

## Transport Layer: Outline

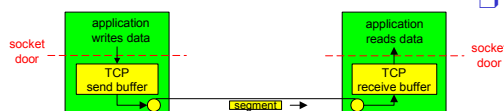
- ❑ Transport-layer services
- ❑ Multiplexing and demultiplexing
- ❑ Connectionless transport: UDP
- ❑ Principles of reliable data transfer
- ❑ **Connection-oriented transport: TCP**
  - Segment structure
  - Reliable data transfer
  - Flow control
  - Connection management
- ❑ Principles of congestion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

1

## TCP: Overview

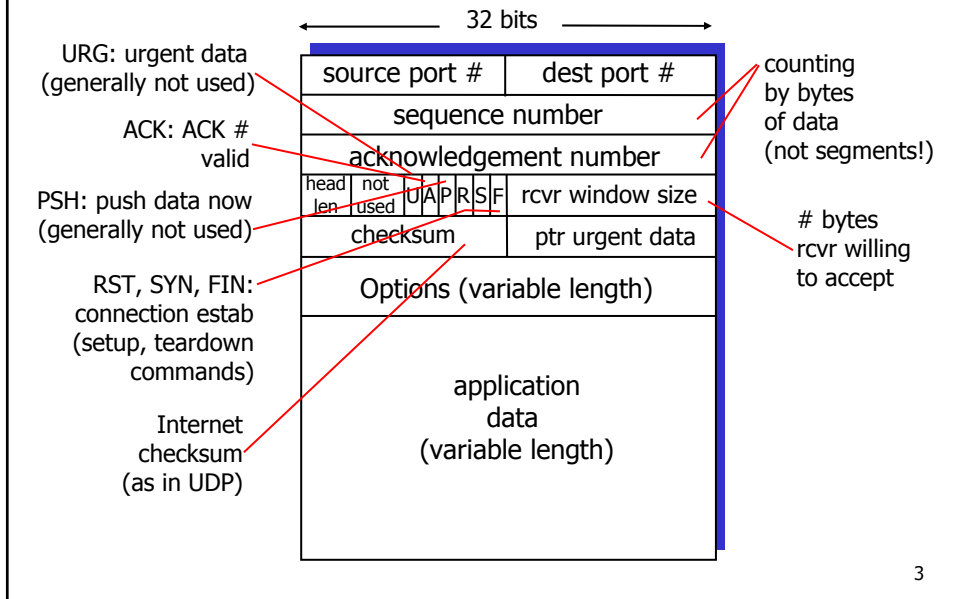
RFCs: 793, 1122, 1323, 2018, 2581

- ❑ **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



## 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 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

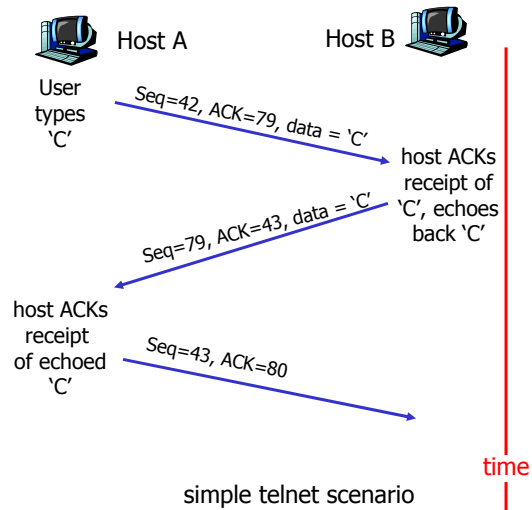
### Seq. #'s:

- 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



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

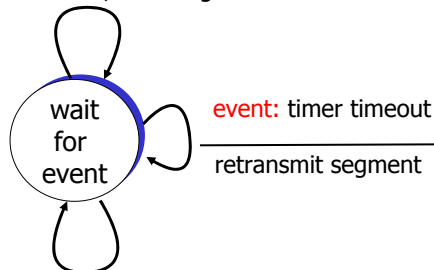
### Ack rcvd:

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

7

## TCP: reliable data transfer

event: data received from application above  
create, send segment



event: ACK received, with ACK # y  
ACK processing

Simplified sender, assuming

- One way data transfer
- No flow, congestion control

8

## TCP sender (simplified)

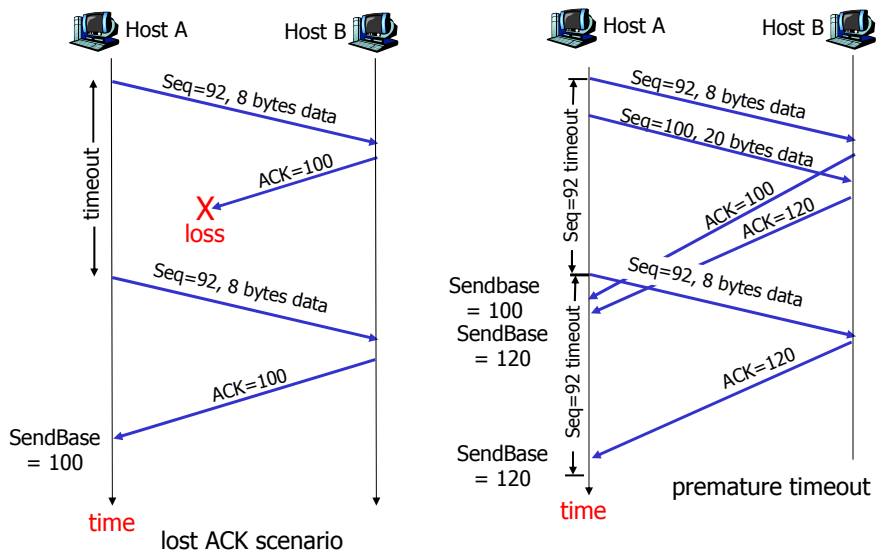
```

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       if (timer currently not running) start timer
08       pass segment to IP
09       nextseqnum = nextseqnum + length(data)
10     event: timer timeout
11       retransmit not-yet-acknowledged segment with
12         smallest sequence number
13       restart timer
14     event: ACK received, with ACK field value of y
15       if (y > sendbase) { /* cumulative ACK of all data up to y */
16         sendbase = y
17         if (currently not-yet-acknowledged segments) {
18           restart timer
19         }
20       }
21     }
22   } /* end of loop forever */

```

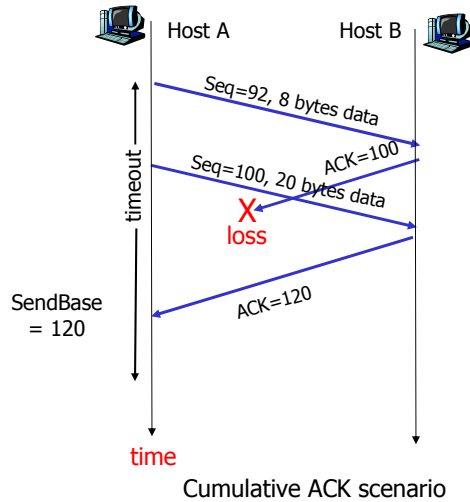
9

## TCP retransmission scenarios



10

## TCP retransmission scenarios (2.)



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

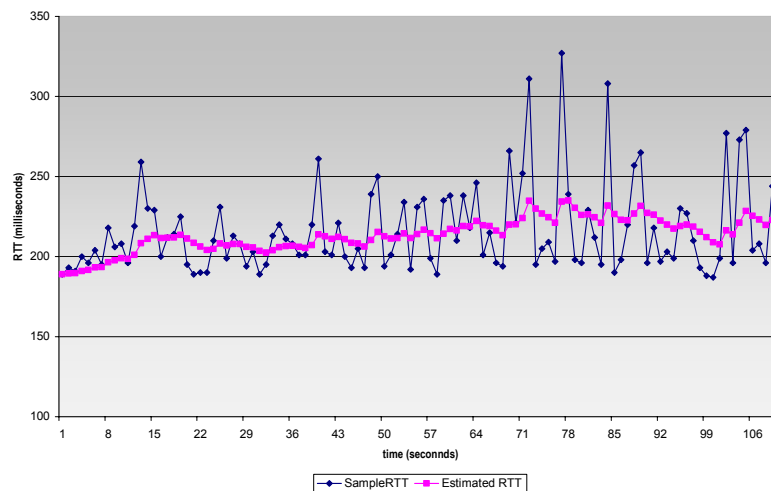
$$\text{EstimatedRTT} = (1 - \alpha) * \text{EstimatedRTT} + \alpha * \text{SampleRTT}$$

- Exponential weighted moving average
- Influence of given sample decreases exponentially fast
- Typical value of  $\alpha$ : 0.125
  
- Key observation:
  - At high loads round trip variance is high

13

## Example RTT estimation

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



14

## 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**:

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

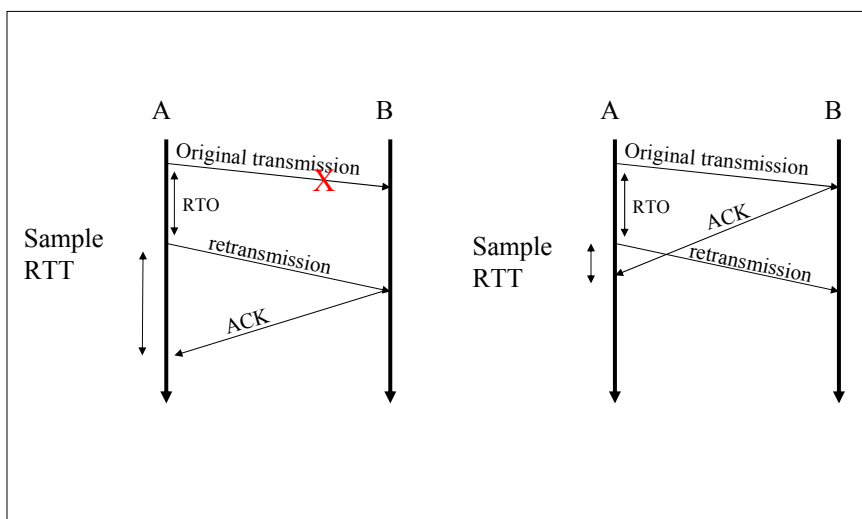
(typically,  $\beta = 0.25$ )

Then set timeout interval:

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

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]

<u>Event at Receiver</u>	<u>TCP Receiver action</u>
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-expected seq. # . Gap detected	Immediately send <i>duplicate ACK</i> , 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

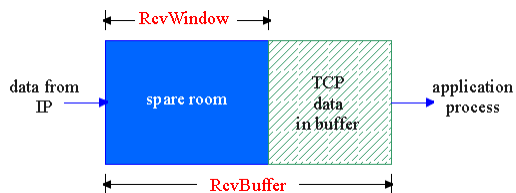
## Transport layer: outline

- ❑ 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

23

## TCP flow control

- ❑ Receive side of TCP connection has a receive buffer:



- ❑ App process may be slow at reading from buffer

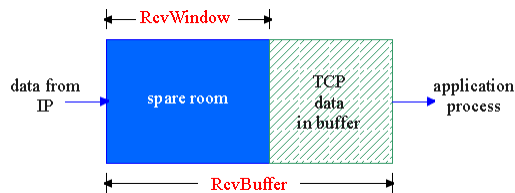
### **flow control**

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

24

## 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

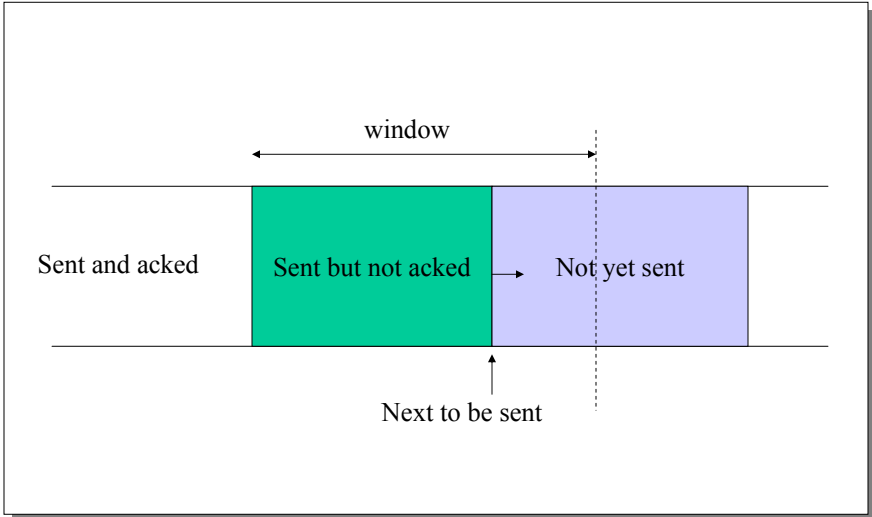
25

## TCP flow control: How it works (2.)

- 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

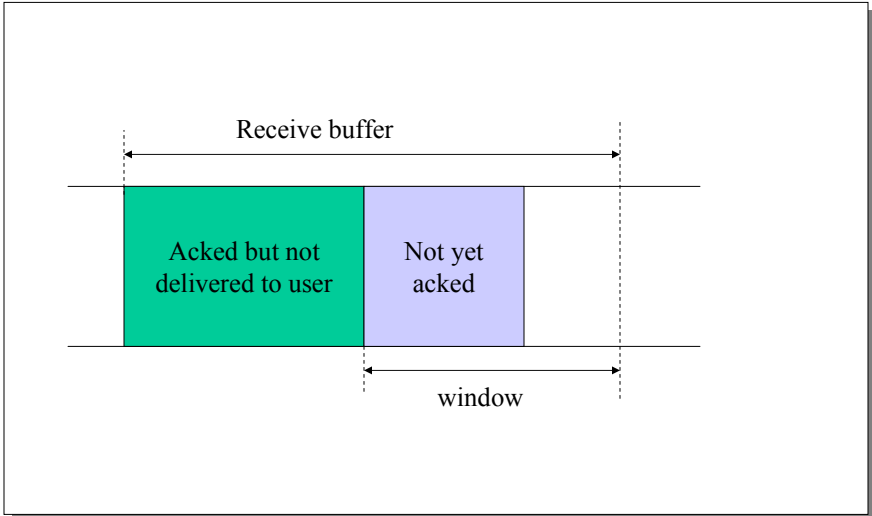
26

# Window flow control: Sender side



27

# Window flow control: Receiver side



28

## 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");
```

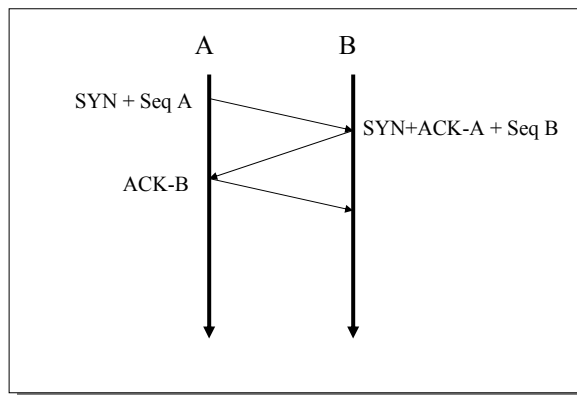
□ *server*: contacted by client

```
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.)

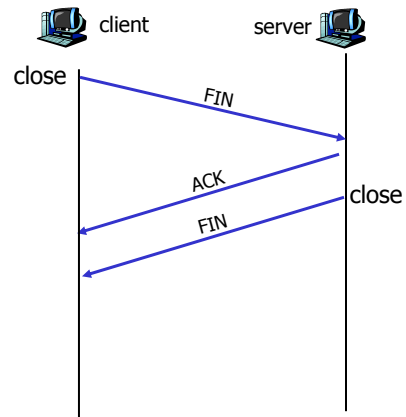
### Closing a connection:

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

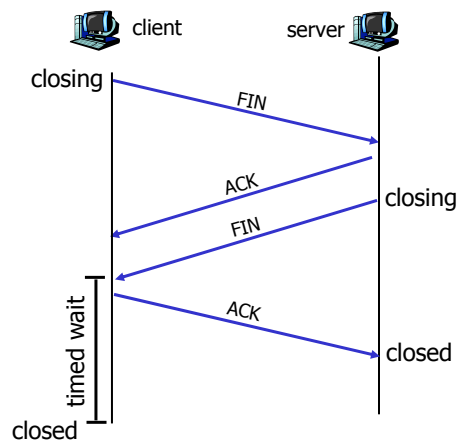
## TCP connection management (3.)

**Step 3:** Client receives FIN, replies with ACK.

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

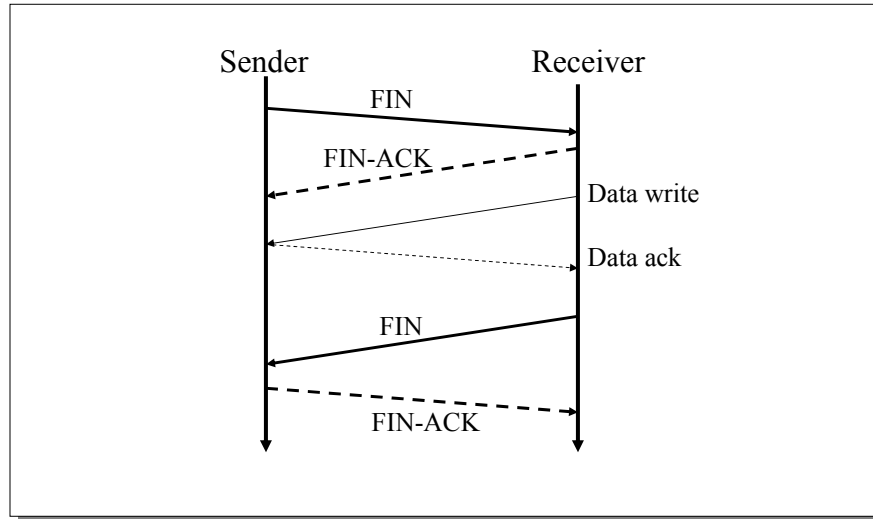
**Step 4:** Server, receives ACK. Connection closed.

**Note:** With small modification, can handle simultaneous FINs.



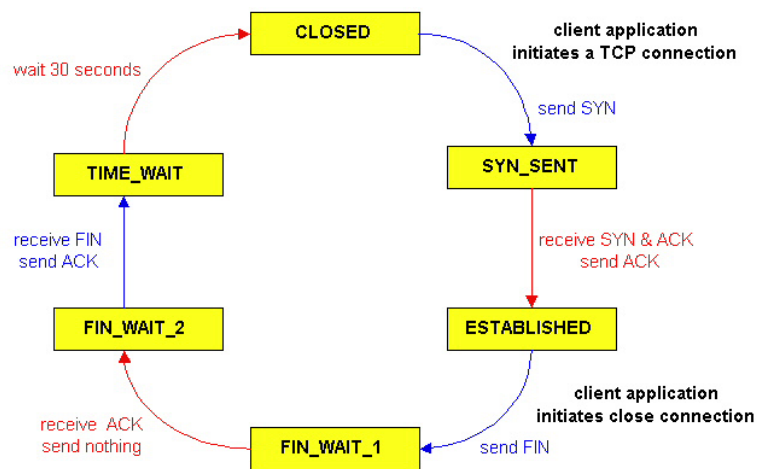
36

## Tear-down packet exchange



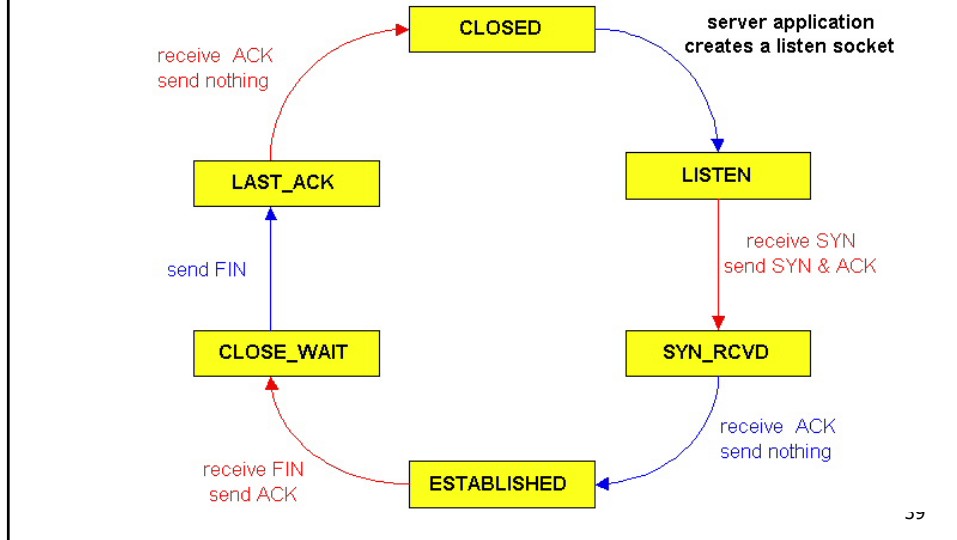
## TCP connection management (cont.)

### TCP client lifecycle



## TCP connection management (cont.)

### TCP server lifecycle



## Detecting half-open connections

TCP A

TCP B

- |                                  |                               |                         |
|----------------------------------|-------------------------------|-------------------------|
| 1. (CRASH)                       |                               | (send 300, receive 100) |
| 2. CLOSED                        |                               | ESTABLISHED             |
| 3. SYN-SENT → <SEQ=400><CTL=SYN> | →                             | (??)                    |
| 4. (!!)                          | ← <SEQ=300><ACK=100><CTL=ACK> | ← ESTABLISHED           |
| 5. SYN-SENT → <SEQ=100><CTL=RST> | →                             | (Abort!!)               |
| 6. SYN-SENT                      |                               | CLOSED                  |
| 7. SYN-SENT → <SEQ=400><CTL=SYN> | →                             |                         |

## Observed TCP problems

- ❑ Too many small packets
  - Silly window syndrome
  - Nagel's algorithm
- ❑ Initial sequence number selection
- ❑ Amount of state maintained

41

## Silly window syndrome

- ❑ Problem: (Clark, 1982)
  - If receiver advertises small increases in the receive window then the sender may waste time sending lots of small packets
- ❑ Solution
  - Receiver must not advertise small window increases
  - Increase window by  $\min(\text{MSS}, \text{RecvBuffer}/2)$

42

## Nagel's algorithm

- ❑ Small packet problem:
  - Don't want to send a 41 byte packet for each keystroke
  - How long to wait for more data?
- ❑ Solution:
  - Allow only one outstanding small (not full sized) segment that has not yet been acknowledged

43

## Why is selecting ISN important?

- ❑ Suppose machine X selects ISN based on predictable sequence
- ❑ Fred has .rhosts to allow login to X from Y
- ❑ Evil Ed attacks
  - Disables host Y – denial of service attack
  - Make a bunch of connections to host X
  - Determine ISN pattern and guess next ISN
  - Fake pkt1: [<src Y> <dst X>, guessed ISN]
  - Fake pkt2: desired command

44

## Time Wait issues

- ❑ Web servers not clients close connection first
  - Established → Fin-Waits → Time-Wait → Closed
  - Why would this be a problem?
- ❑ Time-Wait state lasts for  $2 * \text{MSL}$ 
  - MSL is should be 120 seconds (is often 60s)
  - Servers often have order of magnitude more connections in Time-Wait

45

## 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

46