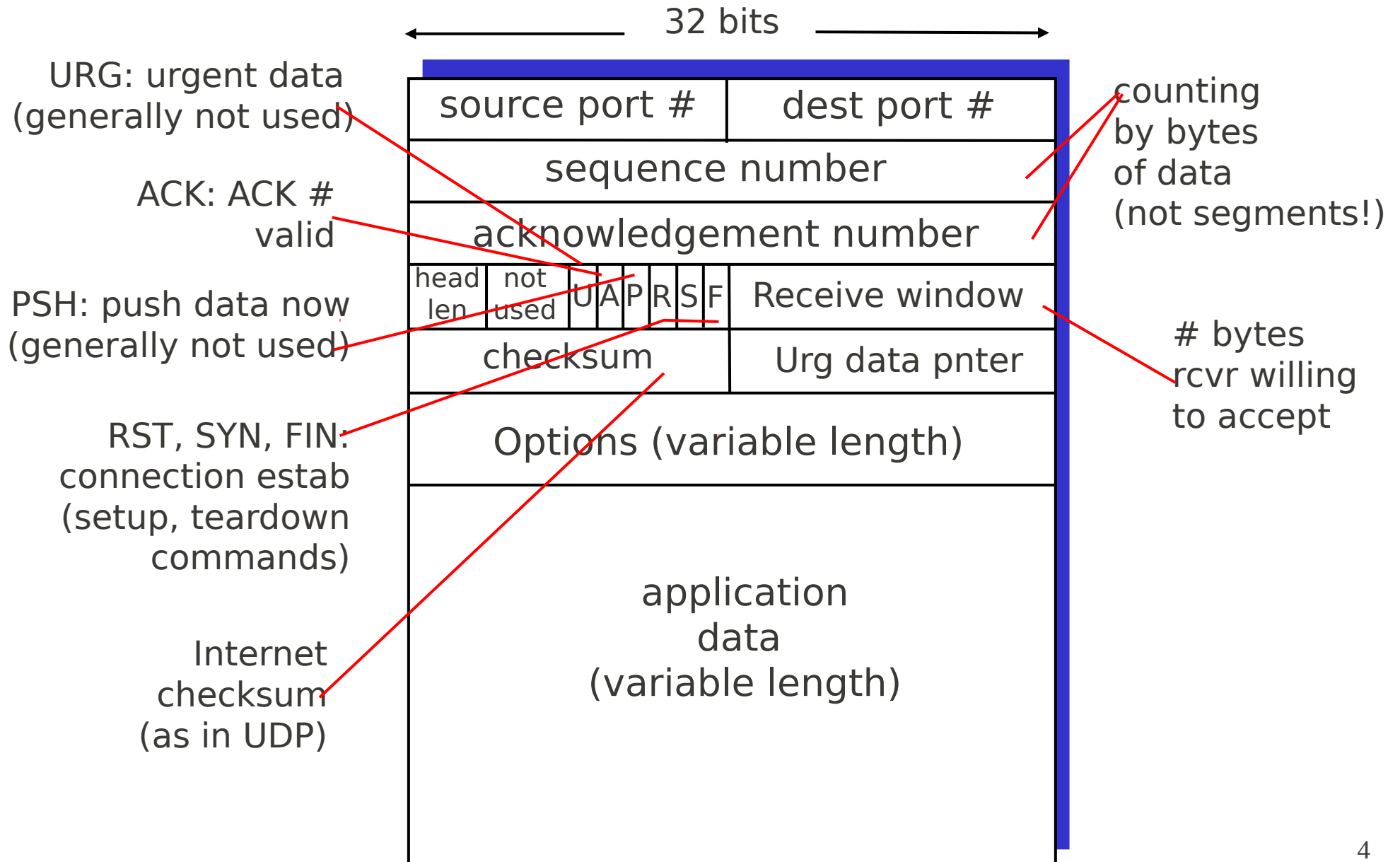
TCP: Overview

RFCs: 793, 1122, 1323, 2018, 2581

- **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) initializes sender, receiver state before data exchange
- **flow controlled:**
 - sender will not overwhelm receiver



TCP segment structure



TCP seq. #'s and ACKs

Seq. #'s:

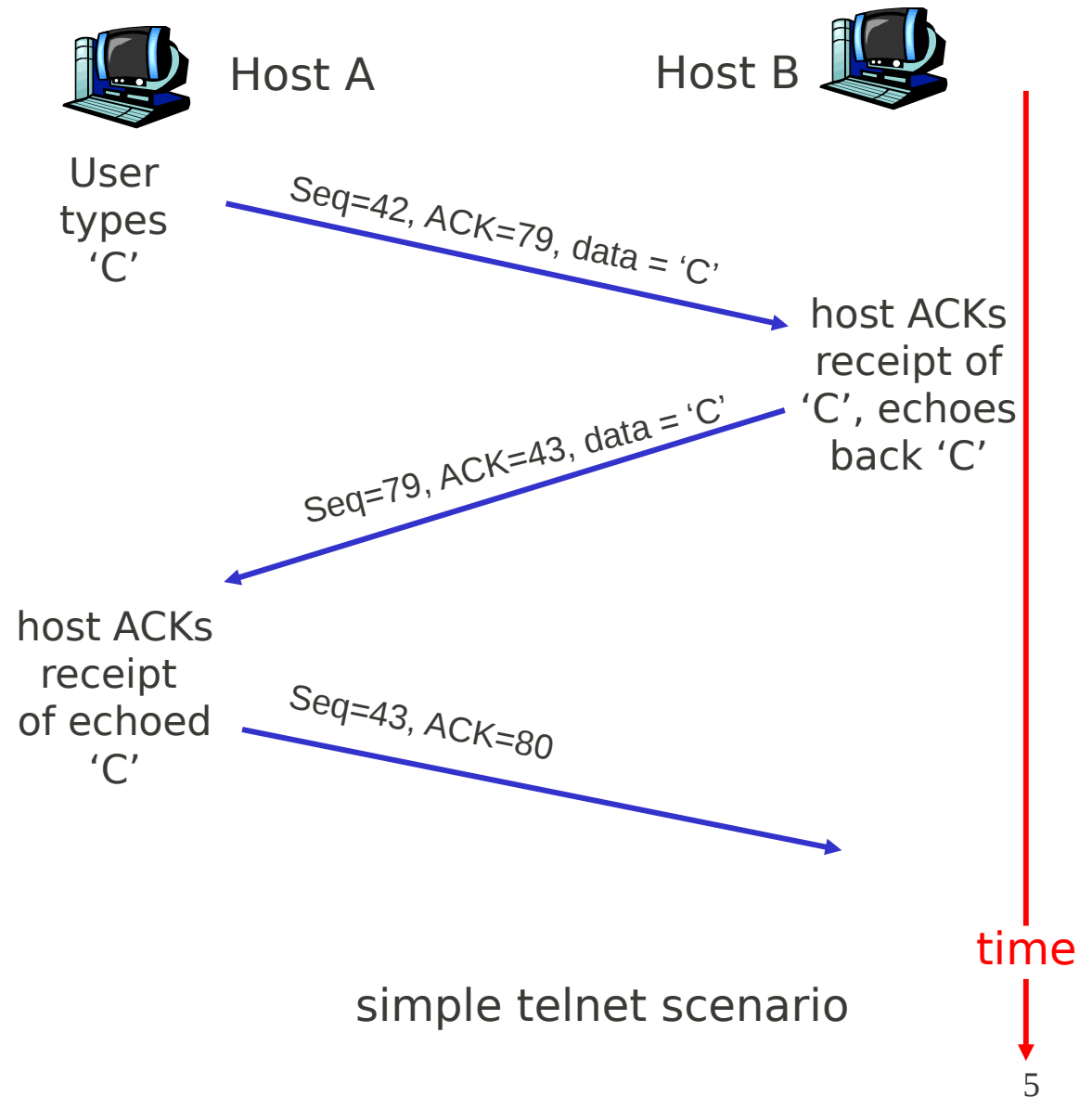
- byte stream “number” of first byte in segment’s data

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



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

TCP Round Trip Time and Timeout

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**

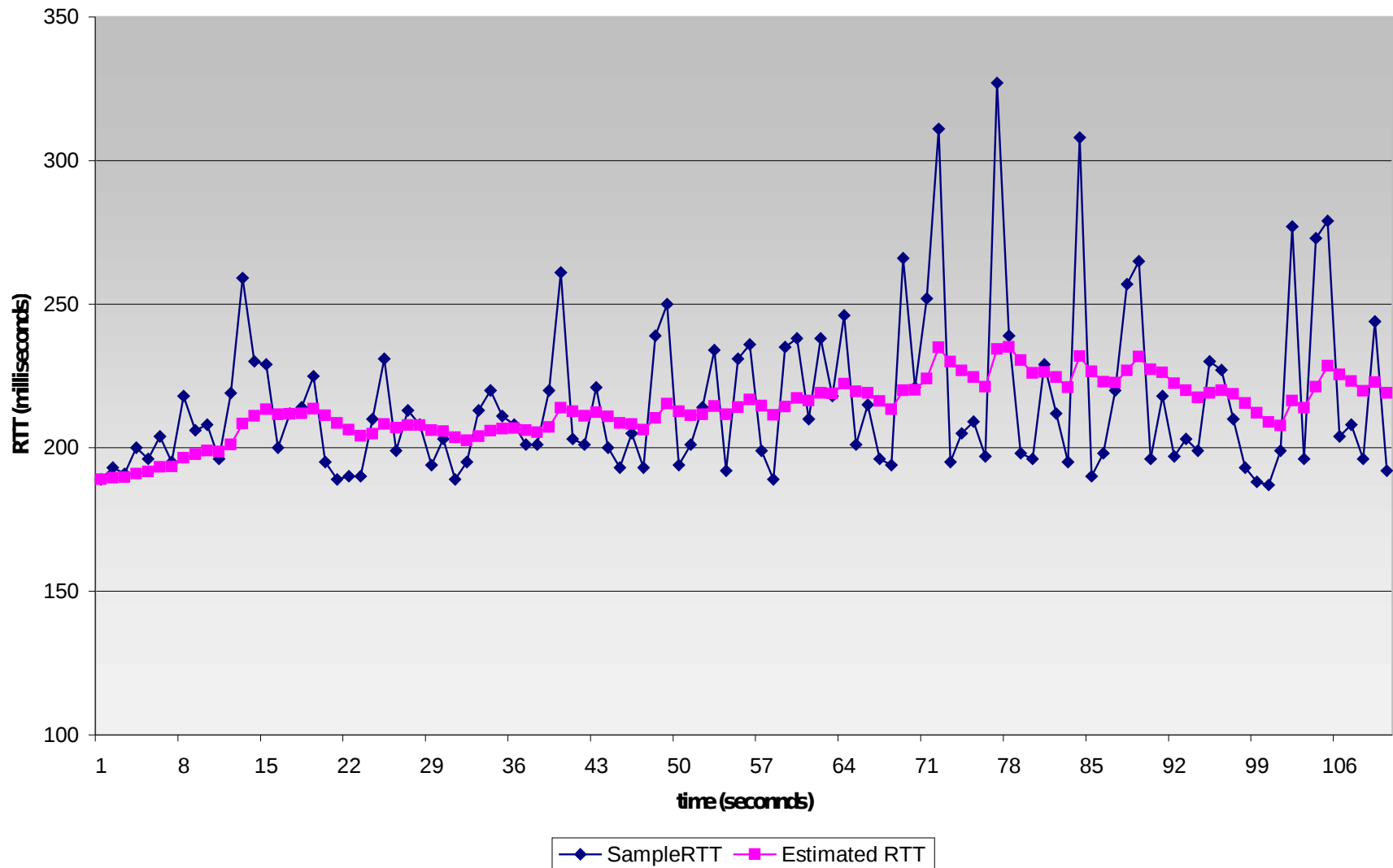
TCP Round Trip Time and Timeout

$$\text{EstimatedRTT} = (1 - \alpha) * \text{EstimatedRTT} + \alpha * \text{SampleRTT}$$

- ❖ Exponential weighted moving average
- ❖ influence of past sample decreases exponentially fast
- ❖ typical value: $\alpha = 0.125$

Example RTT estimation:

RTT: ~~gaia.cs.umass.edu~~ to fantasia.eurecom.fr



TCP Round Trip Time and Timeout

Setting the timeout

- **EstimatedRTT** plus “safety margin”
 - large variation in **EstimatedRTT** -> larger safety margin
- first estimate of how much SampleRTT deviates from EstimatedRTT:

$$\text{DevRTT} = (1-\beta) * \text{DevRTT} + \beta * |\text{SampleRTT} - \text{EstimatedRTT}|$$

(typically, $\beta = 0.25$)

Then set timeout interval:

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4 * \text{DevRTT}$$

Chapter 3 outline

3.1 Transport-layer services

3.2 Multiplexing and demultiplexing

3.3 Connectionless transport: UDP

3.4 Principles of reliable data transfer

3.5 Connection-oriented transport: TCP

- ◆ segment structure
- ◆ reliable data transfer
- ◆ flow control
- ◆ connection management

3.6 Principles of congestion control

3.7 TCP congestion control

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

NextSeqNum = InitialSeqNum

SendBase = InitialSeqNum

```
loop (forever) {  
    switch(event)
```

event: data received from application above
 create TCP segment with sequence number NextSeqNum
 if (timer currently not running)
 start timer
 pass segment to IP
 NextSeqNum = NextSeqNum + length(data)

event: timer timeout
 retransmit not-yet-acknowledged segment with
 smallest sequence number
 start timer

event: ACK received, with ACK field value of y
 if (y > SendBase) {
 SendBase = y
 if (there are currently not-yet-acknowledged segments)
 start timer
 }

```
} /* end of loop forever */
```

TCP sender (simplified)

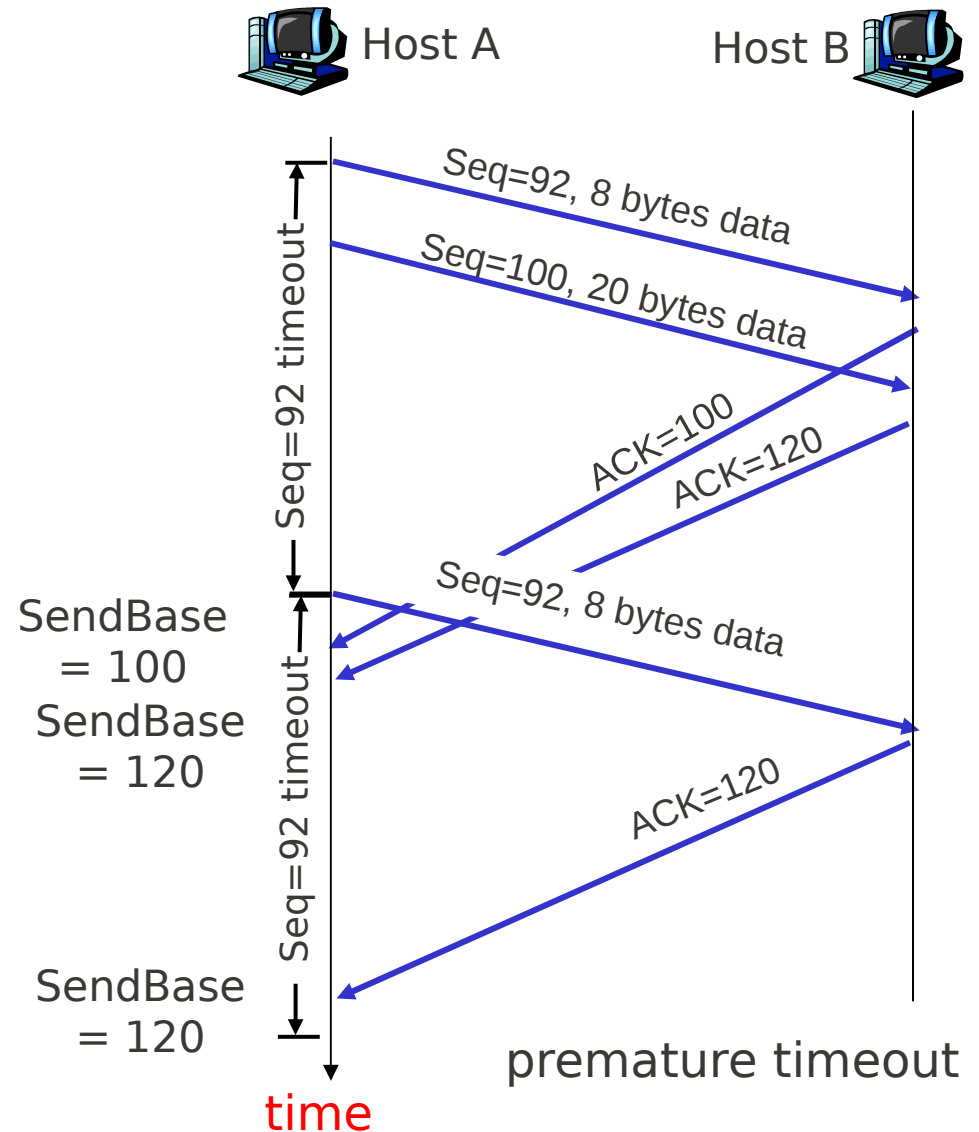
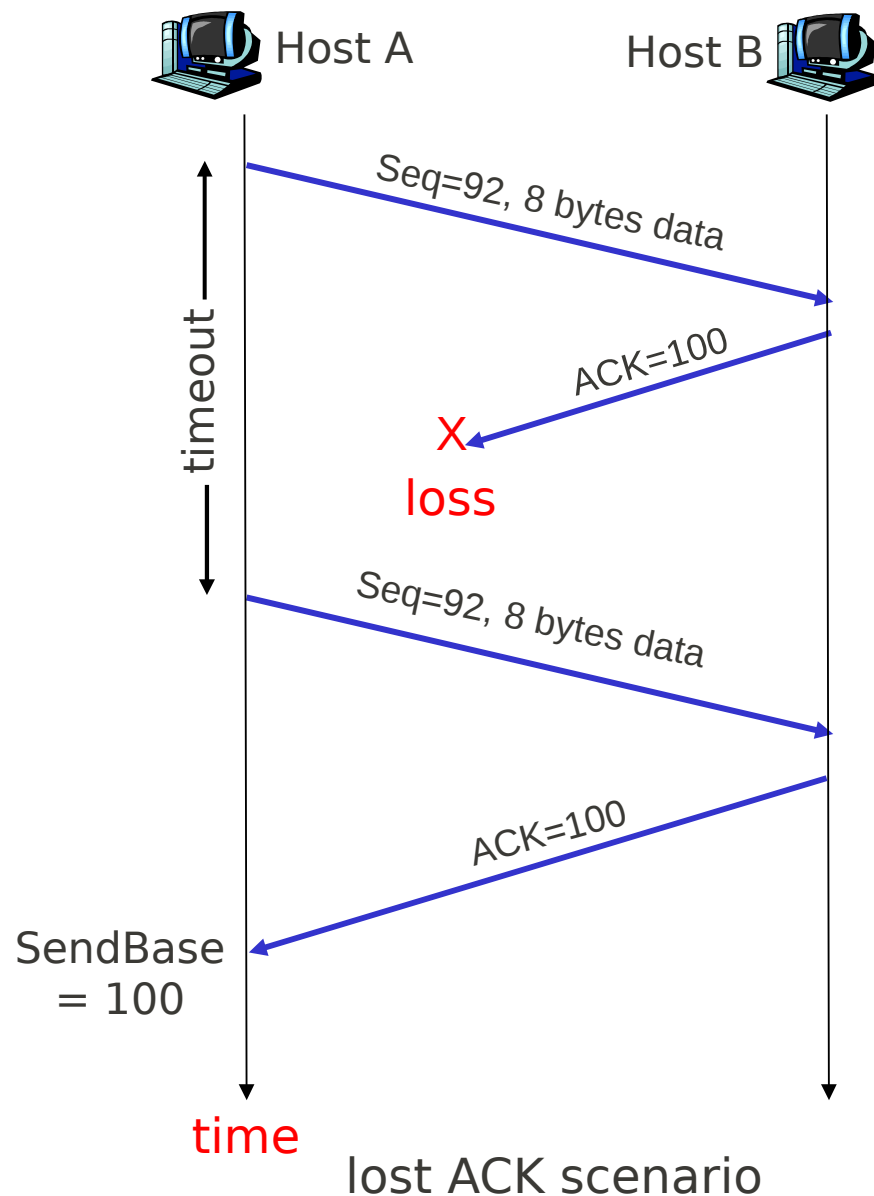
Comment:

- SendBase-1: last cumulatively acked byte

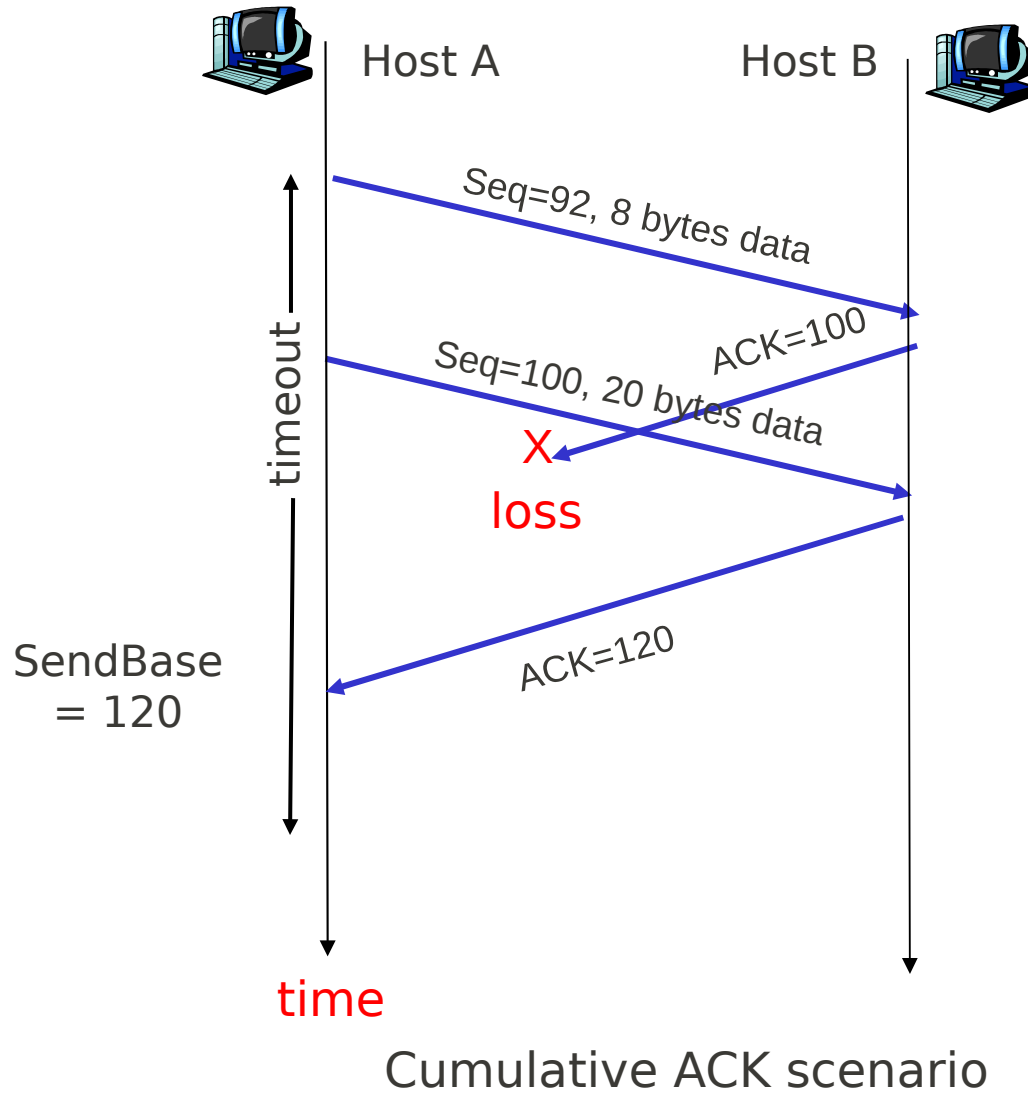
Example:

- SendBase-1 = 71;
y = 73, so the rcvr wants 73+ ;
y > SendBase, so that new data is acked

TCP: retransmission scenarios



TCP retransmission scenarios (more)



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 starts at 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

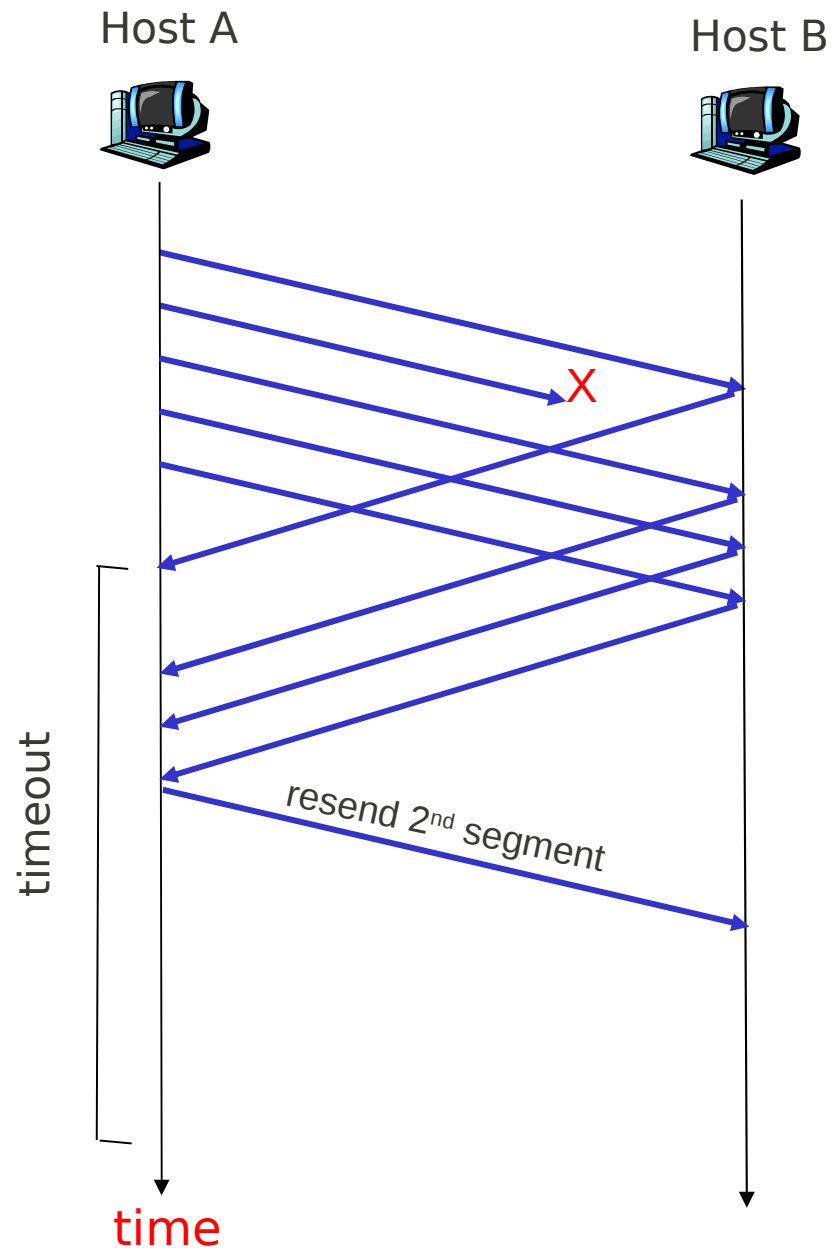


Figure 3.37 Resending a segment after triple duplicate ACK

Fast retransmit algorithm:

```
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 {
        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

Chapter 3 outline

3.1 Transport-layer services

3.2 Multiplexing and demultiplexing

3.3 Connectionless transport: UDP

3.4 Principles of reliable data transfer

3.5 Connection-oriented transport: TCP

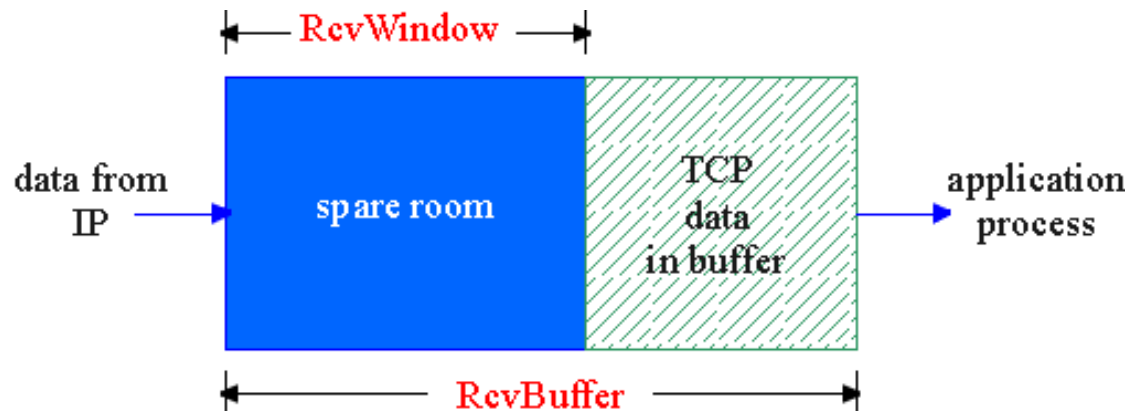
- ◆ segment structure
- ◆ reliable data transfer
- ◆ **flow control**
- ◆ connection management

3.6 Principles of congestion control

3.7 TCP congestion control

TCP Flow Control

- receive side of TCP connection has a receive buffer:



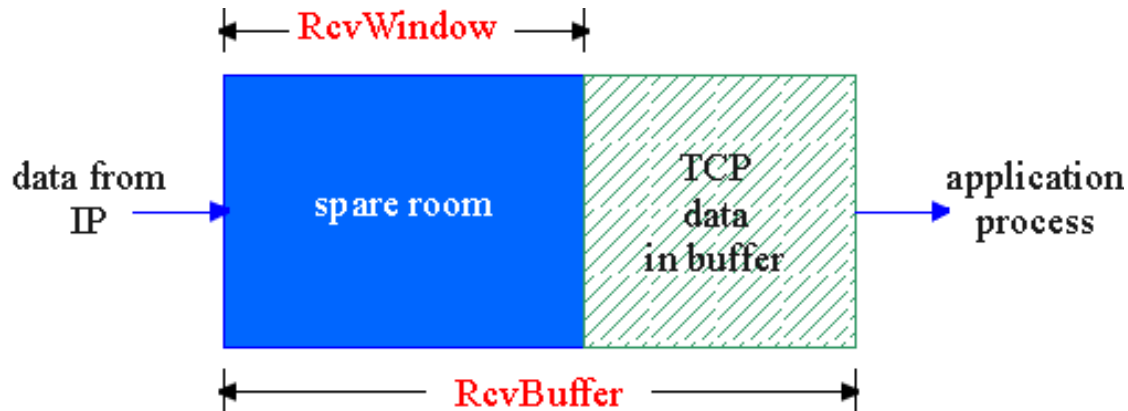
- ❖ app process may be slow at reading from buffer

flow control

sender won't overflow receiver's buffer by transmitting too much, too fast

- speed-matching service: matching the send rate to the receiving app's drain rate

TCP Flow control: how it works



(suppose TCP receiver discards out-of-order segments)

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

- rcvr advertises spare room by including value of **RcvWindow** in segments
- sender limits unACKed data to **RcvWindow**
 - guarantees receive buffer doesn't overflow

Chapter 3 outline

3.1 Transport-layer services

3.2 Multiplexing and demultiplexing

3.3 Connectionless transport: UDP

3.4 Principles of reliable data transfer

3.5 Connection-oriented transport: TCP

- ◆ segment structure
- ◆ reliable data transfer
- ◆ flow control
- ◆ connection management

3.6 Principles of congestion control

3.7 TCP congestion control

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 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

TCP Connection Management (cont.)

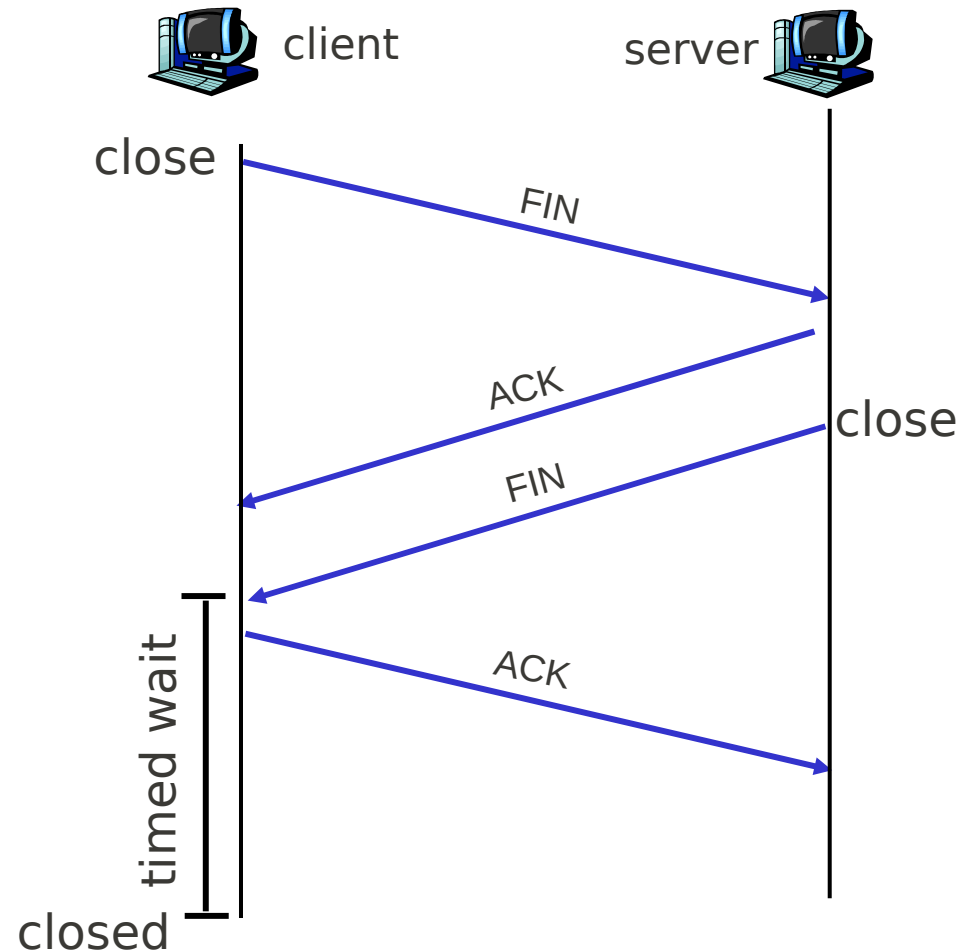
Closing a connection:

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.



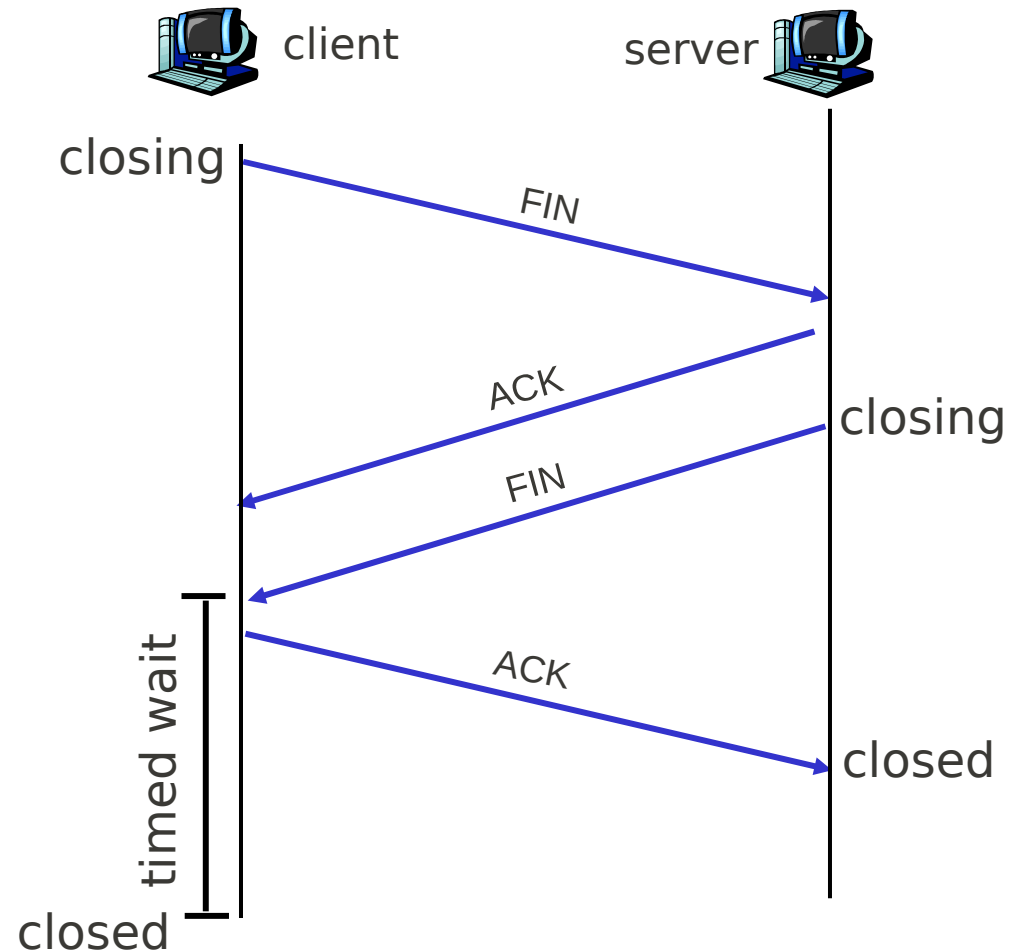
TCP Connection Management (cont.)

Step 3: client receives FIN,
replies with ACK.

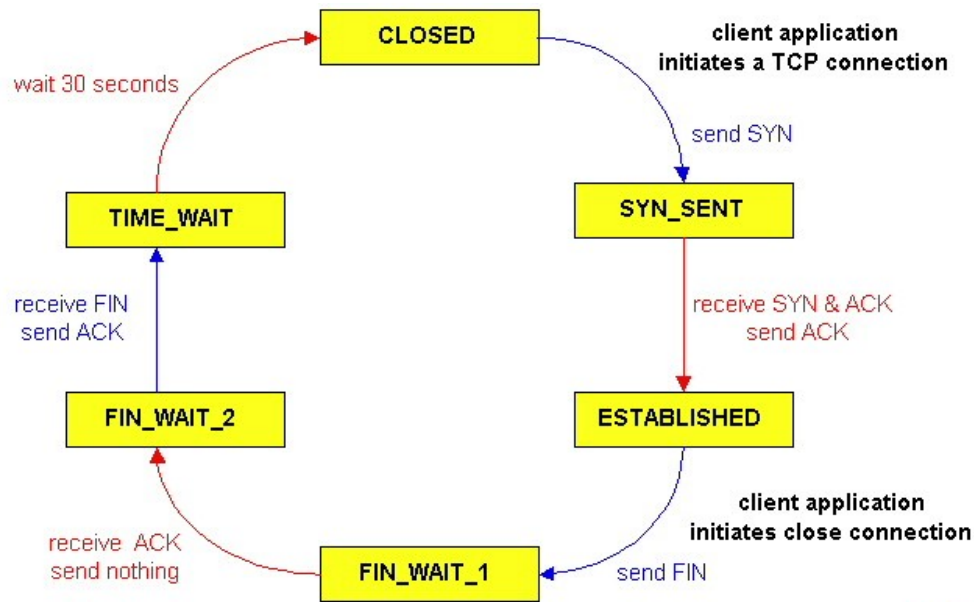
- Enters “timed wait” - will respond with ACK to received FINs

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

