Transport Layer: outline

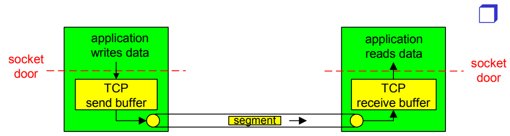
- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
  - Segment structure
  - Reliable data transfer
  - Flow control
  - Connection management
- Principles of congestion control
- TCP congestion control

1

TCP: Overview

RFCs: 793, 1122, 1323, 2018, 2581

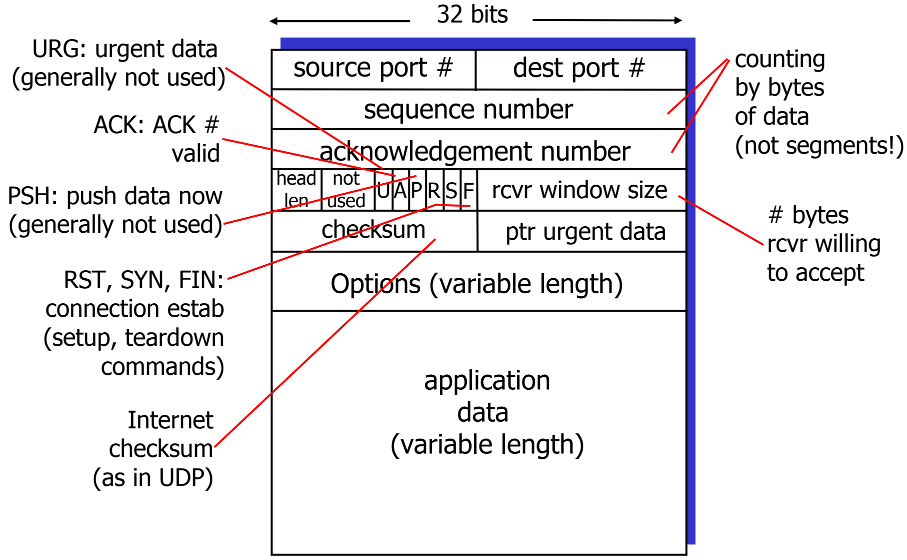
- Point-to-point:
  - One sender, one receiver
- Reliable, in-order *byte stream:*
  - No "message boundaries"
- Pipelined:
  - TCP congestion and flow control set window size
- Send & receive buffers



- 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

2

TCP segment structure



3

Transport Layer: outline

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
  - Segment structure
  - Reliable data transfer
  - Flow control
  - Connection management
- Principles of congestion control
- TCP congestion control

4

TCP reliable data transfer

- TCP creates rdt service on top of IP's unreliable service
- Pipelined segments
- Cumulative acks
- TCP uses single retransmission timer
- Retransmissions are triggered by:
  - Timeout events
  - Duplicate acks
- Initially consider simplified TCP sender:
  - Ignore duplicate acks
  - Ignore flow control, congestion control
  - One way dataflow

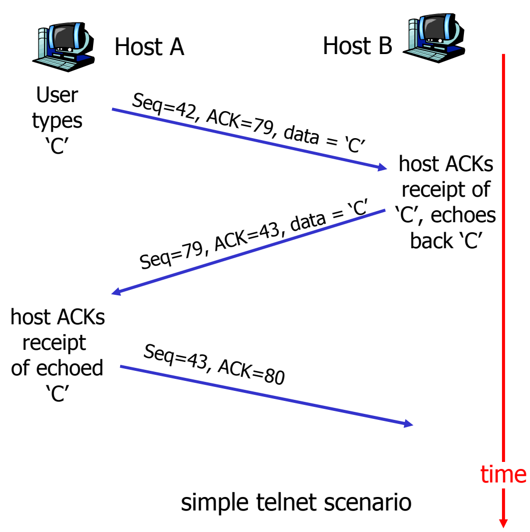
5

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

6

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:

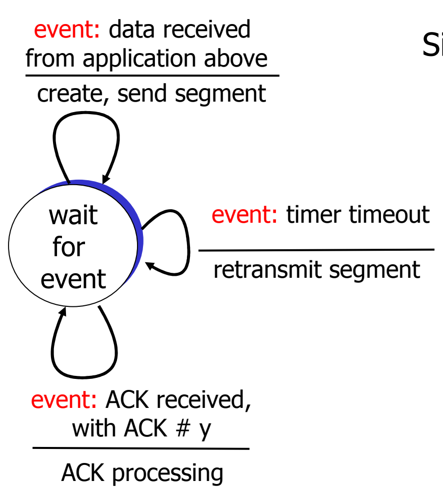
- Retransmit segment that caused timeout
- Restart timer

Ack rcvd:

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

7

TCP: reliable data transfer



Simplified sender, assuming

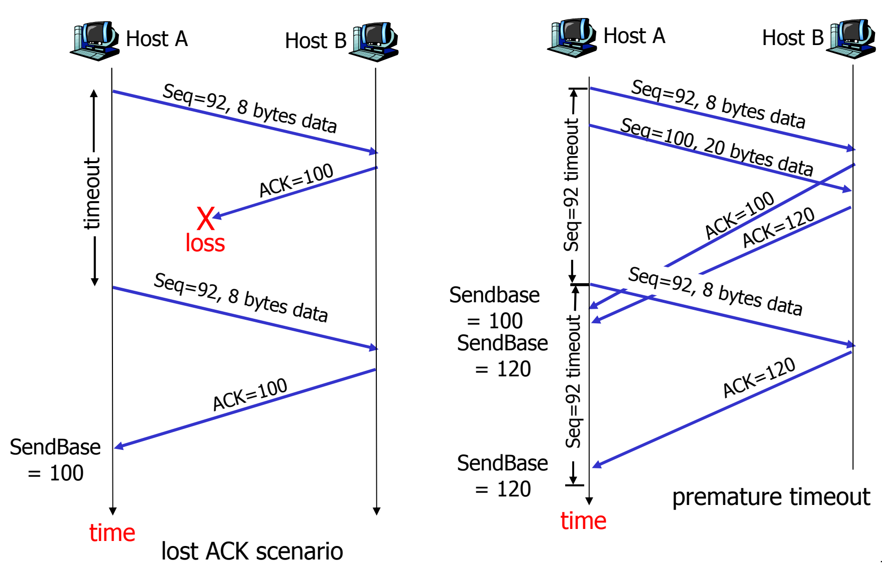
- One way data transfer
- No flow, congestion control

8

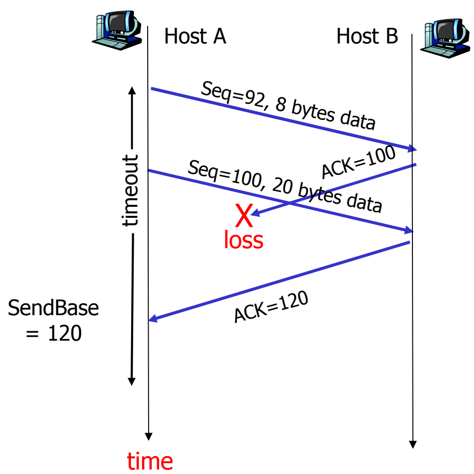
TCP sender (simplified)

```
sendbase = initial_sequence number
nextseqnum = initial_sequence number

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
      restart timer
   event: ACK received, with ACK field value of y
      if (y > sendbase) { /* cumulative ACK of all data up to y */
          sendbase = y
          if (currently not-yet-acknowledged segments) {
             restart timer
          }
      }
 }  /* end of loop forever */
```


TCP retransmission scenarios



TCP retransmission scenarios (2.)



Cumulative ACK scenario

11

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`**

12

TCP round trip time and timeout

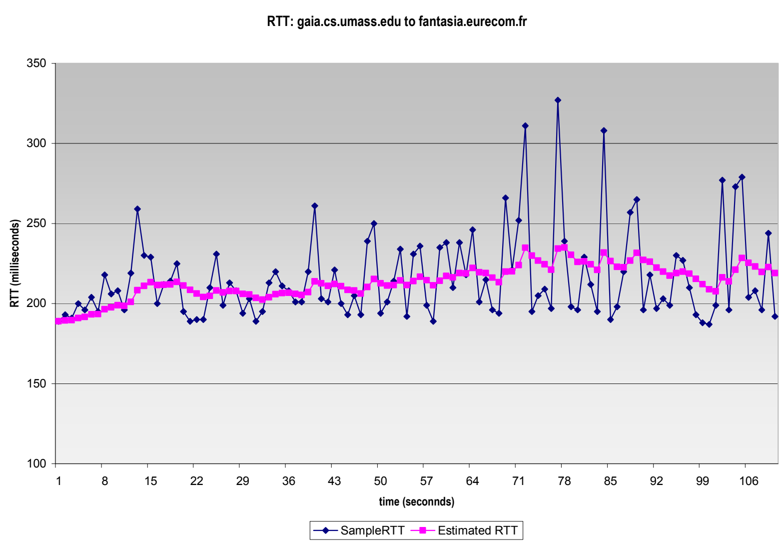
```
EstimatedRTT = (1 - α)*EstimatedRTT + α * SampleRTT
```

- Exponential weighted moving average
- Influence of given sample decreases exponentially fast
- Typical value of α: 0.125

- Key observation:
  - At high loads round trip variance is high

13

Example RTT estimation



14

TCP round trip time and timeout

Setting the timeout

- **`EstimtedRTT`** plus "safety margin"
  - Large variation in **`EstimatedRTT`** -> larger safety margin
- First estimate of how much SampleRTT deviates from EstimatedRTT:

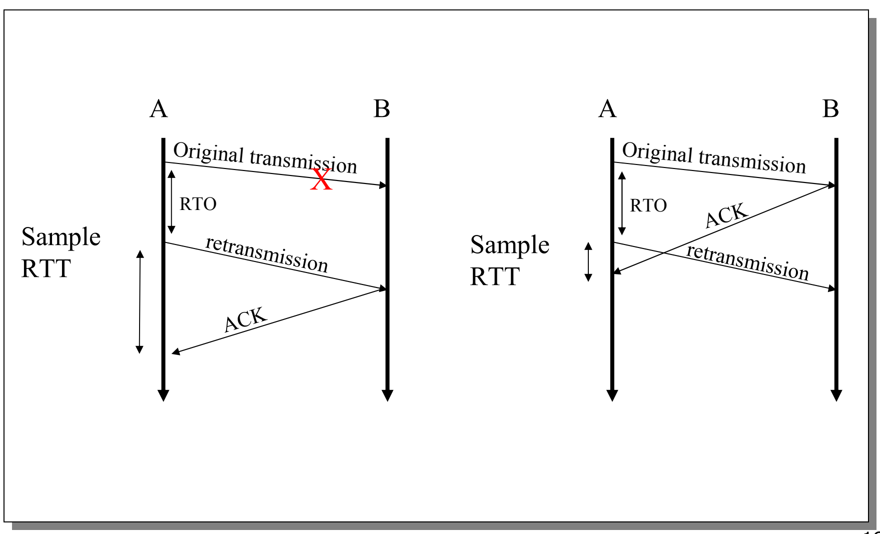
```
DevRTT = (1-β)*DevRTT +
             β*|SampleRTT-EstimatedRTT|

(typically, β = 0.25)
```

Then set timeout interval:

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

15

Retransmission Ambiguity



16

Karn's RTT Estimator

- Accounts for retransmission ambiguity
- If a segment has been retransmitted:
  - Don't count RTT sample on ACKs for this segment
  - Keep backed off time-out for next packet
  - Reuse RTT estimate only after one successful transmission

17

Timestamp Extension

- Used to improve timeout mechanism by more accurate measurement of RTT
- When sending a packet, insert current timestamp into option
  - 4 bytes for seconds, 4 bytes for microseconds
- Receiver echoes timestamp in ACK
  - Actually will echo whatever is in timestamp
- Removes retransmission ambiguity
  - Can get RTT sample on any packet

18

Timer Granularity

- Many TCP implementations set RTO in multiples of 200,500,1000ms
- Why?
  - Avoid spurious timeouts – RTTs can vary quickly due to cross traffic
  - Make timers interrupts efficient

19

TCP ACK generation [RFC 1122, RFC 2581]

Event at Receiver	TCP Receiver action
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
In-order segment with expected seq #. One other segment has ACK pending	Immediately send single cumulative ACK, ACKing both in-order segments
Out-of-order segment higher-than-expect seq. # . Gap detected	Immediately send *duplicate ACK*, indicating seq. # of next expected byte
Segment that partially or completely fills gap	Immediate send ACK, provided that segment starts at lower end of gap

20

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

21

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
               }
```

Duplicate ACK for already ACKed segment

Fast retransmit

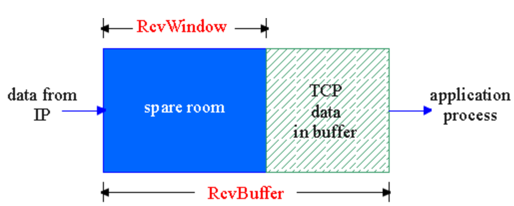
22

Transport layer: outline

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
  - Segment structure
  - Reliable data transfer
  - Flow control
  - Connection management
- Principles of congestion control
- TCP congestion control

TCP flow control

- Receive side of TCP connection has a receive buffer:



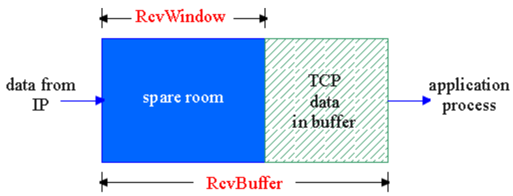
- App process may be slow at reading from buffer

flow control

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

- Speed-matching service: match 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

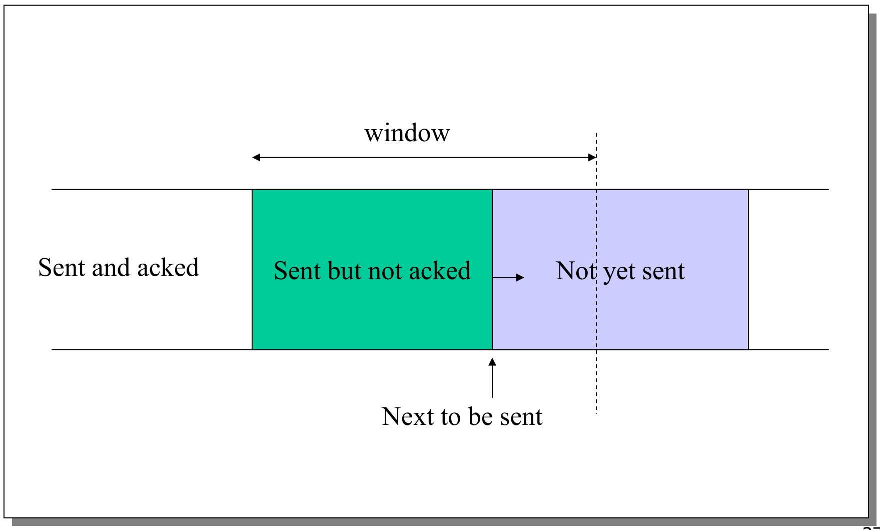
25

TCP flow control: How it works

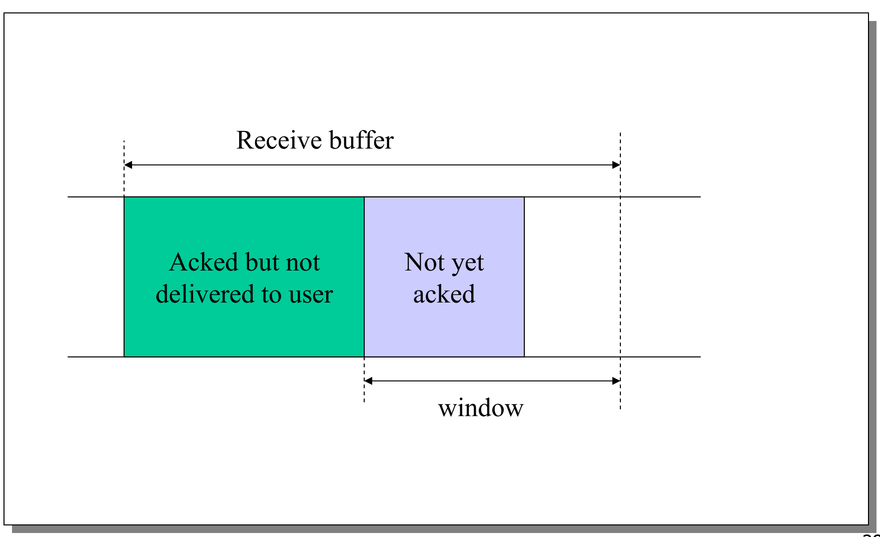
- TCP is a sliding window protocol
  - For window size *n*, can send up to *n* bytes without receiving an acknowledgement
  - When the data is acknowledged then the window slides forward
- Each packet advertises a window size
  - Indicates number of bytes the receiver has space for
- Original TCP always sent entire window
  - Congestion control now limits this

26

Window flow control: Send Side



Window flow control: Receive Side



TCP persist

- What happens if window is 0?
  - Receiver updates window when application reads data
  - What if this update is lost?
- TCP Persist state
  - Sender periodically sends 1 byte packets
  - Receiver responds with ACK even if it can't store the packet

29

Transport layer: outline

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
  - Segment structure
  - Reliable data transfer
  - Flow control
  - Connection management
- Principles of congestion control
- TCP congestion control

30

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");
```

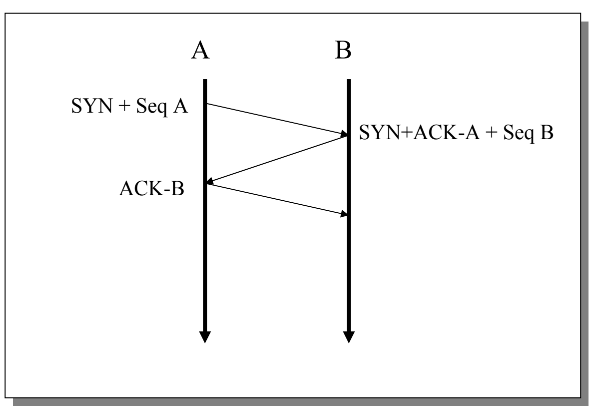
- *server:* contacted by client

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

31

Connection establishment

- Use 3-way handshake



32

Sequence number selection

- Why not simply chose 0?
- Must avoid overlap with earlier incarnation

33

TCP connection: Three way handshake

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

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

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

34

TCP connection management (2.)

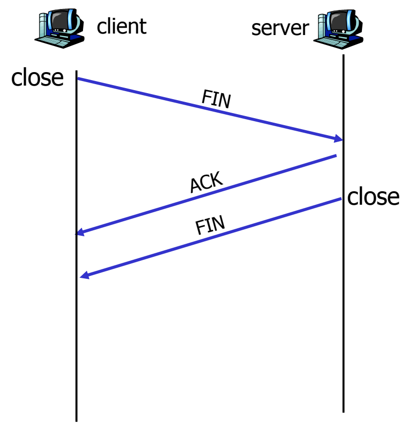
Closing a connection:

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.



35

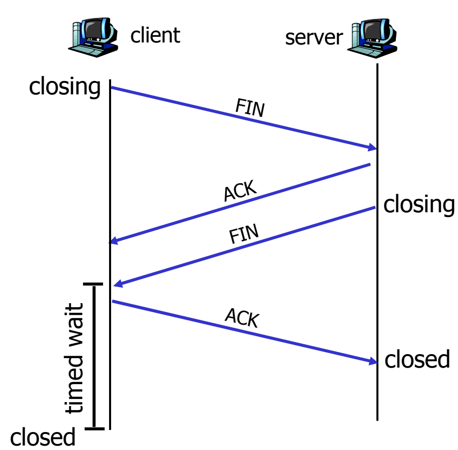
TCP connection management (3.)

Step 3: client receives FIN, replies with ACK.

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

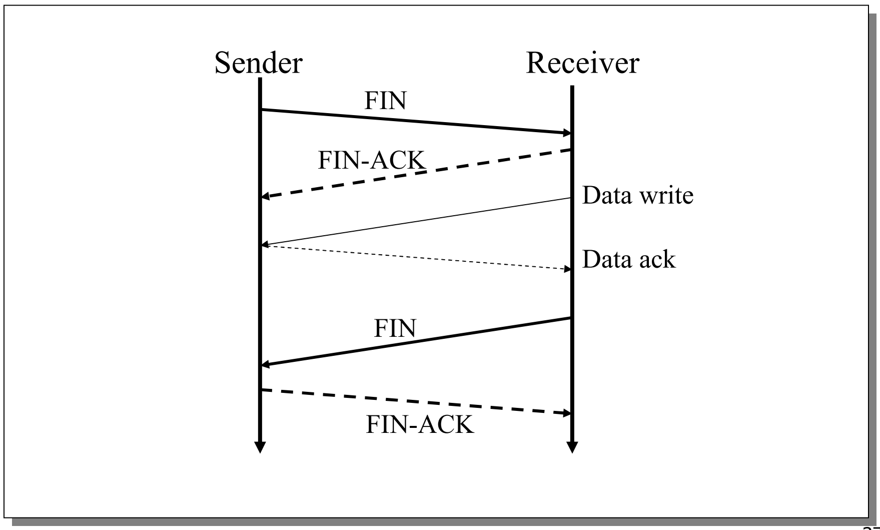
Step 4: server, receives ACK. Connection closed.

Note: with small modification, can handle simultaneous FINs.



36

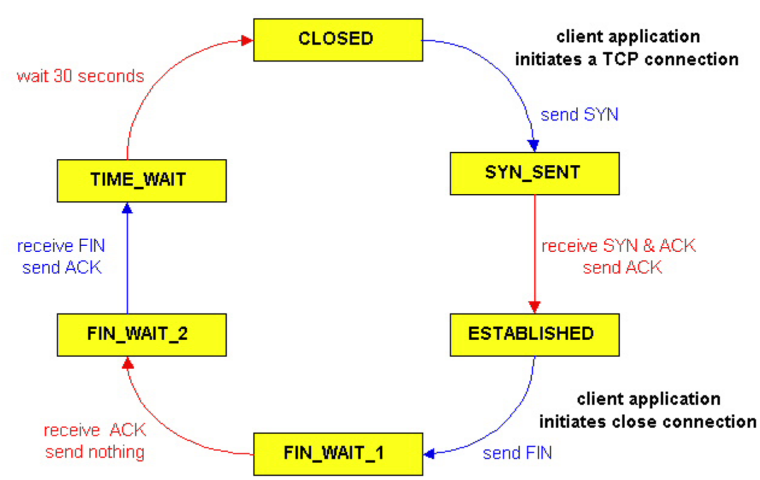
Tear-down packet exchange



37

TCP connection management (cont.)

TCP client lifecycle



38

TCP connection management (cont.)

TCP server lifecycle