

# Computer Networks

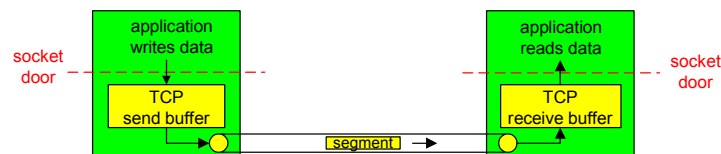
## Connection-Oriented Transport: TCP

Based on Computer Networking, 4<sup>th</sup> Edition by Kurose and Ross

Stan Kurkovsky

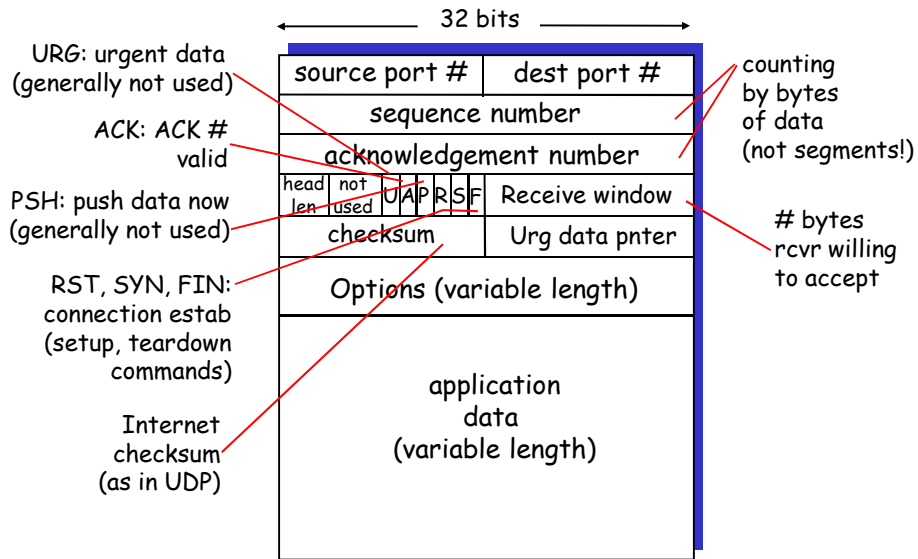
### TCP: Overview

- **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



Stan Kurkovsky

## TCP segment structure



Stan Kurkovsky

## TCP seq. #'s and ACKs

### Seq. #'s:

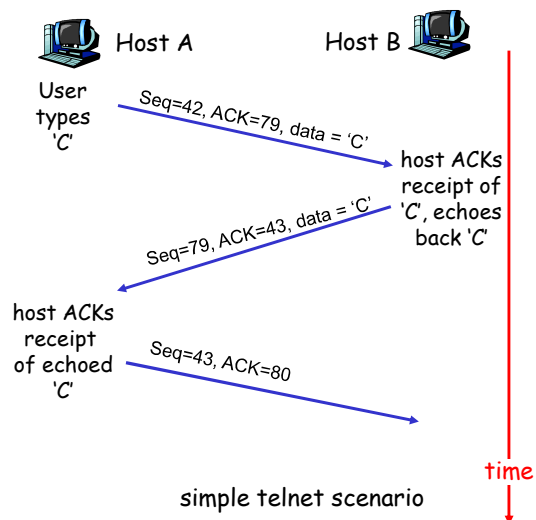
- byte stream "number" of first byte in segment's data

### 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

Stan Kurkovsky

## 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

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**  
$$\text{EstimatedRTT} = (1 - \alpha) * \text{EstimatedRTT} + \alpha * \text{SampleRTT}$$
  - Exponential weighted moving average
  - influence of past sample decreases exponentially fast
  - typical value:  $\alpha = 0.125$

Stan Kurkovsky

## 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:

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

Stan Kurkovsky

## 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

Stan Kurkovsky

## 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:

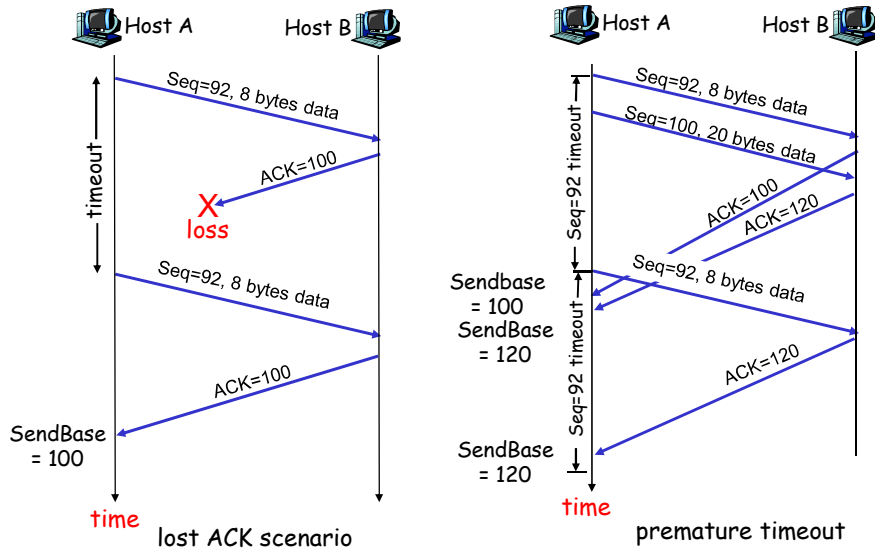
- retransmit segment that caused timeout
- restart timer

### Ack rcvd:

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

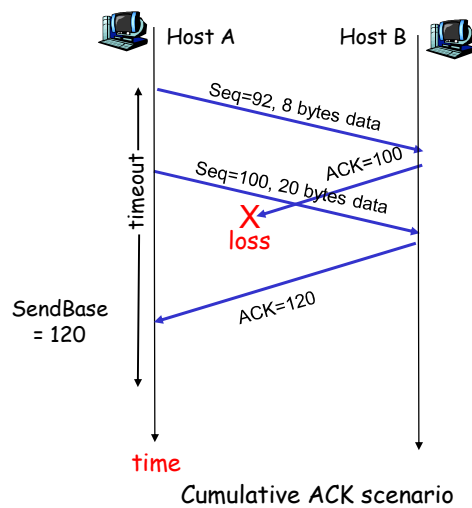
Stan Kurkovsky

## TCP retransmission scenarios



Stan Kurkovsky

## TCP retransmission scenarios



Stan Kurkovsky

## TCP ACK generation

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 <i>duplicate ACK</i> , 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

Stan Kurkovsky

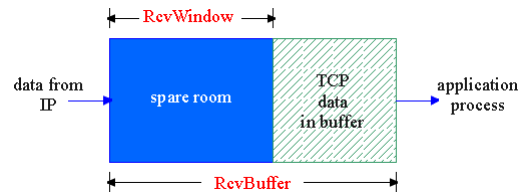
## Fast Retransmit

- 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

Stan Kurkovsky

## TCP Flow Control

- sender won't overflow receiver's buffer by transmitting too much, too fast
- receive side of TCP connection has a receive buffer:



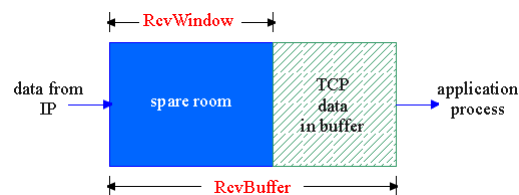
- app process may be slow at reading from buffer
- speed-matching service: matching the send rate to the receiving app's drain rate

Stan Kurkovsky

## 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



Stan Kurkovsky

## 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

Stan Kurkovsky

## TCP Connection Management (cont.)

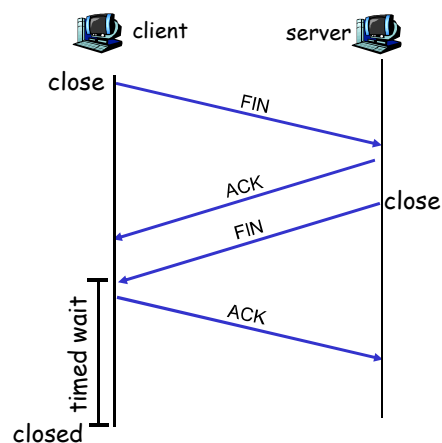
### Closing a connection:

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.



Stan Kurkovsky

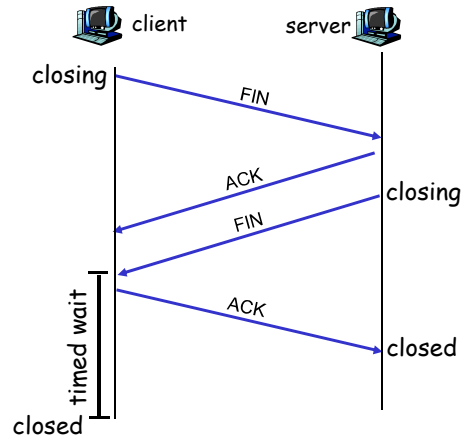
## TCP Connection Management (cont.)

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

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

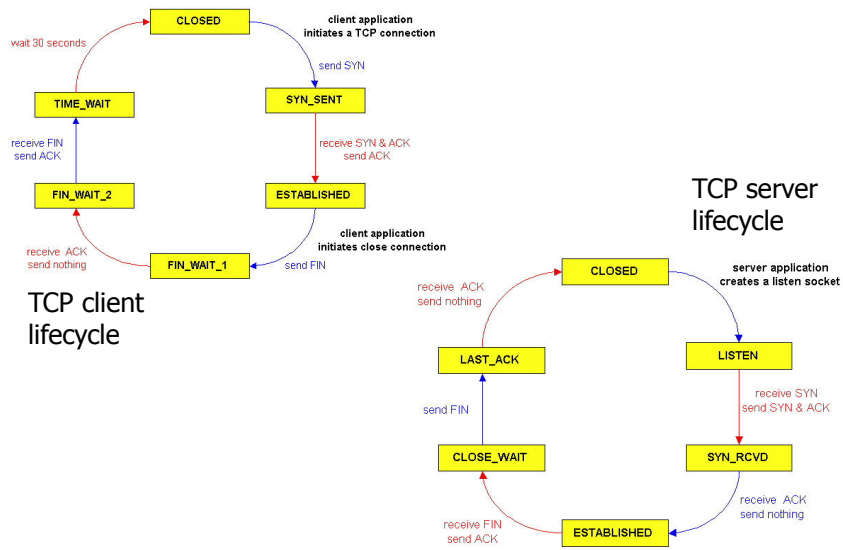
**Step 4:** server, receives ACK.  
Connection closed.

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



Stan Kurkovsky

## TCP Connection Management (cont)



Stan Kurkovsky