

# Transport layer

- Review principles:
  - Reliable data transfer
  - Flow control
  - Congestion control
- Instantiation in the Internet
  - UDP
  - TCP

# UDP: User Datagram Protocol [RFC 768]

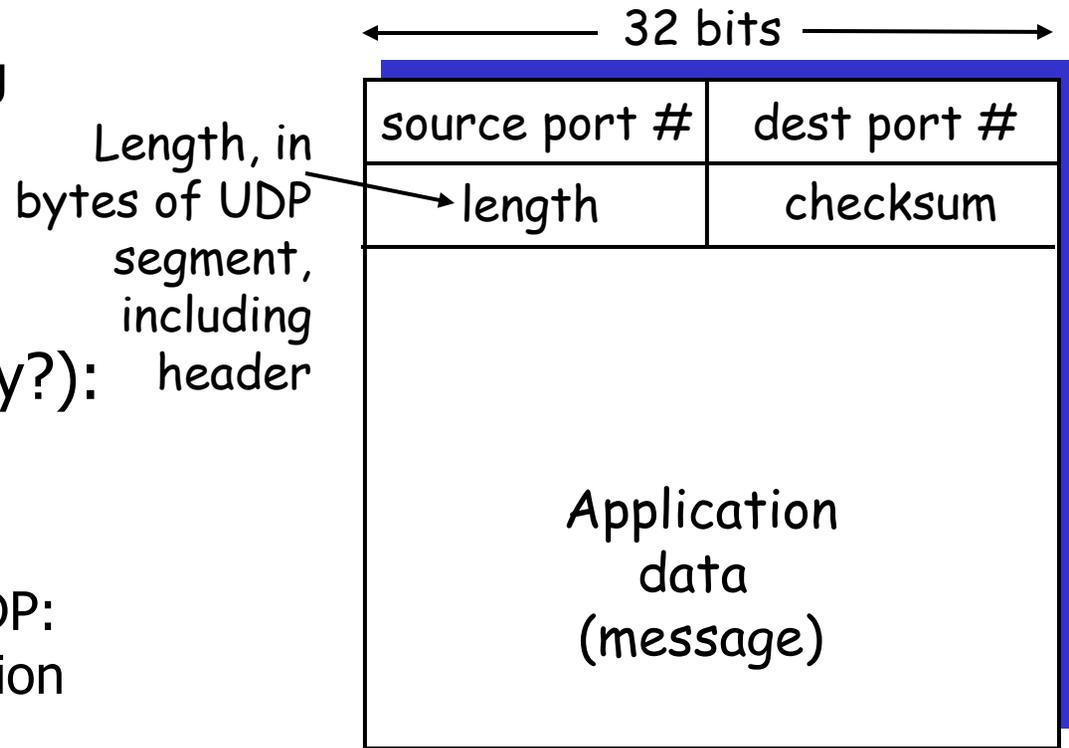
- ❑ “No frills,” “bare bones” Internet transport protocol
- ❑ “Best effort” service, UDP segments may be:
  - Lost
  - Delivered out of order to application
- ❑ *Connectionless:*
  - No handshaking between UDP sender, receiver
  - Each UDP segment handled independently of others

## Why is there a UDP?

- ❑ No connection establishment (which can add delay)
- ❑ Simple: no connection state at sender, receiver
- ❑ Small segment header
- ❑ No congestion control: UDP can blast away as fast as desired

# UDP: More

- ❑ Often used for streaming multimedia apps
  - Loss tolerant
  - Rate sensitive
- ❑ Other UDP uses (why?):
  - DNS
  - SNMP
- ❑ Reliable transfer over UDP: add reliability at application layer
  - Application-specific error recover!



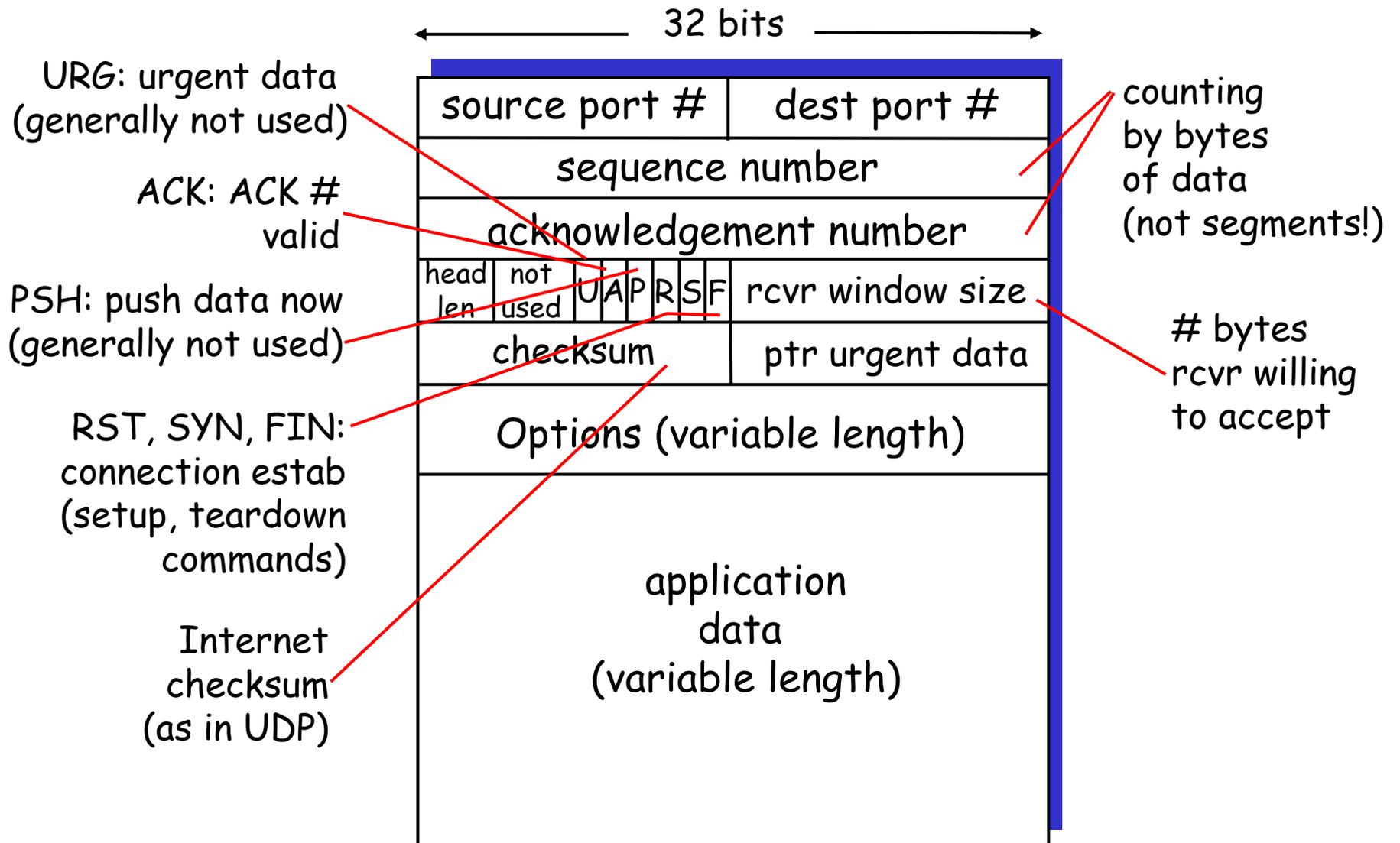
UDP segment format

# 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
- ❑ **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

# TCP segment structure



# TCP seq. #'s and ACKs

## Seq. #'s:

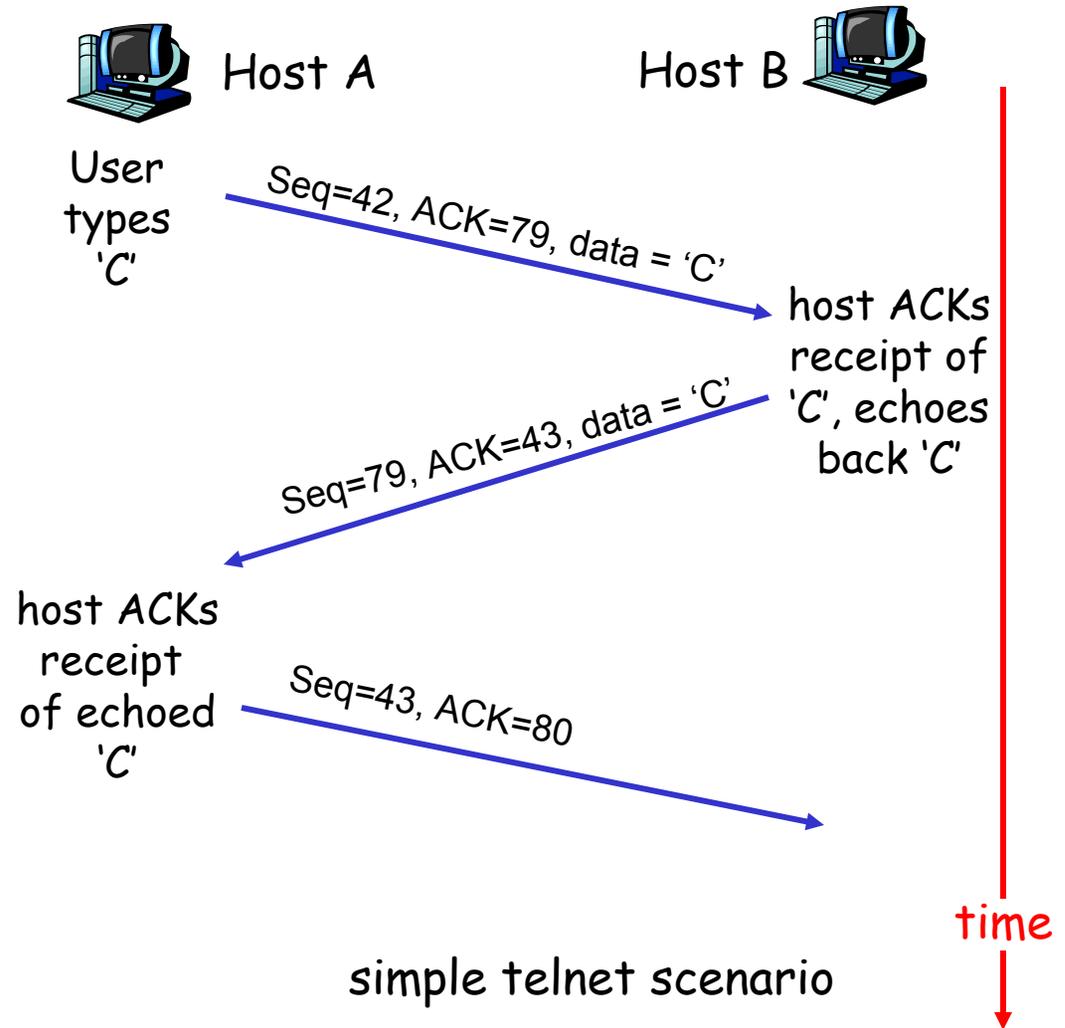
- Byte stream "number" of first byte in segment's data

## ACKs:

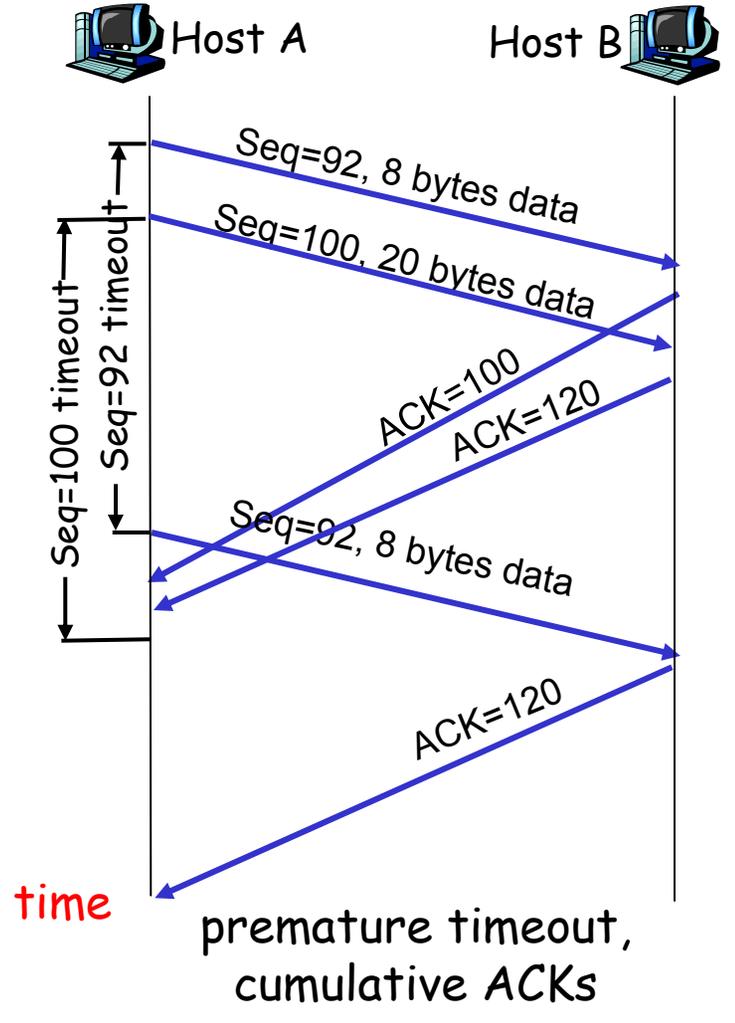
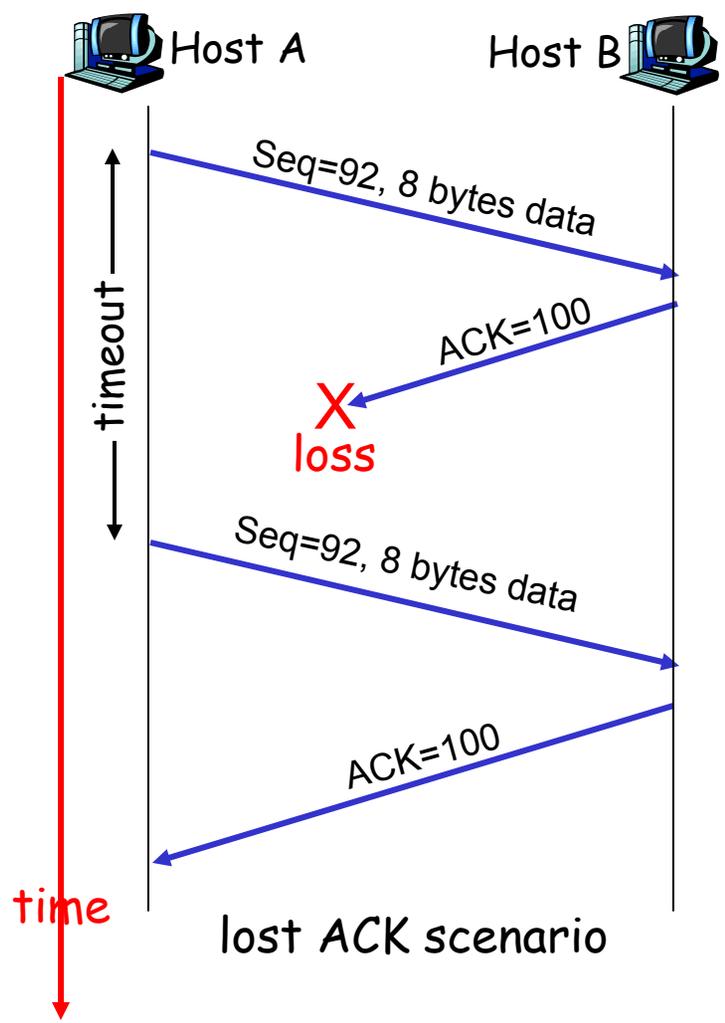
- Seq # of next byte expected from other side
- Cumulative ACK

**Q:** How receiver handles out-of-order segments

- A: TCP spec doesn't say – up to implementor



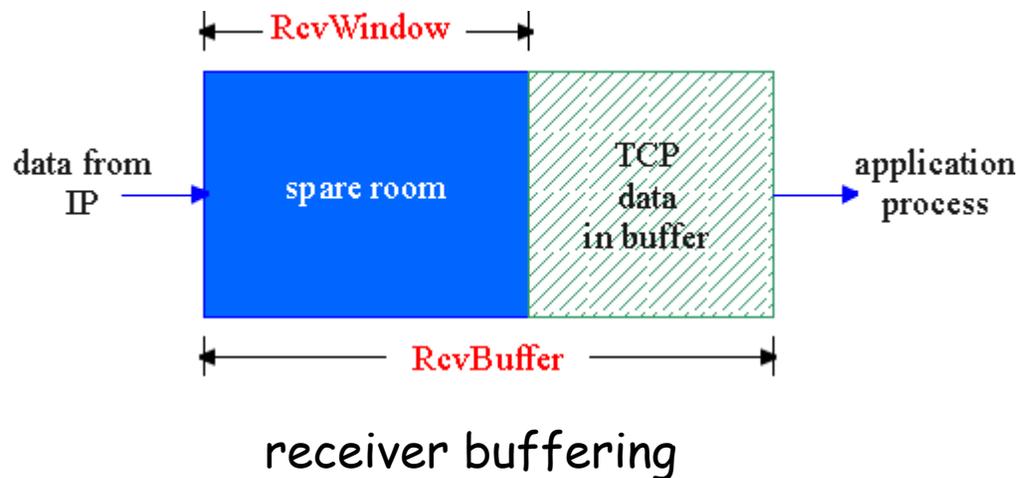
# TCP: Retransmission scenarios



# TCP Flow Control

## flow control

sender won't overrun receiver's buffers by transmitting too much, too fast



**Receiver:** explicitly informs sender of (dynamically changing) amount of free buffer space

- **rcvr window size** field in TCP segment

**Sender:** amount of transmitted, unACKed data less than most recently-receiver **rcvr window size**

# TCP Round-trip Time (RTT) and Timeout estimation

- ❑ Wait at least one RTT before retransmitting
- ❑ Importance of accurate RTT estimators:
  - Low RTT → unneeded retransmissions
  - High RTT → poor throughput
- ❑ RTT estimator must adapt to change in RTT
  - But not too fast, or too slow!
- ❑ Spurious timeouts
  - “Conservation of packets” principle – more than a window worth of packets in flight

# 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
- ❑ *Server:* contacted by client

# TCP connection management (2)

## 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

# TCP connection management (3)

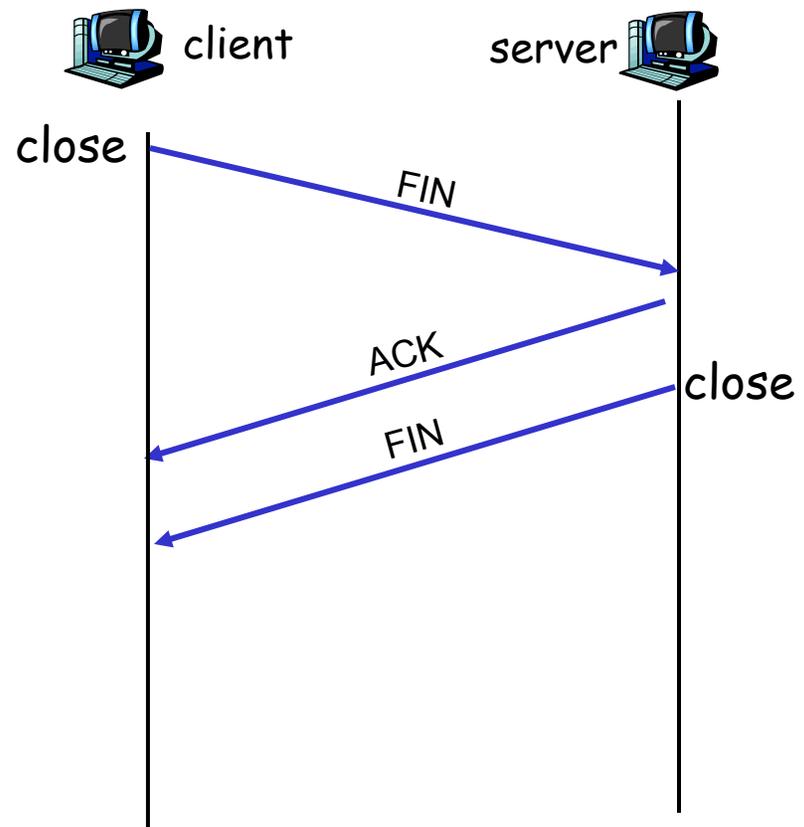
## Closing a connection:

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.



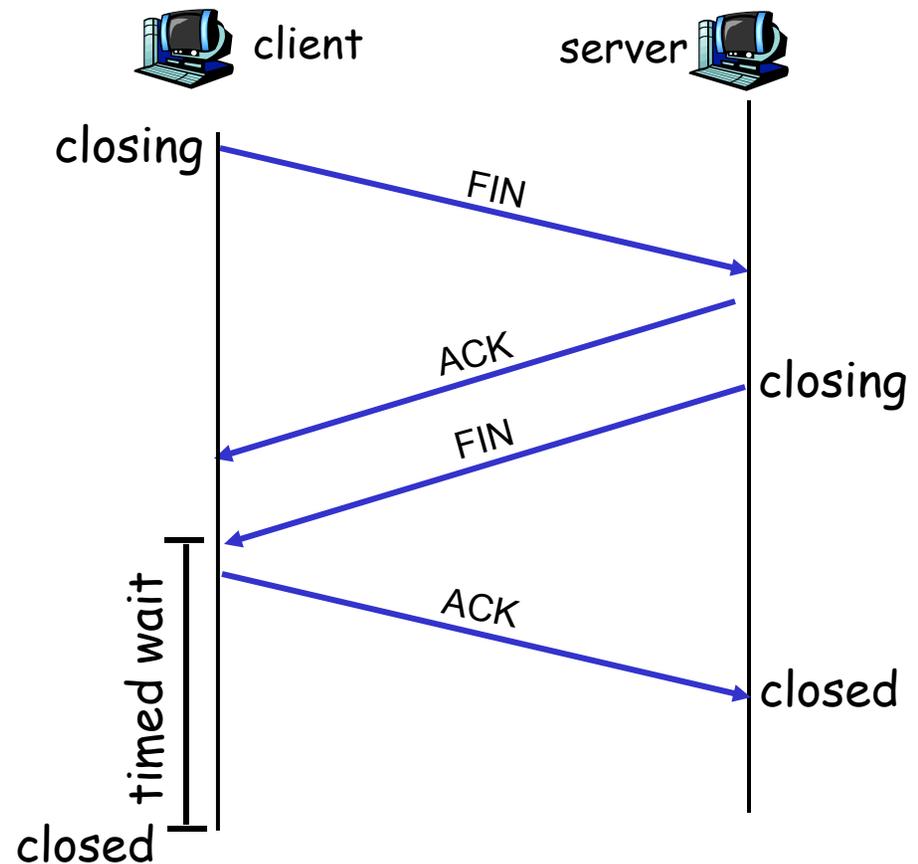
# TCP connection management (4)

**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, this can handle simultaneous FINs.

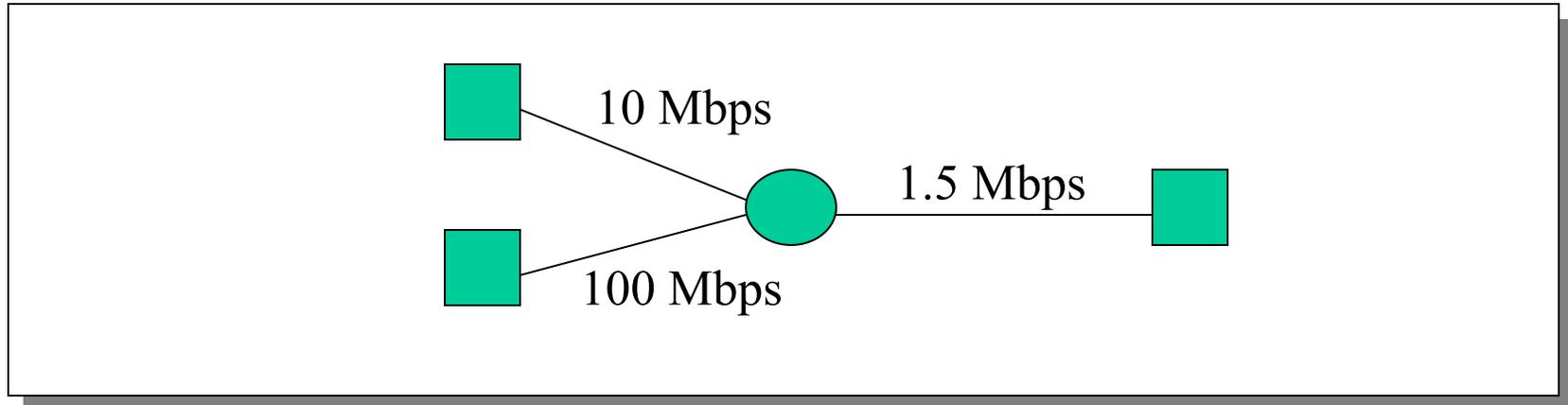


# Principles of congestion control

## Congestion:

- ❑ Informally: “too many sources sending too much data too fast for *network* to handle”
- ❑ Different from flow control!
- ❑ Manifestations:
  - Lost packets (buffer overflow at routers)
  - Long delays (queueing in router buffers)

# Congestion



- ❑ Different sources compete for resources inside network
- ❑ Why is it a problem?
  - Sources are unaware of current state of resource
  - Sources are unaware of each other
  - In many situations will result in  $< 1.5$  Mbps of throughput (congestion collapse)

# Congestion collapse

- ❑ Definition: *Increase in network load results in decrease of useful work done*
- ❑ Many possible causes
  - Spurious retransmissions of packets still in flight
    - Classical congestion collapse
    - How can this happen with packet conservation
    - Solution: better timers and TCP congestion control
  - Undelivered packets
    - Packets consume resources and are dropped elsewhere in network
    - Solution: congestion control for ALL traffic

# TCP congestion control

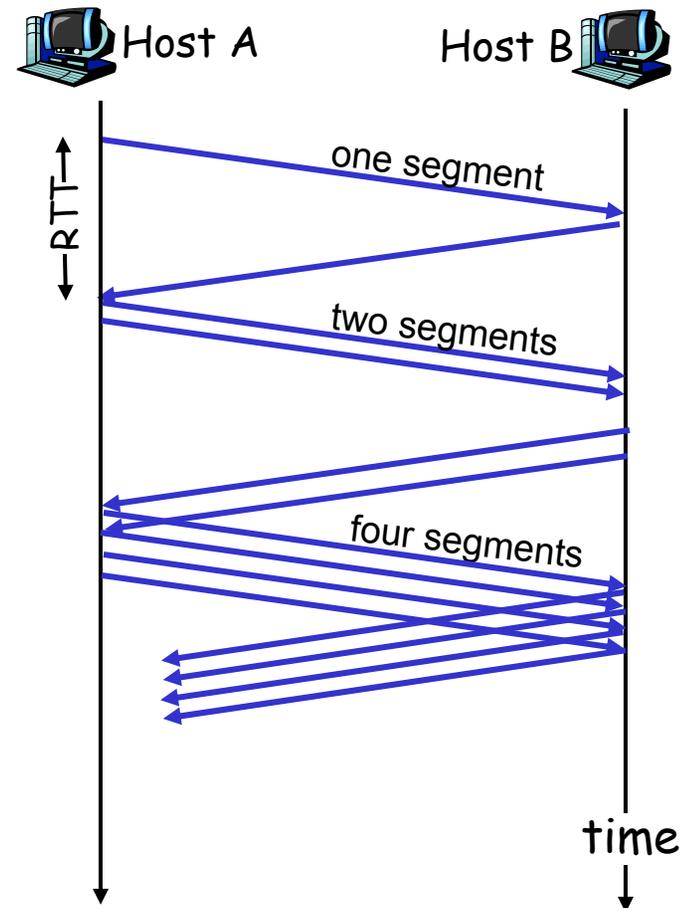
- “probing” for usable bandwidth:
  - Ideally: transmit as fast as possible (**Congwin** as large as possible) without loss
  - Increase **Congwin** until loss (congestion)
  - Loss: *decrease Congwin*, then begin probing (increasing) again
- Two “phases”
  - Slow start
  - Congestion avoidance
- Important variables:
  - **Congwin**
  - **threshold**: defines threshold between two slow start phase, congestion control phase

# TCP slowstart

- Exponential increase (per RTT) in window size (not so slow!)
- Loss event: timeout (Tahoe TCP) and/or or three duplicate ACKs (Reno TCP)

## Slowstart algorithm

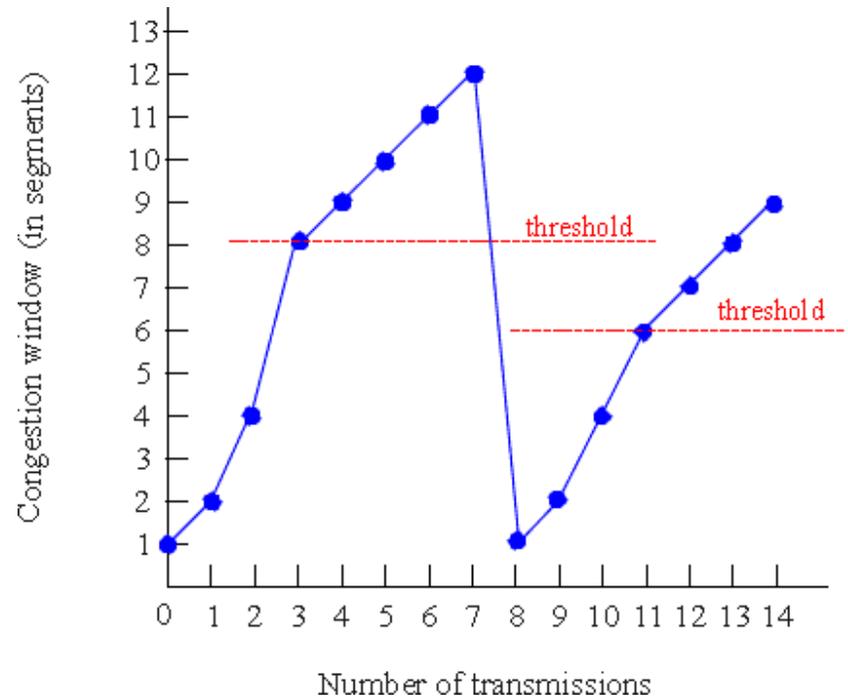
```
initialize: Congwin = 1
for (each segment ACKed)
  Congwin++
until (loss event OR
      CongWin > threshold)
```



# TCP congestion avoidance

## Congestion avoidance

```
/* slowstart is over */
/* Congwin > threshold */
Until (loss event) {
  every w segments ACKed:
    Congwin++
}
threshold = Congwin/2
Congwin = 1
perform slowstart1
```



1: TCP Reno skips slowstart (fast recovery) after three duplicate ACKs

# Return to slow start

- If packet is lost we lose self clocking
  - Need to implement slow-start and congestion avoidance together
- When timeout occurs set *ssthresh* to  $0.5w$ 
  - If  $cwnd < ssthresh$ , use slow start
  - Else use congestion avoidance

# Summary

- ❑ Reviewed principles of transport layer:
  - Reliable data transfer
  - Flow control
  - Congestion control
  - (Multiplexing)
- ❑ Instantiation in the Internet
  - UDP
  - TCP