

CSCI 491-01

Topics: Internet Programming

Fall 2008

Transport Layer

Derek Leonard

Hendrix College

October 15, 2008

Chapter 3: Roadmap

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 Reliable Data Transfer

- TCP creates rdt service on top of IP's unreliable service
 - Combination of Go-back-N and Selective Repeat
- Pipelined segments
- Cumulative acks
- TCP uses single retransmission timer
 - For the oldest unACK'ed packet
- Retransmissions are triggered by:
 - Timeout events
 - Duplicate acks
- Initially consider simplified TCP sender:
 - Ignore duplicate acks
 - Ignore flow control, congestion control

TCP Sender Events:

Data received from app:

- Create a segment with next 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: Timeout (n)

Timeout:

- Retransmit segment that caused timeout
- Restart timer

Ack received:

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

```
NextSeqNum = InitialSeqNum // random for each transfer
SendBase = InitialSeqNum // SendBase-1 is the last ack'ed byte
```

```
loop (forever) {
```

```
  switch(event) {
```

```
    (a) data received from application above (assuming it fits into window):
```

```
      create TCP segment with sequence number NextSeqNum
```

```
      if (timer currently not running)
```

```
          start timer
```

```
      pass segment to IP
```

```
      NextSeqNum = NextSeqNum + length(data)
```

```
    (b) timer timeout:
```

```
      retransmit not-yet-acknowledged segment with smallest
```

```
      sequence number; start timer
```

```
    (c) ACK received, with ACK field value of y
```

```
      if (y > SendBase) {
```

```
        SendBase = y
```

```
        if (there are currently not-yet-acknowledged segments)
```

```
            start timer
```

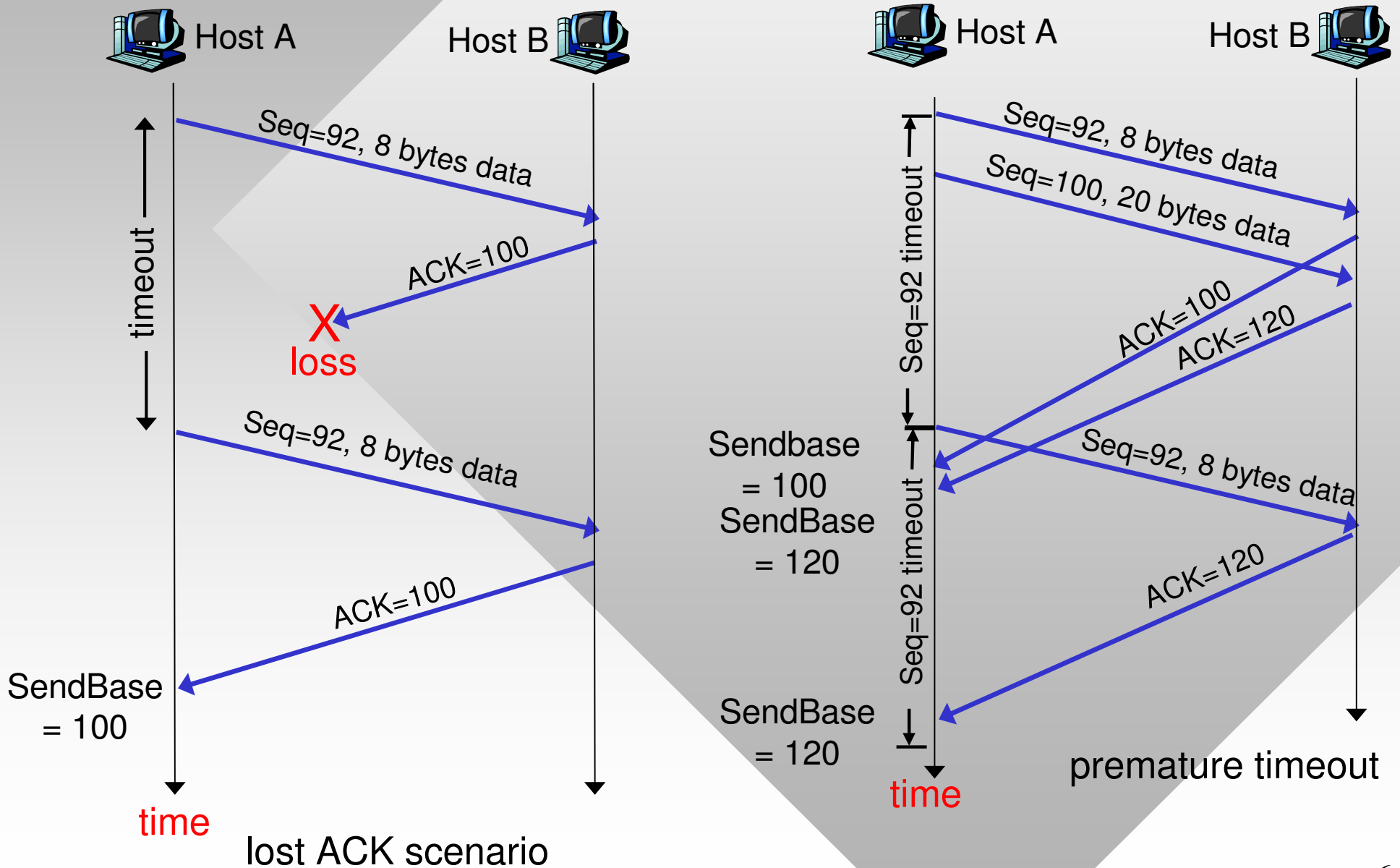
```
        else cancel timer }
```

```
      }
```

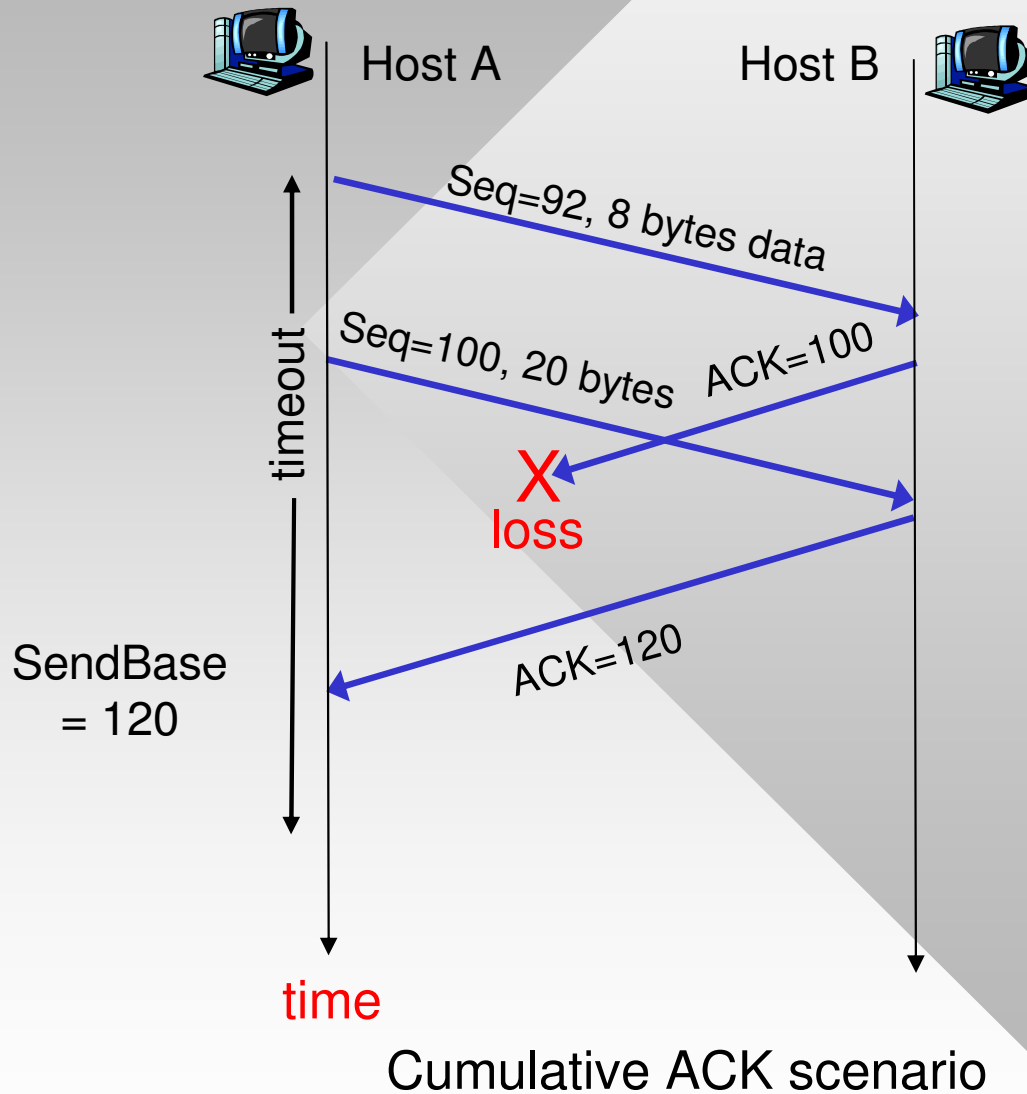
```
    } /* end of loop forever */
```

TCP Sender (Simplified)

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-expected seq. # .
Gap detected

Immediately send duplicate ACK, indicating seq. # of next expected byte

Arrival of segment that partially or completely fills gap

Immediately send ACK

Fast Retransmit

- Time-out period often relatively long:
 - Long delay before resending lost packet
- Idea: **infer** loss 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 duplicate ACKs for the same data, it supposes that segment after ACKed data was lost:
 - **Fast Retransmit**: resend segment before timer expires

Fast Retransmit Algorithm:

```
event: ACK received, with ACK field value of y
    if (y > SendBase) {
        SendBase = y; dupACK = 0;
        fast_retx = false;
        if (SendBase != NextSeqNum)
            start timer;
        else
            cancel timer; // last pkt in window
    }
    else if (y == SendBase && fast_retx == false) {
        dupACK++;
        if (dupACK == 3) {
            fast_retx = true;
            resend segment with sequence number y
        }
    }
```

a duplicate ACK for
already ACKed segment

fast retransmit

Chapter 3: Roadmap

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

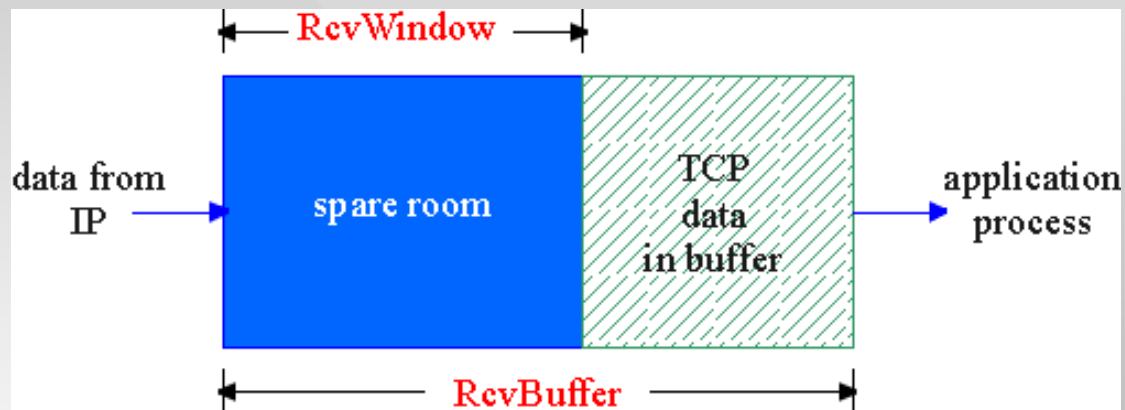
3.7 TCP congestion control

TCP Flow Control

- Receive side of TCP connection has a receive buffer:

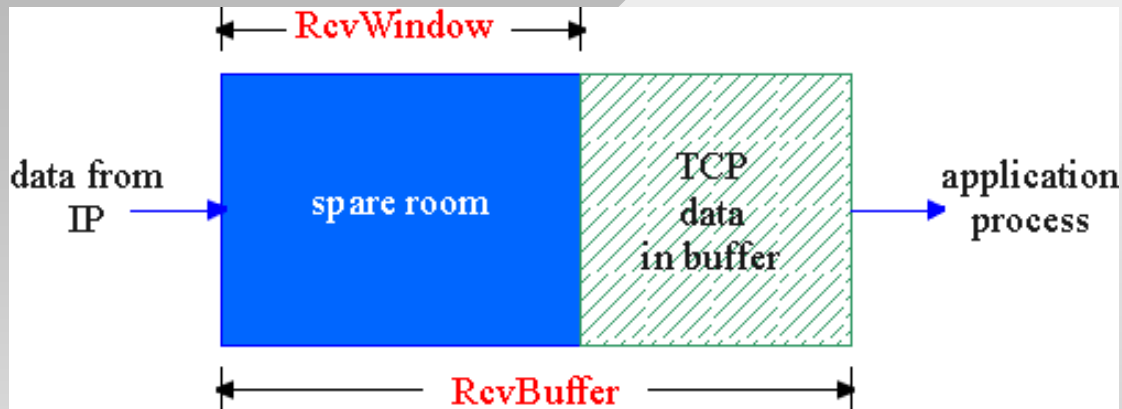
Flow control

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



- App process may be slow at reading from buffer
- Speed-matching service: matching the send rate to the receiving app's drain rate

TCP Flow Control: How It Works



- Spare room in buffer

$$\text{RcvWindow} = \text{RcvBuffer} - [\text{LastByteRcvd} - \text{LastByteRead}]$$

↑
received from sender

↑
read by the application

- Receiver advertises spare room by including value of RcvWindow in segments
- Sender limits unACKed data to RcvWindow
 - Guarantees receive buffer doesn't overflow
- Then the **effective** sender window is $\min(\text{SndWindow}, \text{RcvWindow})$

Chapter 3: Roadmap

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 a connection before exchanging data segments
- Purpose:
 - Exchange initial seq. #s
 - Exchange flow control info (i.e., RcvWindow)
 - Negotiate options (SACK, large windows, etc.)
- *Client*: connection initiator
- *Server*: contacted by client

Three way handshake:

Step 1: client host sends TCP SYN segment to server

- Specifies initial seq # X and buffer size RcvWindow
- No data

Step 2: server receives SYN, replies with SYN+ACK

- Sends server initial seq. # Y and buffer size RcvWindow
- No data

Step 3: client receives SYN+ACK, replies with ACK segment, which may contain regular data

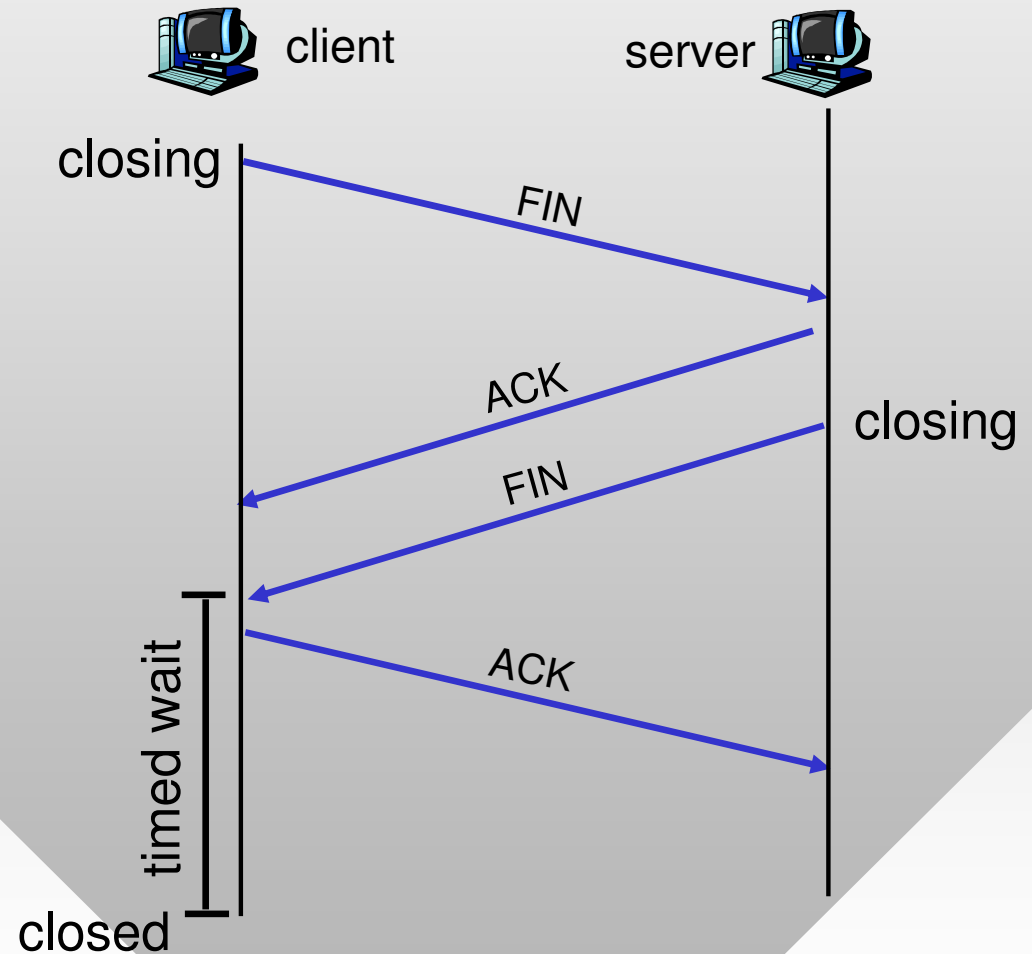
TCP Connection Management (Cont.)

Closing a connection:

- Client closes socket:
`close(sock);`

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

Step 2: server receives FIN, replies with ACK. Connection in “closing” state, sends FIN



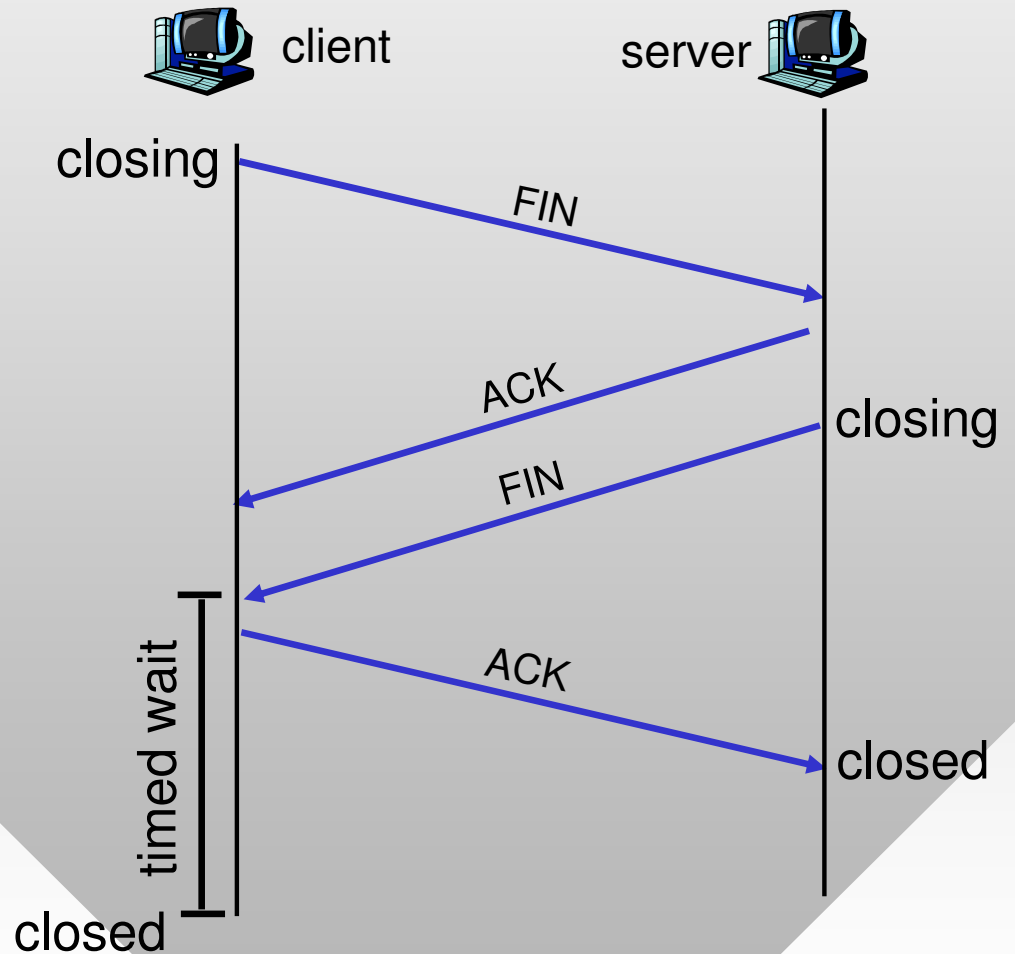
TCP Connection Management (Cont.)

Step 3: client receives FIN, replies with ACK

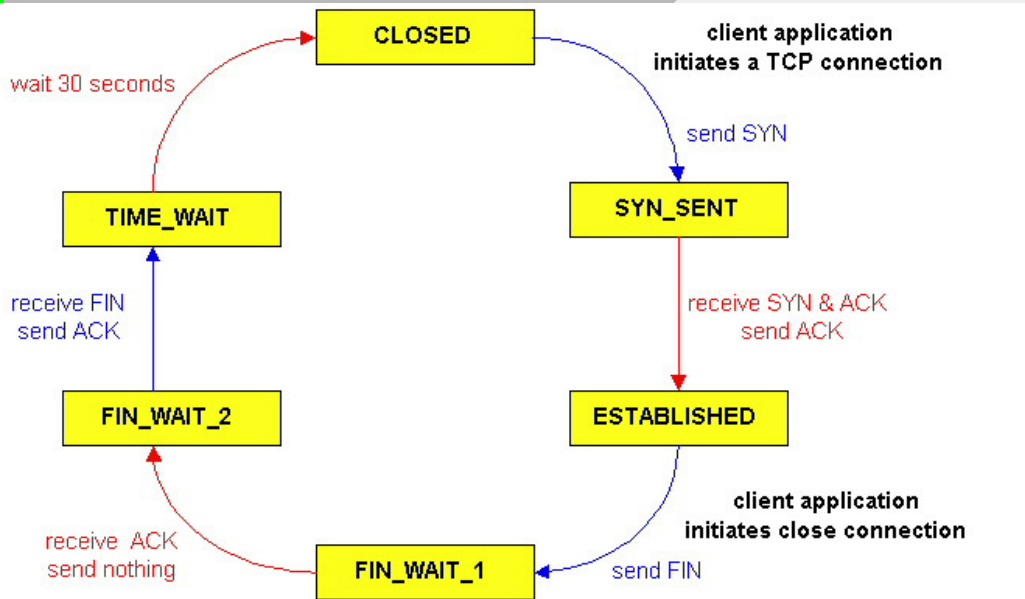
- Enters “timed wait” - will respond with ACK to received FINs

Step 4: server, receives ACK. Connection closed

Step 5: after timeout, client’s connection is closed as well



TCP Connection Management (Cont)



TCP client lifecycle

TCP server lifecycle

