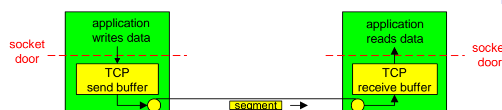
Outline

- 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 TCP congestion control

TCP: Overview

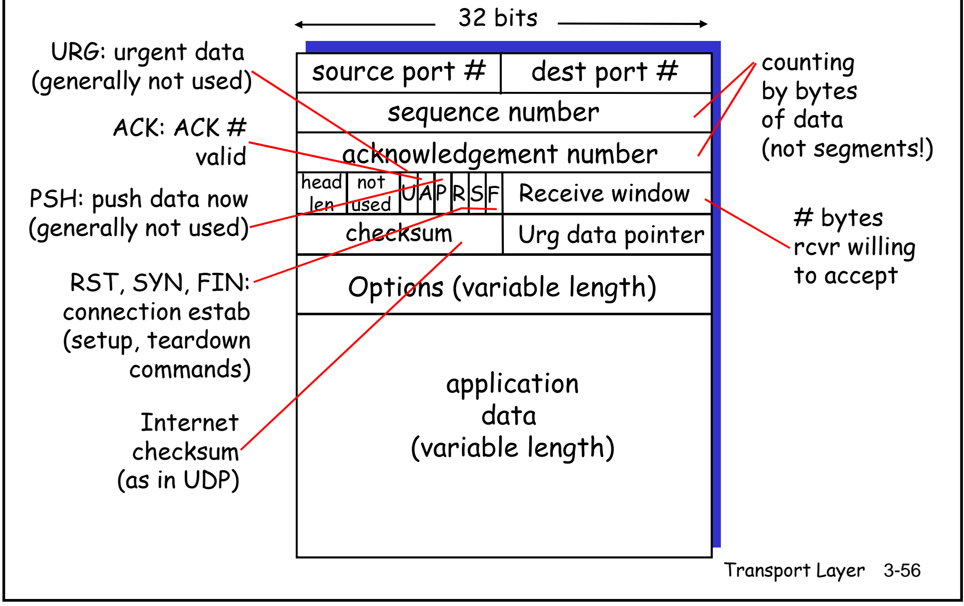
RFCs: 793, 1122, 1323, 2018, 2581

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - 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

TCP segment structure



TCP seq. #'s and ACKs

Seq. #'s:

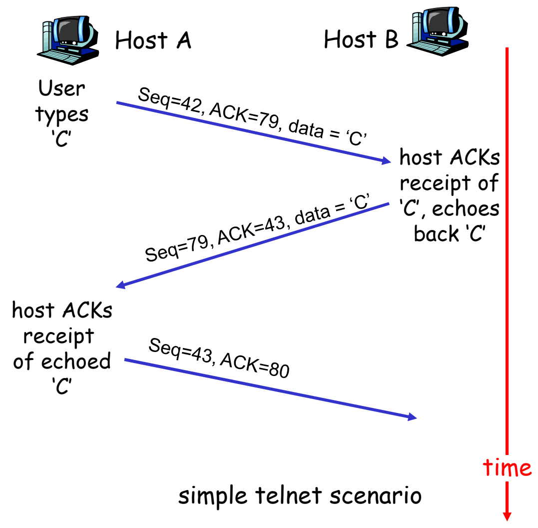
- byte stream "number" of first byte in segment's data

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 implementer



simple telnet scenario

Outline

- 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 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

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:

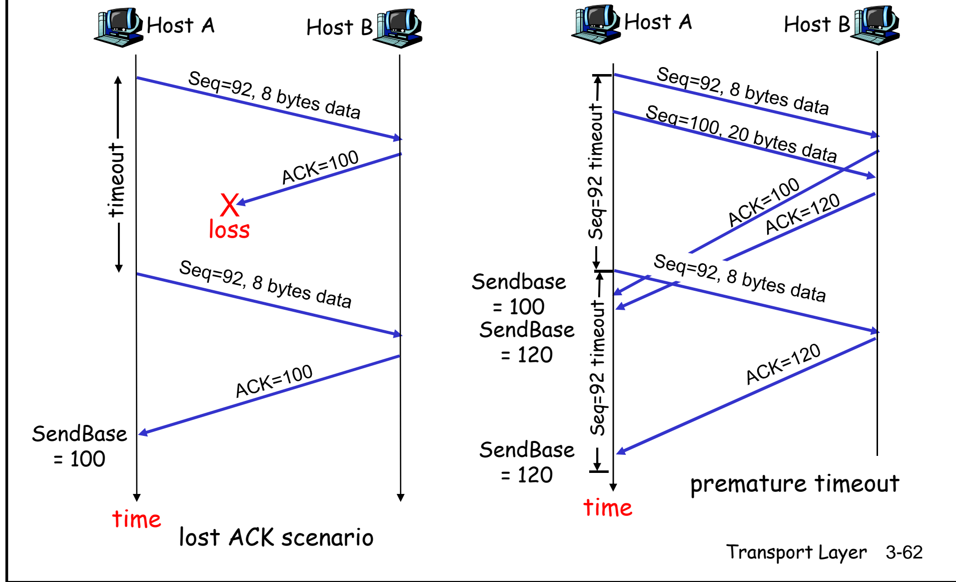
- if acknowledges previously unACKed segments
  - update what is known to be ACKed
  - start timer if there are outstanding segments

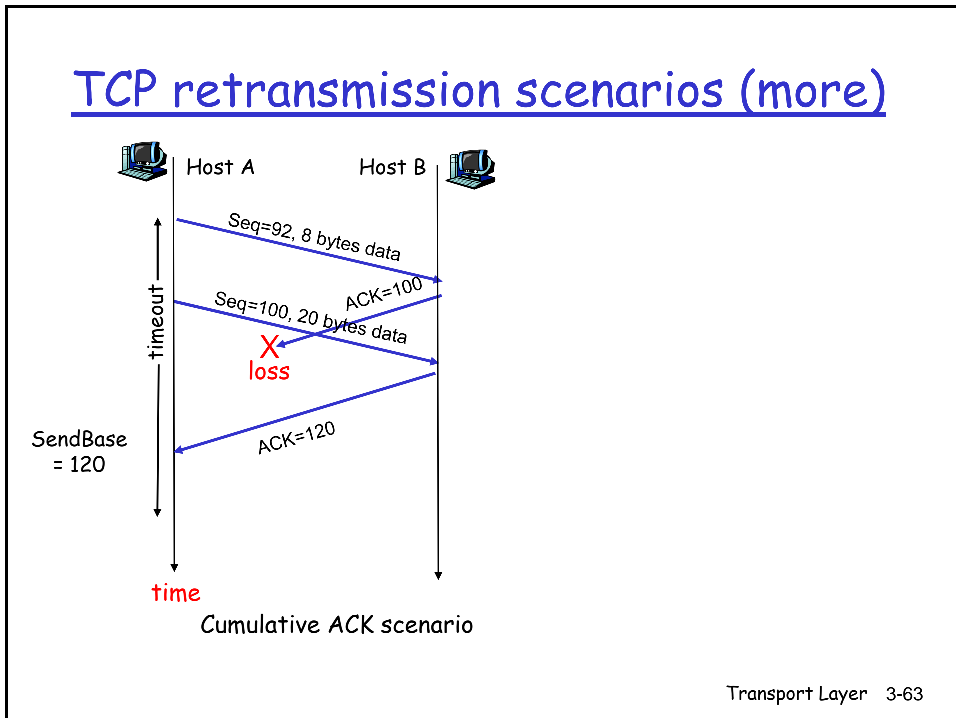
TCP sender (simplified)

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

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
      start timer

  event: ACK received, with ACK field value of y
      if (y > SendBase) {
          SendBase = y
          if (there are currently not-yet-acknowledged segments)
               start timer
          }

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

Comment:
- SendBase-1: last cumulatively ACKed byte

Example:
- SendBase-1 = 71; y= 73, so the rcvr wants 73+ ; y > SendBase, so that new data is ACKed


TCP: retransmission scenarios
Host A
Host B
Seq=92, 8 bytes data
ACK=100
loss
timeout
Seq=92, 8 bytes data
ACK=100
SendBase
= 100
time
lost ACK scenario
Host A
Host B
Seq=92, 8 bytes data
Seq=100, 20 bytes data
ACK=100
ACK=120
Seq=92 timeout
Seq=92, 8 bytes data
Sendbase
= 100
SendBase
= 120
ACK=120
Seq=92 timeout
SendBase
= 120
time
premature timeout
Transport Layer 3-62


TCP retransmission scenarios (more)
Host A
Host B
Seq=92, 8 bytes data
Seq=100, 20 bytes data
ACK=100
loss
timeout
SendBase
= 120
ACK=120
time
Cumulative ACK scenario
Transport Layer 3-63

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-expect seq. # . Gap detected	Immediately send *duplicate ACK*, 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

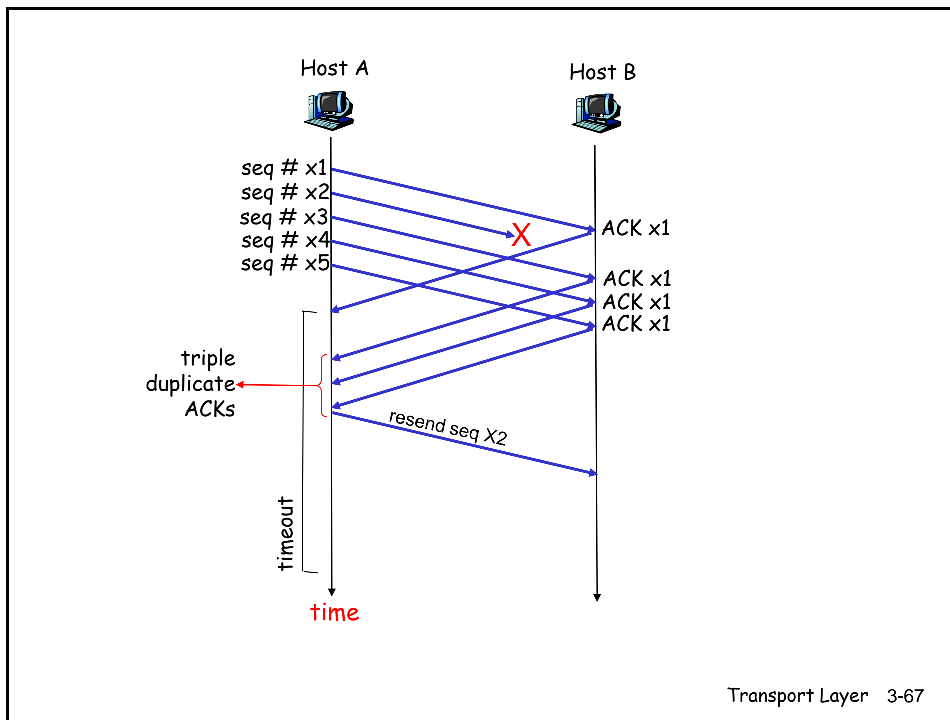
TCP Selective ACKs [RFC 2018]

- A non-mandatory extension to TCP cumulative ACKs that is widely used
- Selective ACK (SACK) allows receiver to ACK a sequence of bytes in addition to number of next expected byte
- Use of SACK is negotiated during TCP connection opening
  - uses TCP options field to convey sequence number ranges

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 for that segment

- If sender receives 3 ACKs for same data, it assumes 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
               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
               }
```

a duplicate ACK for already ACKed segment

fast retransmit

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?

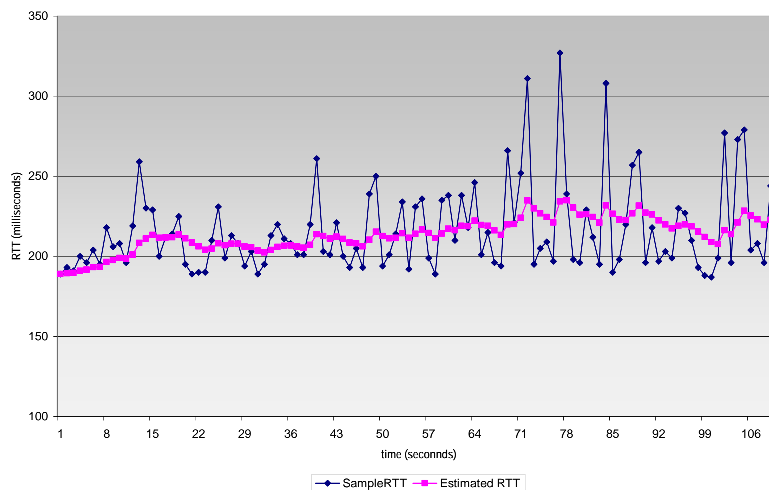
- **SampleRTT**: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- **SampleRTT** will vary, want estimated RTT "smoother"
  - average several recent measurements, not just current **SampleRTT**

TCP Round Trip Time and Timeout

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

- Exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: α = 0.125

Example RTT estimation:



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:

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

(typically, β = 0.25)
```

Then set timeout interval:

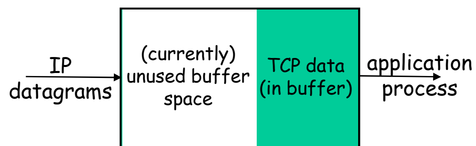
```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```


Outline

- 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 TCP congestion control

TCP Flow Control

- receive side of TCP connection has a receive buffer:

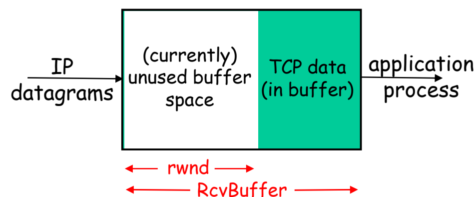


- app process may be slow at reading from buffer

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

- *speed-matching service:* matching send rate to receiving application's drain rate

TCP Flow control: how it works



(suppose TCP receiver discards out-of-order segments)

- unused buffer space:

= **rwnd**

= **RcvBuffer-[LastByteRcvd - LastByteRead]**

- receiver: advertises unused buffer space by including `rwnd` value in segment header
- sender: limits # of unACKed bytes to `rwnd`
  - guarantees receiver's buffer doesn't overflow

TCP Flow Control Example

- Example: slow receiver
  - Recv buffer fills up and window shrinks to 0
  - Send TCP learns of empty window and stops
  - Send buffer fills up with bytes from appl process
  - Send TCP asks OS to block sender appl process
- Once receiver catches up
  - Window opens, Send TCP learns new window size
  - Send TCP resumes transmission
  - Send TCP buffer frees up
  - Send TCP asks OS to unblock sender process

Outline

- 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 TCP congestion control

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

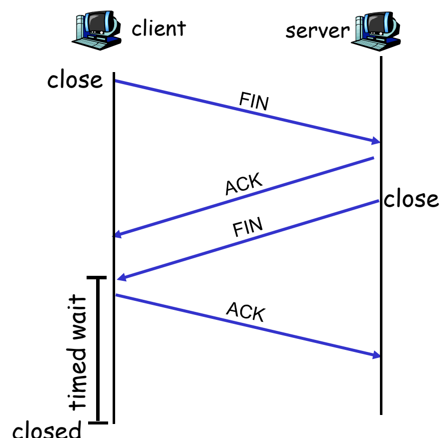
TCP Connection Management (cont.)

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.



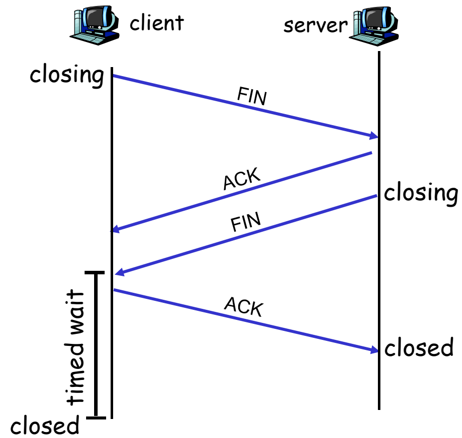
TCP Connection Management (cont.)

Step 3: client receives FIN, replies with ACK.

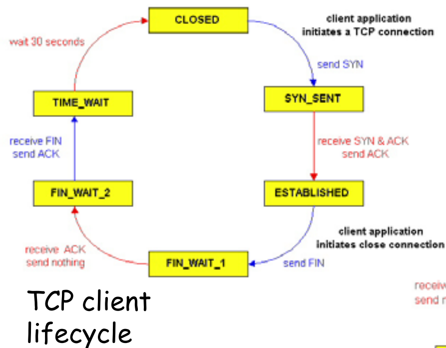
- Enters "timed wait" - will respond with ACK to received FINs

Step 4: server, receives ACK. Connection closed.

Note: with small modification, can handle simultaneous FINs.

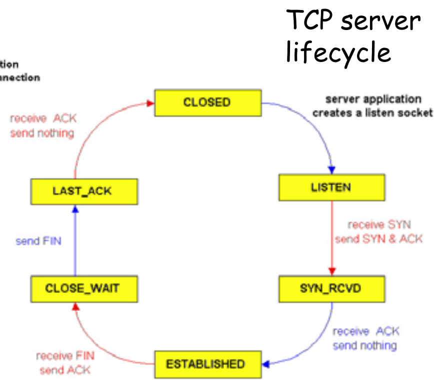


TCP Connection Management (cont)



TCP client lifecycle

TCP server lifecycle



Outline

- 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 TCP congestion control

Transport Layer 3-82

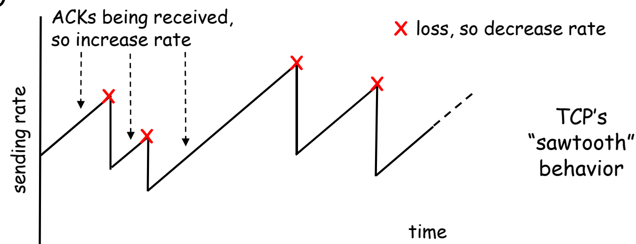
TCP congestion control:

- *goal:* TCP sender should transmit as fast as possible, but without congesting network
  - Q: how to find rate *just* below congestion level
- decentralized: each TCP sender sets its own rate, based on *implicit* feedback:
  - *ACK:* segment received (a good thing!), network not congested, so increase sending rate
  - *lost segment:* assume loss due to congested network, so decrease sending rate

Transport Layer 3-83

TCP congestion control: bandwidth probing

- "probing for bandwidth": increase transmission rate on receipt of ACK, until eventually loss occurs, then decrease transmission rate
  - continue to increase on ACK, decrease on loss (since available bandwidth is changing, depending on other connections in network)



TCP's "sawtooth" behavior

- Q: how fast to increase/decrease?
  - details to follow

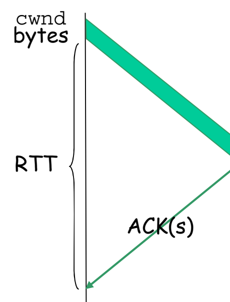
TCP Congestion Control: details

- sender limits rate by limiting number of unACKed bytes "in pipeline":

  LastByteSent-LastByteAcked ≤ cwnd
  - **cwnd:** differs from **rwnd** (how, why?)
  - sender limited by **min(cwnd,rwnd)**
- roughly,

$$\text{rate} = \frac{\text{cwnd}}{\text{RTT}} \text{ bytes/sec}$$

- **cwnd** is dynamic, function of perceived network congestion



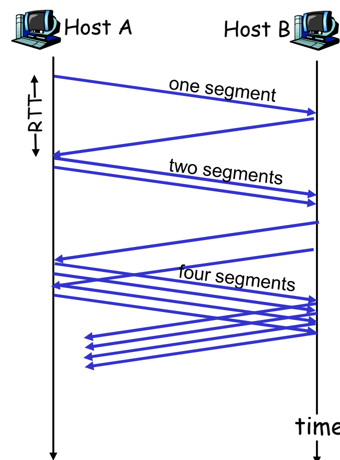
TCP Congestion Control: more details

segment loss event: reducing
- timeout: no response from receiver
  - cut cwnd to 1
- 3 duplicate ACKs: at least some segments getting through (recall fast retransmit)
  - cut cwnd in half, less aggressively than on timeout

ACK received: increase
- slowstart phase:
  - increase exponentially fast (despite name) at connection start, or following timeout
- congestion avoidance:
  - increase linearly

TCP Slow Start

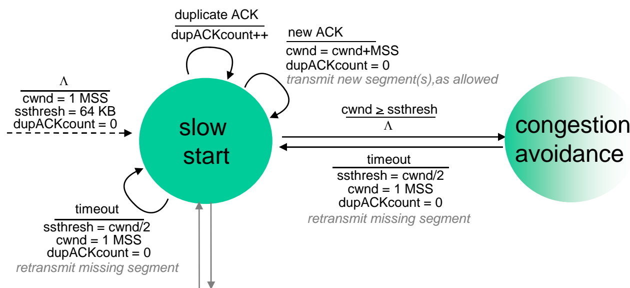
- when connection begins, cwnd = 1 MSS
  - example: MSS = 500 bytes & RTT = 200 msec
  - initial rate = 20 kbps
- available bandwidth may be >> MSS/RTT
  - desirable to quickly ramp up to respectable rate
- increase rate exponentially until first loss event or when threshold reached
  - double cwnd every RTT
  - done by incrementing cwnd by 1 for every ACK received



Transitioning into/out of slowstart

ssthresh: **cwnd** threshold maintained by TCP

- on loss event: set **ssthresh** to **cwnd/2**
  - remember (half of) TCP rate when congestion last occurred
- when **cwnd >= ssthresh**: transition from slowstart to congestion avoidance phase



TCP: congestion avoidance

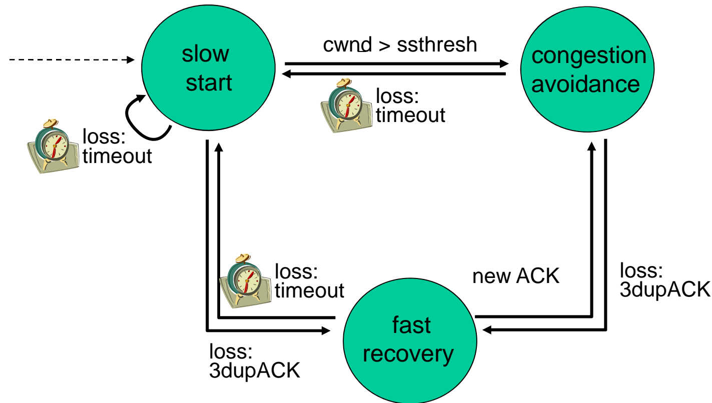
- when **cwnd > ssthresh** grow **cwnd** linearly
  - increase **cwnd** by 1 MSS per RTT
  - approach possible congestion slower than in slowstart
  - implementation: **cwnd = cwnd + MSS/cwnd** for each ACK received

AIMD

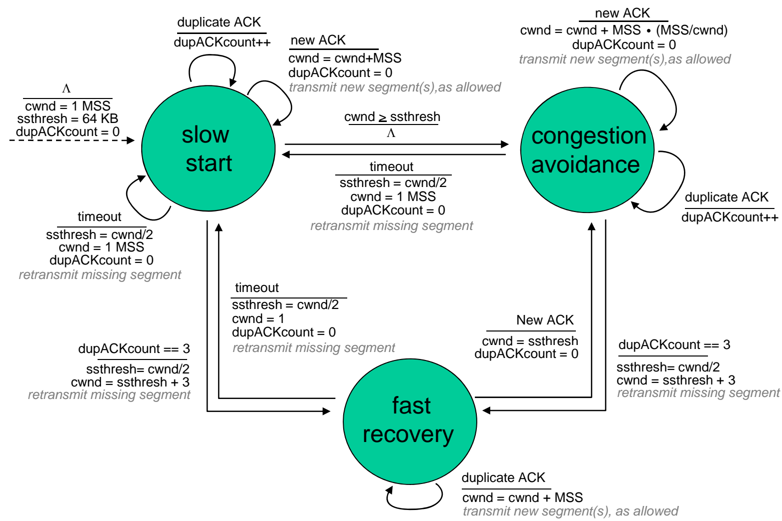
- ACKs: increase **cwnd** by 1 MSS per RTT: additive increase
- loss: cut **cwnd** in half (non-timeout-detected loss): multiplicative decrease

AIMD: Additive Increase Multiplicative Decrease

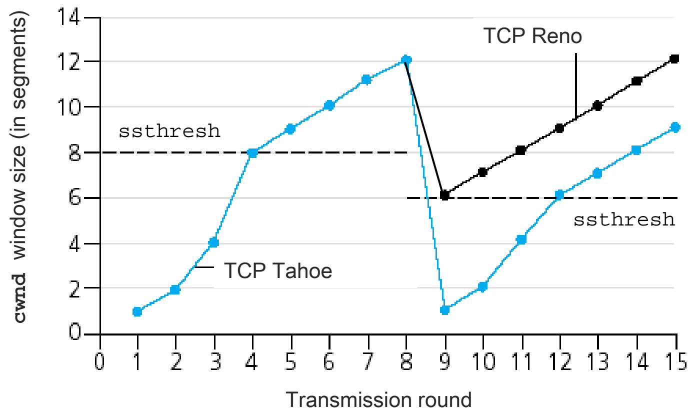
TCP congestion control FSM: overview



TCP congestion control FSM: details



Popular "flavours" of TCP



Summary: TCP Congestion Control

- when **cwnd < ssthresh**, sender in slow-start phase, window grows exponentially.
- when **cwnd >= ssthresh**, sender is in congestion-avoidance phase, window grows linearly.
- when triple duplicate ACK occurs, **ssthresh** set to **cwnd/2, cwnd** set to ~ **ssthresh**
- when timeout occurs, **ssthresh** set to **cwnd/2, cwnd** set to 1 MSS.

Summary

- principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- instantiation and implementation in the Internet
  - UDP
  - TCP