CEG 7450:
(TCP Congestion Control)

Reading

- **[BCS94]** R. Braden, D. Clark & S.Shenker. "Integrated Services in the Internet Architecture: an Overview", RFC 1633, June 1994.

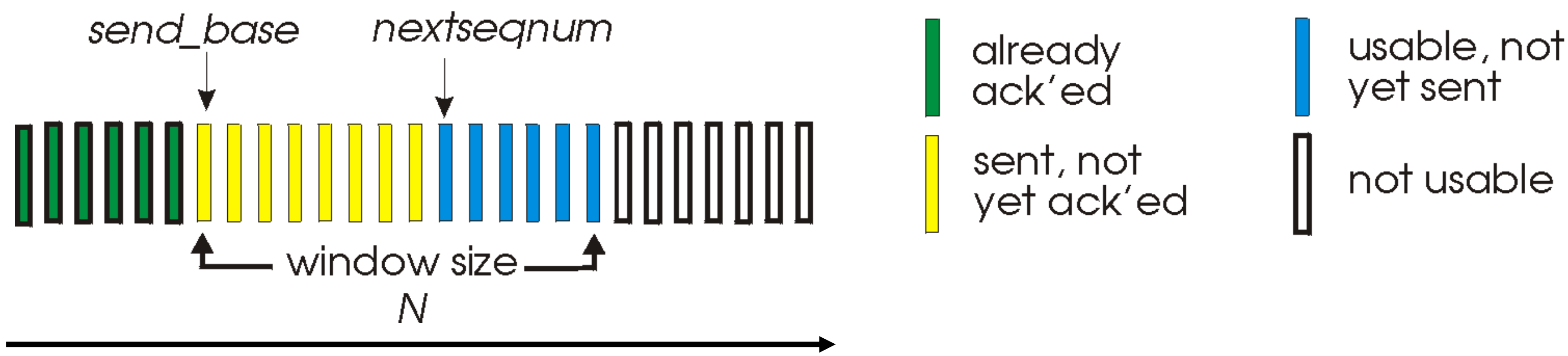
Problem

- How much traffic do you send?
- Two components
  - flow control
    - make sure that the receiver can receive as fast as you send
  - congestion control
    - make sure that the network delivers the packets to the receiver
- However, in TCP, these mechanisms are inherently integrated with reliability

Go-Back-N

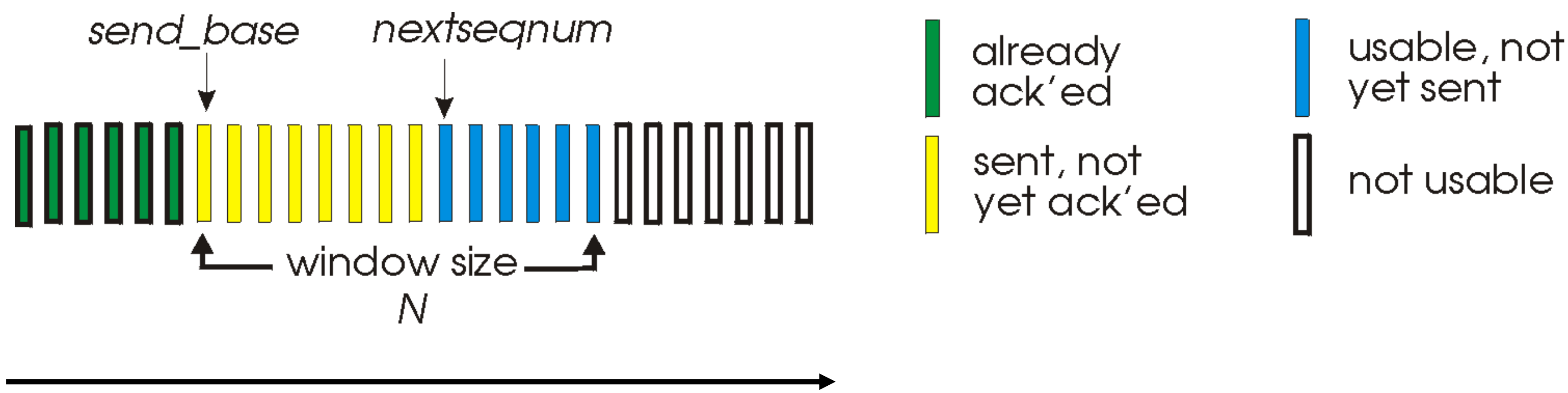
Sender:

- k-bit seq # in packet header
- "window" of up to N, consecutive unack'ed packets allowed



- timer for each in-flight packet
- *timeout(n):* retransmit packet n and all higher seq # packets that were sent in window

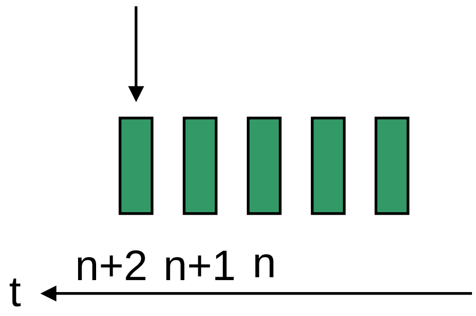
Cumulative ACK



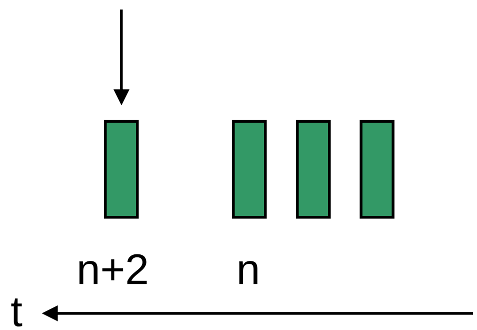
- ACK(n): ACKs all packets up to, including seq # n
  - may receive duplicate ACKs

Packet receiving order

- In order

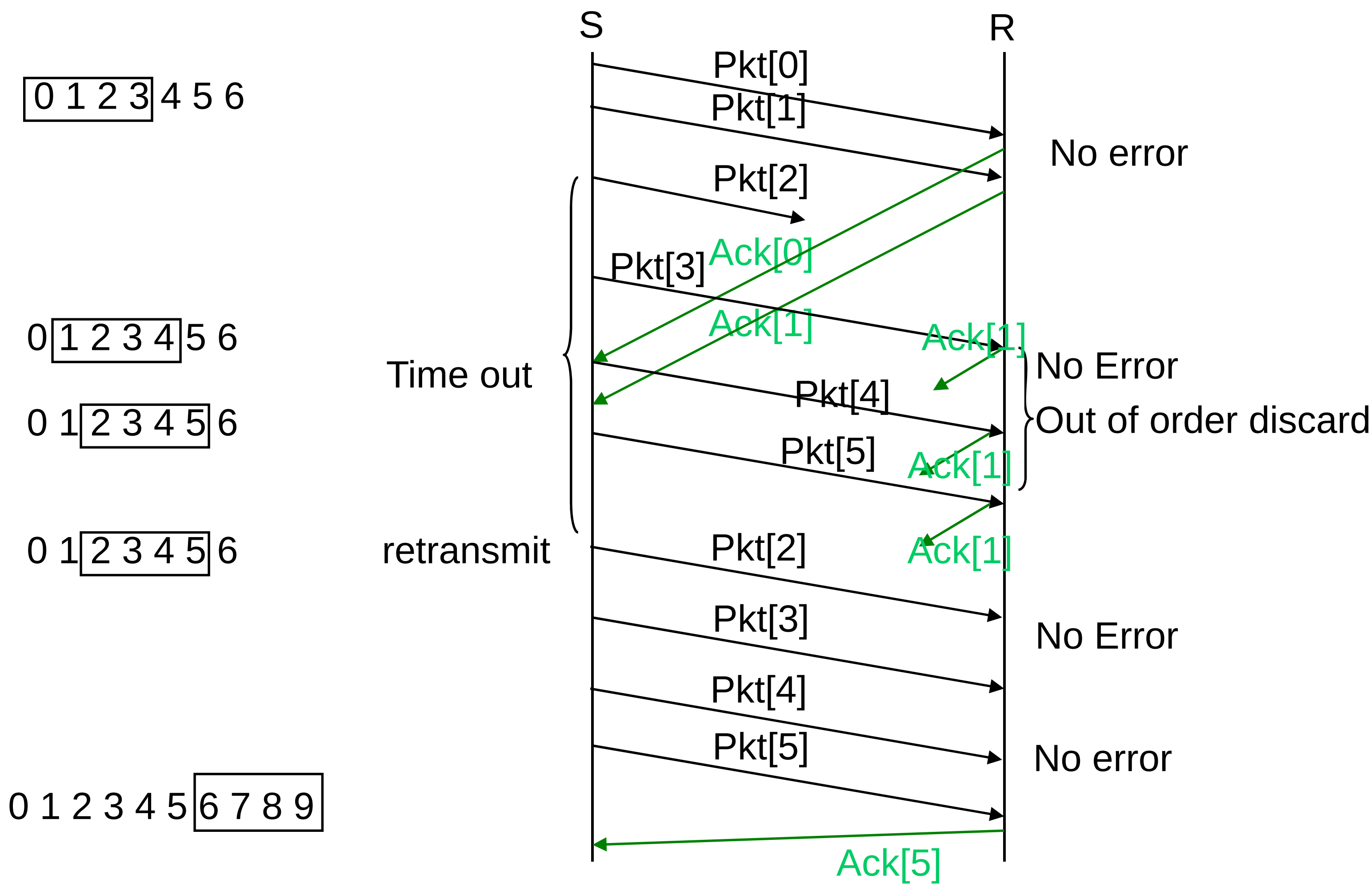


- Out of order



Go-Back-N

Window size = 4



Sender Actions

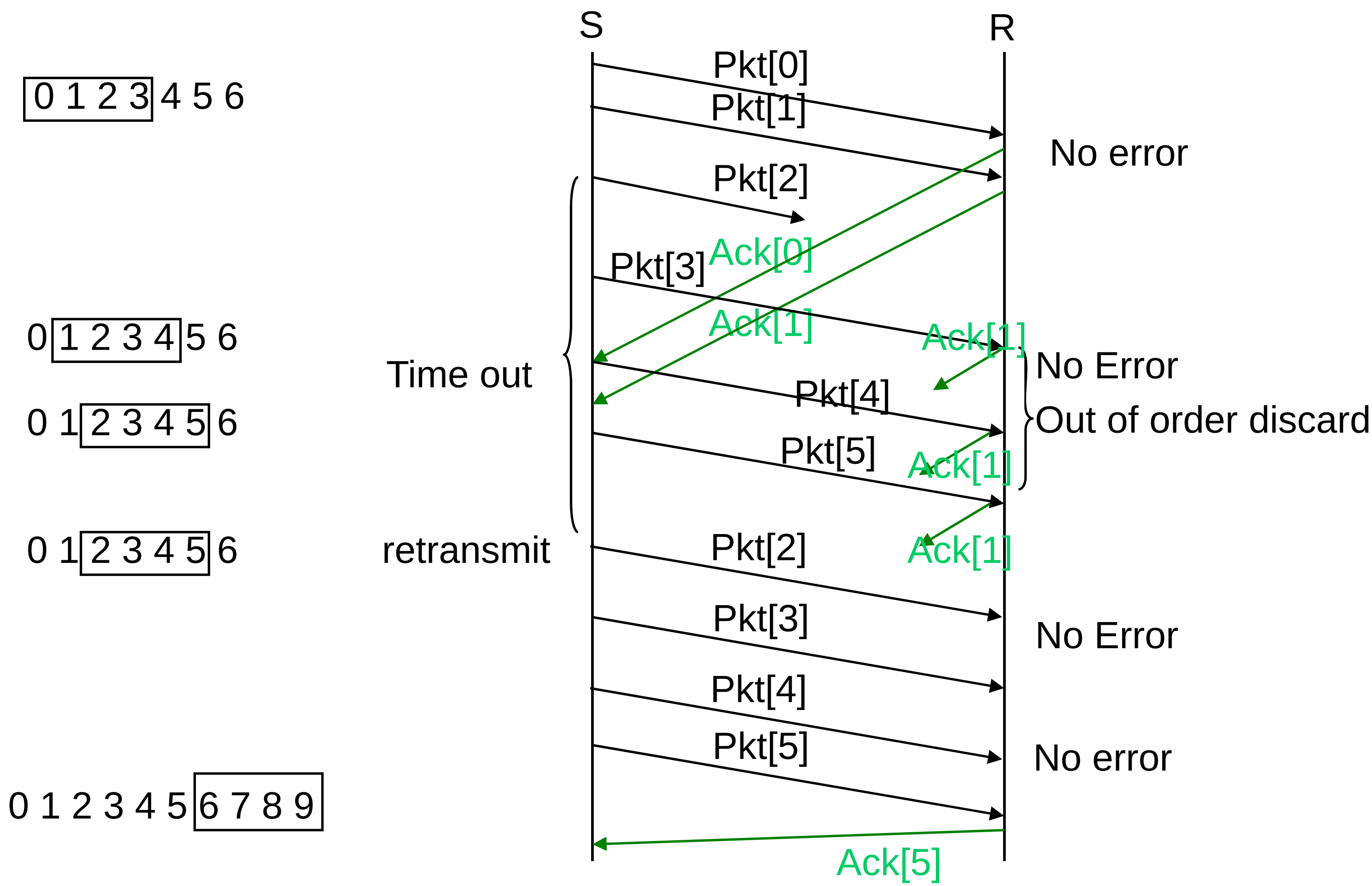
- Application data received
  - Wait if window is full
  - Construct segment ← nextseqnum
  - Buffer a copy
  - Set up a timer
  - Send: hand to IP layer
  - nextsequm ++
- ACK received
  - If ACK(n) > send_base: update send_base
  - Slide window
  - Otherwise, ignore
- Timer times out
  - *timeout(n):* retransmit packet n and all higher seq # packets that were sent in window

Receiver Actions

- Receiver maintain: next in-order sequence number n (next expected segment)
- Segment n received in order
  - Deliver to application layer
  - Send ACK(n)
  - Update next in-order sequence number
- Segment i received our of order
  - Discard i
  - Send ACK(last correctly received seq #)

Go-Back-N

Window size = 4



Drawbacks of Go-Back-N

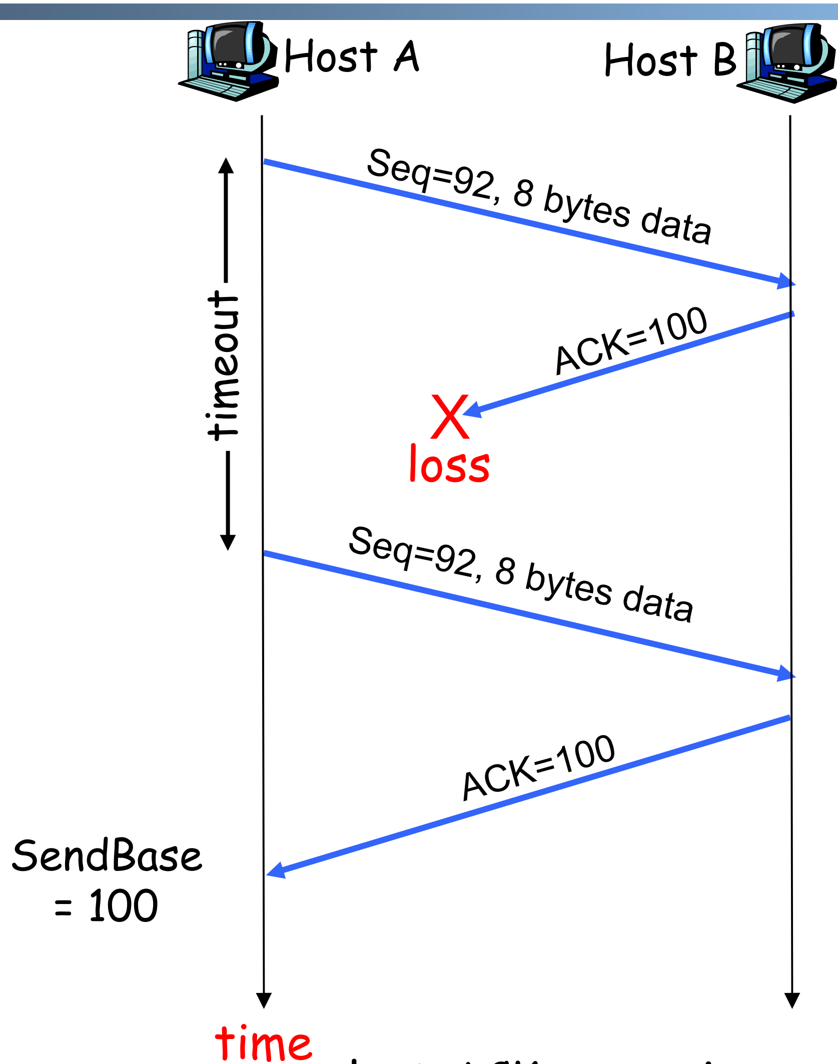
- Retransmit all packets that were sent but not yet acked in the window upon a time-out
- Receiver discard out of order packets

TCP reliable data transfer

- TCP creates reliable service on top of IP's unreliable service
- Pipelined segment transmission
- Cumulative acks
- TCP uses single retransmission timer

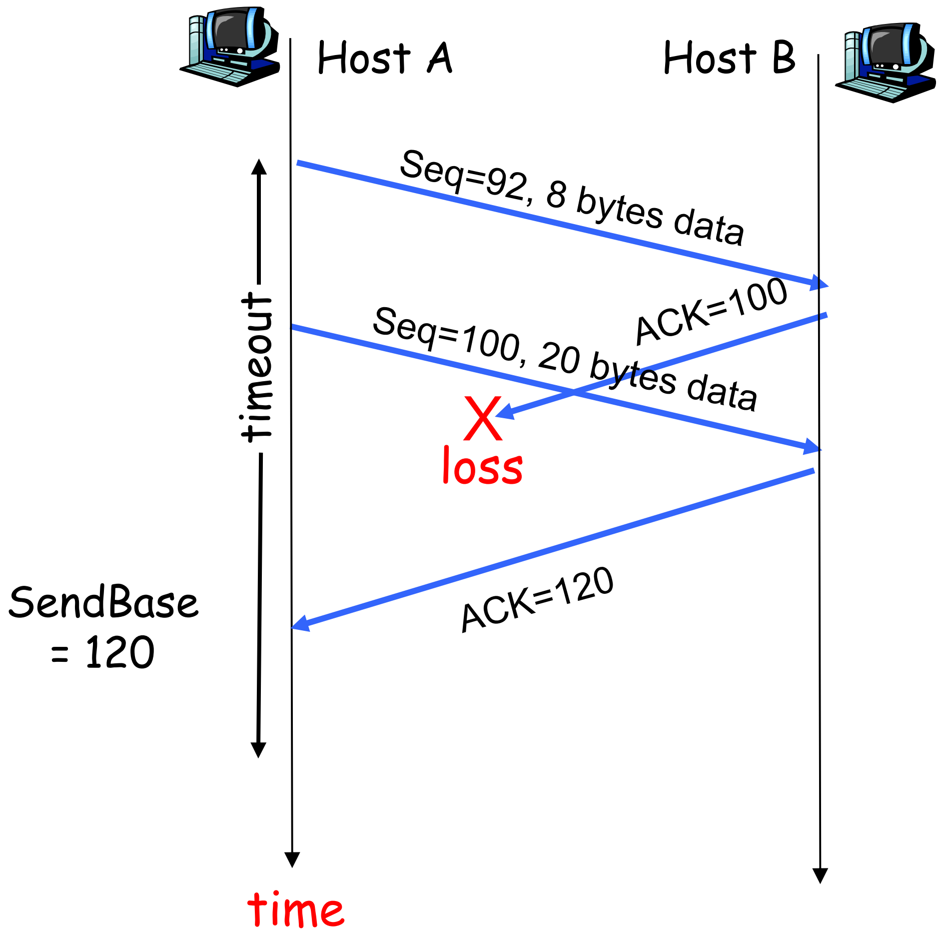
- Retransmissions are triggered by:
  - timeout events

TCP: lost ack



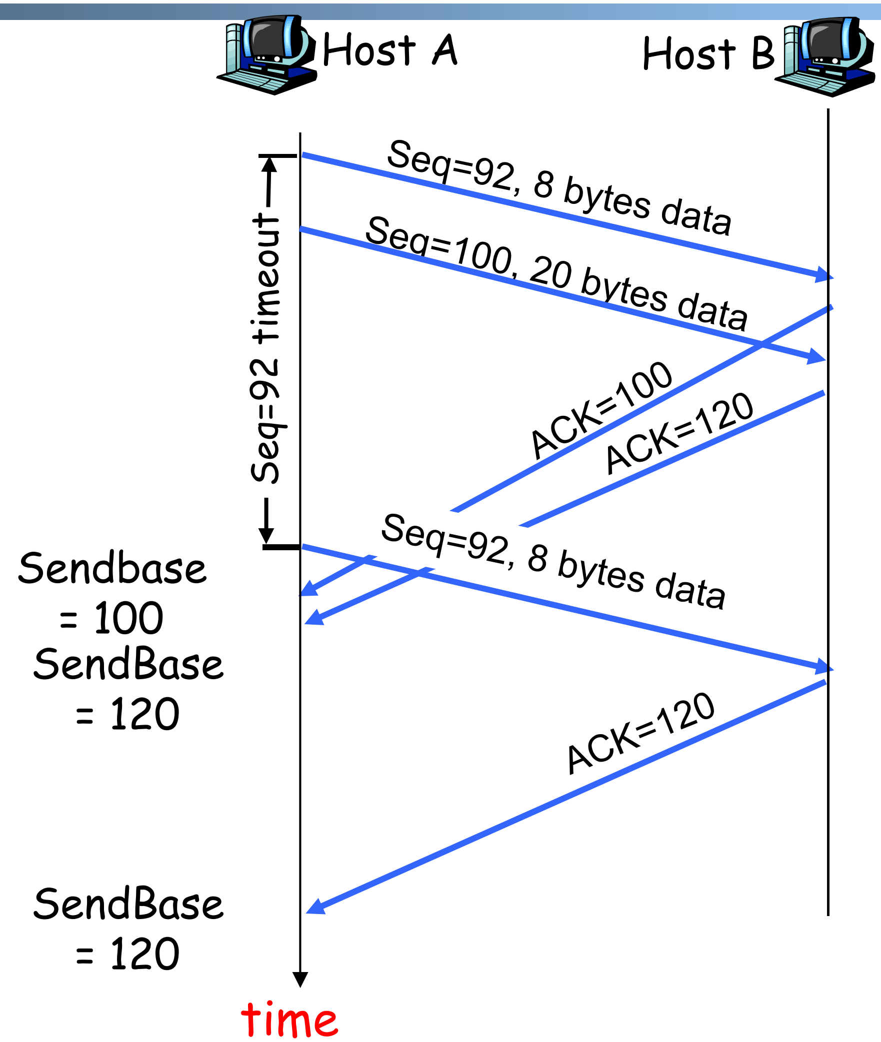
lost ACK scenario

TCP: cumulative ack



Cumulative ACK scenario

TCP: premature timeout



TCP sender events:

data received from application:

- create segment with seq #
- seq # is byte-stream number of first data byte in segment
- start timer if not already running (timer is for oldest unacked segment)

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 with timeout interval doubled

  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 ack'ed byte

Example:

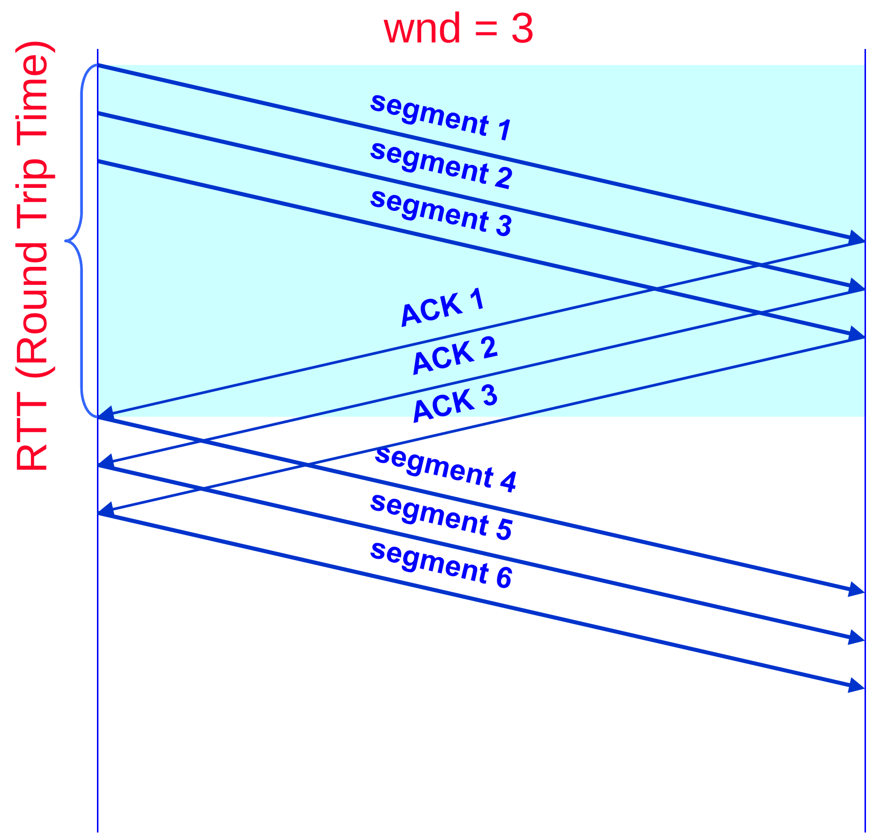
- SendBase-1 = 71; y= 73, so the rcvr wants 73+ ; y > SendBase, so that new data is acked

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

Flow control: Window Size and Throughput

- Sliding-window based flow control:
  - larger window → higher throughput
    - throughput = wnd/RTT
  - need to worry about sequence number wrapping
  - cumulative ack
  - timeout, retransmission
- Remember: window size controls throughput



Why do You Care About Congestion Control?

- Otherwise you get to congestion collapse
- How might this happen?
  - assume network is congested (a router drops packets)
  - you learn the receiver didn’t get the packet
    - either by ACK, NACK, or Timeout
  - what do you do? retransmit packet
  - still receiver didn’t get the packet
  - retransmit again
  - …. and so on …
  - and now assume that everyone is doing the same!
- Network will become more and more congested
  - and this with duplicate packets rather than new packets!

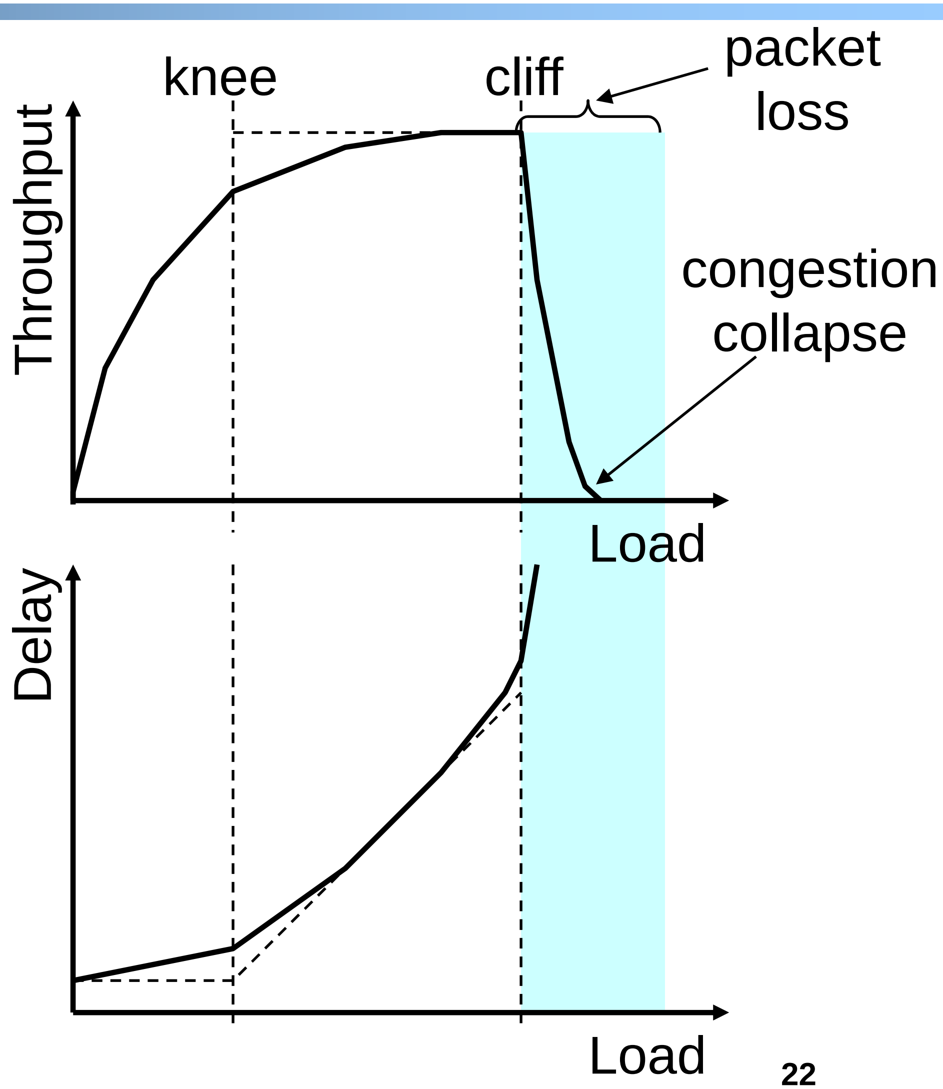
Solutions?

- Increase buffer size. Why not?
- Slow down
  - if you know that your packets are not delivered because network congestion, slow down
- Questions:
  - how do you detect network congestion?
  - by how much do you slow down?

What's Really Happening?

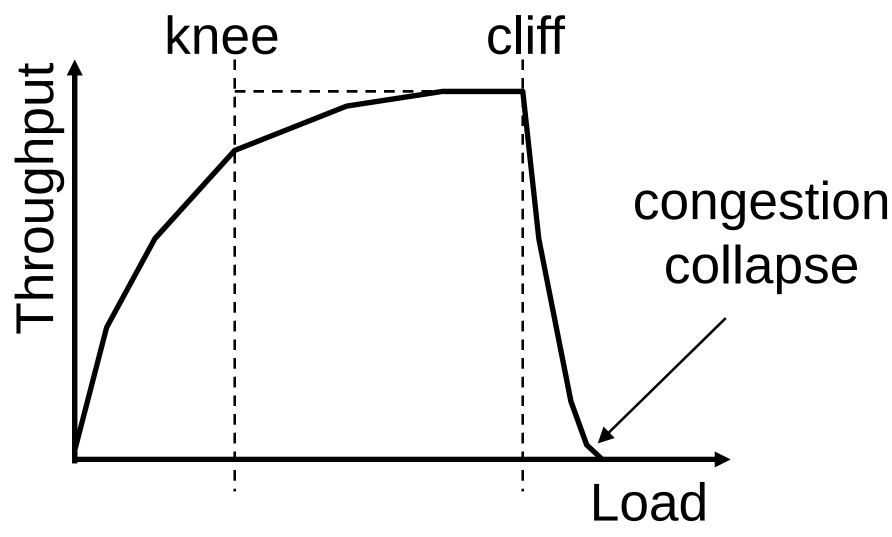
- knee – point after which
  - throughput increases very slow
  - delay increases fast
- cliff – point after which
  - throughput starts to decrease very fast to zero (congestion collapse)
  - delay approaches infinity

- Note (in an M/M/1 queue)
  - delay = $\dfrac{1/\mu}{1-\dfrac{\lambda}{\mu}}$



Congestion Control vs. Congestion Avoidance

- Congestion control goal
  - stay left of cliff
- Congestion avoidance goal
  - stay left of knee



Goals

- Operate near the knee point
- Remain in equilibrium
- How to maintain equilibrium?
  - don’t put a packet into network until another packet leaves. How do you do it?
  - use ACK: send a new packet only after you receive and ACK. Why?
  - maintain number of packets in network “constant”

How Do You Do It?

- Detect when network approaches/reaches knee point
- Stay there

- Questions
  - how do you get there?
  - what if you overshoot (i.e., go over knee point) ?

- Possible solution:
  - increase window size until you notice congestion
  - decrease window size if network congested

Possible Choices

- Window increase, decrease algorithms
  - additive increase, multiplicative decrease
  - multiplicative increase, additive decrease
  - multiplicative increase, multiplicative decrease
- Convergence
  - Which converge?
  - Which converge to an efficient operating point?

The Choice

- Additive increase, multiplicative decrease
- Intuition:
- Let L_i denote average queue length and W_i denote window size over period I
  - on equilibrium: $L_{i+1} = N$
  - on congestion: $L_{i+1} = N + \gamma L_i \rightarrow L_{i+k} = N\sum_{l=0}^{k-1}\gamma^l + \gamma^k L_i$
    - queue size increase exponentially → need to reduce window size at least as fast: $W_{i+1} = dW_i \quad (d < 1)$
  - on no congestion → increase window size: $W_{i+1} = W_i + u \quad (u << W_{\max})$ (see [CJ89])

TCP Congestion Control

- Maintains three variables:
  - cwnd – congestion window
  - flow_win – flow window; receiver advertised window
  - ssthresh – threshold size (used to update cwnd)
- For sending use: win = **min**(flow_win, cwnd)

TCP: Slow Start

- Goal: discover congestion quickly
- How?
  - quickly increase *cwnd* until network congested → get a rough estimate of the optimal of *cwnd*
- How do we know when network is congested?
  - packet loss (TCP, [Jac88])
    - over the cliff here → congestion control
  - congestion notification (DEC Bit scheme, [RJ88]
    - over the knee but before the cliff → congestion avoidance
- How do we know a packet is lost? (latter…)

TCP: Slow Start

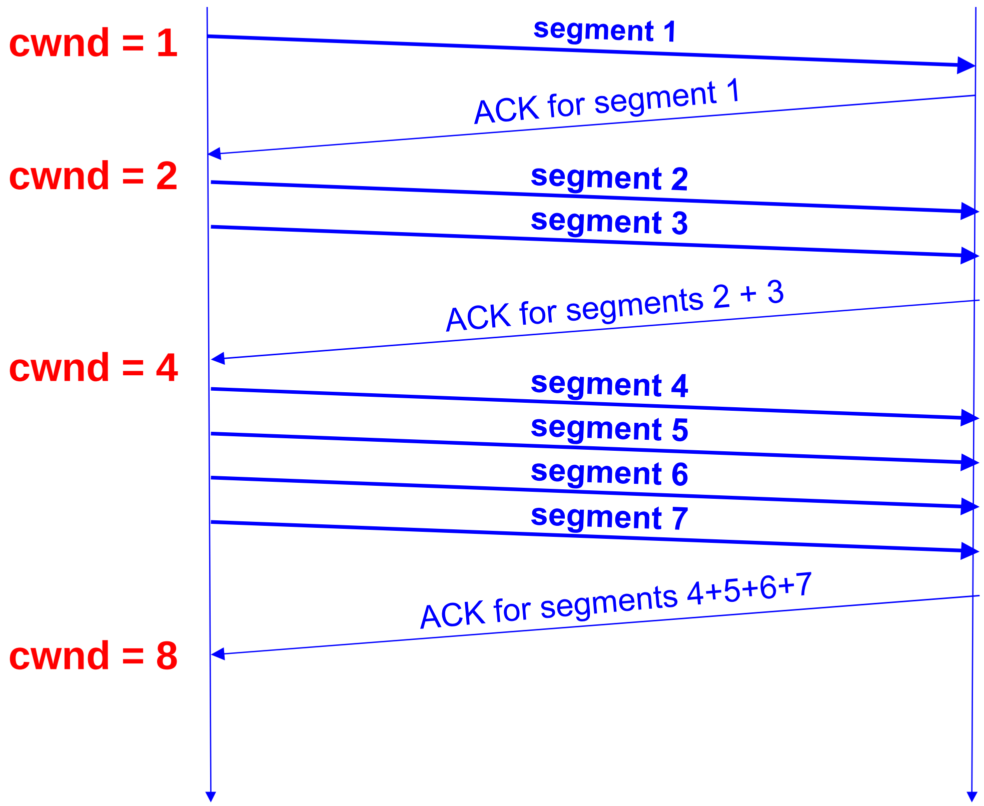
- Whenever starting traffic on a new connection, or whenever increasing traffic after congestion was experienced:
    - Set *cwnd* =1
    - Each time a segment is acknowledged increment *cwnd* by one (*cwnd*++).

- Does Slow Start increment slowly? Not really. In fact, the increase of *cwnd* is exponential

Slow Start Example

- The congestion window size grows very rapidly

- TCP slows down the increase of *cwnd* when ***cwnd >= ssthresh***



Congestion Avoidance

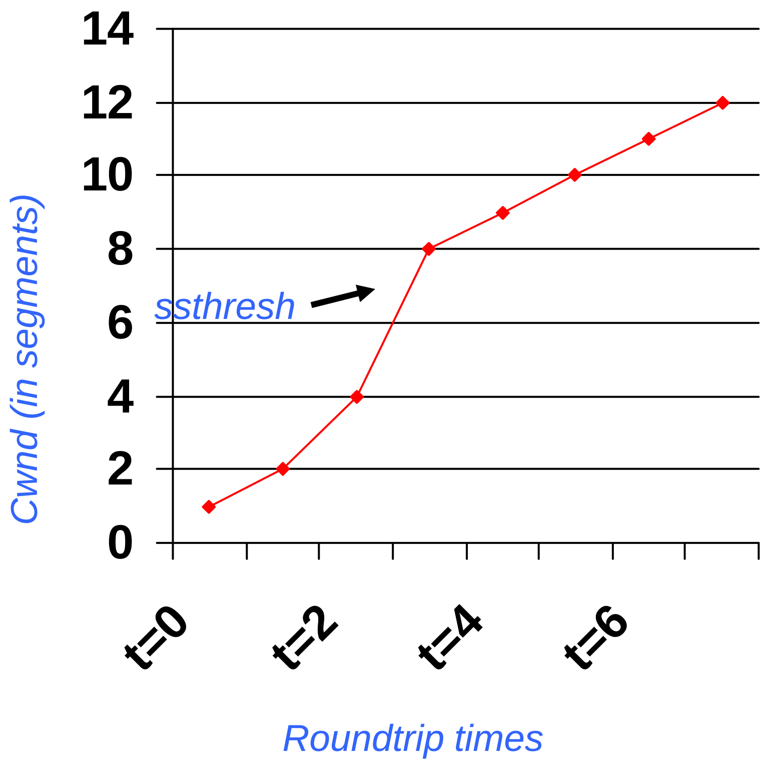
- Goal: maintain operating point at the left of the cliff:
- How?
  - additive increase: starting from the rough estimate, slowly increase cwnd to probe for additional available bandwidth
  - multiplicative decrease: cut congestion window size aggressively if a timeout occurs

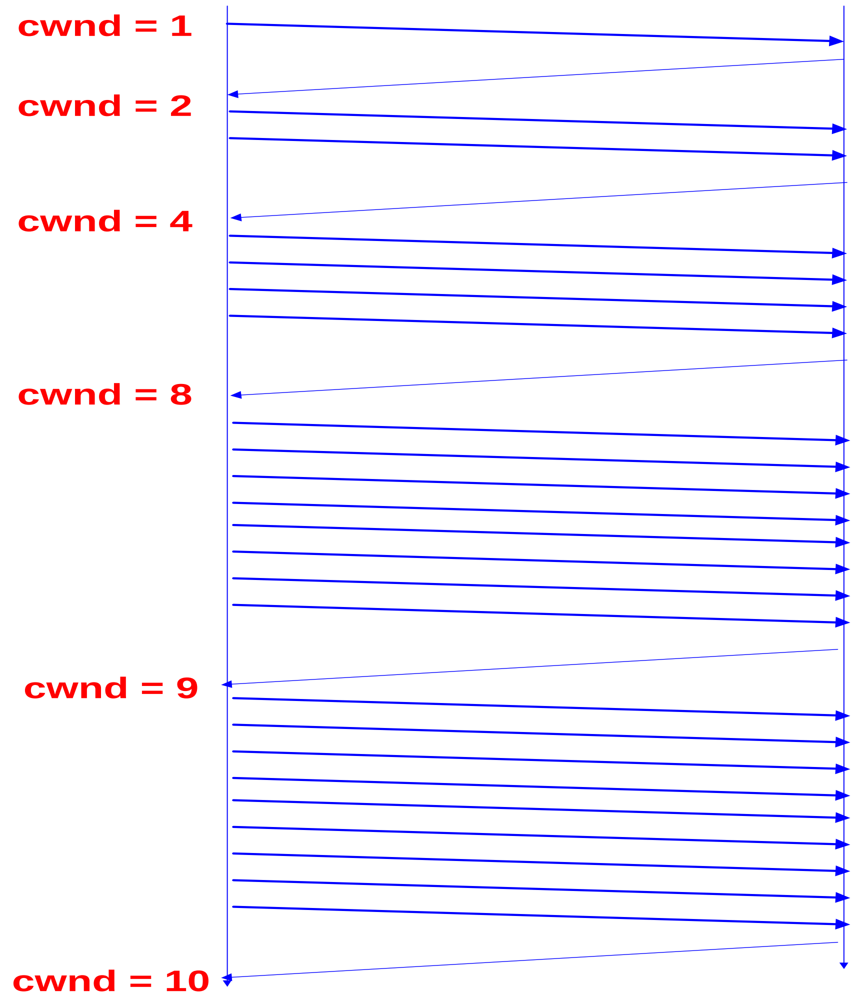
Congestion Avoidance

- Slow down “Slow Start”
- **If** *cwnd > ssthresh* **then**
  each time a segment is acknowledged
  increment *cwnd* by *1/cwnd* (*cwnd += 1/cwnd*).
- So *cwnd* is increased by one only if all segments have been acknowledged.
- (more about *ssthresh* latter)

Slow Start/Congestion Avoidance Example

- Assume that *ssthresh = 8*



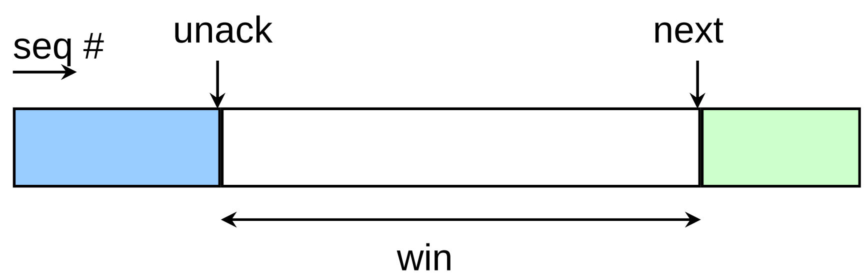


Putting Everything Together: TCP Pseudocode

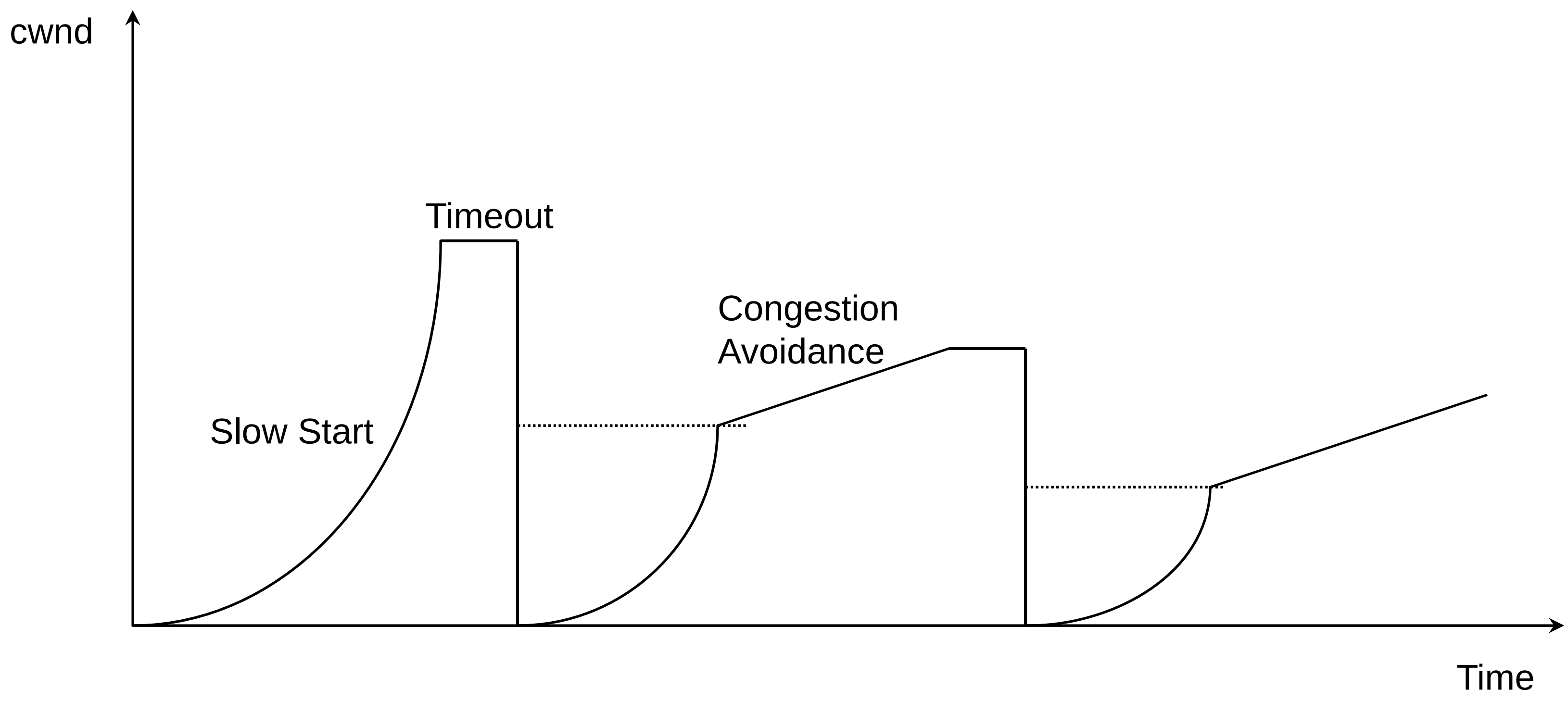
```
Initially:
    CongWin = 1;
    ssthresh = infinite;
New ack received:
    if (CongWin < ssthresh)
        /* Slow Start*/
     CongWin = CongWin+ 1;
    else
        /* Congestion Avoidance */
     CongWin = CongWin +1/CongWin;
Timeout:
    /* Multiplicative decrease */
    ssthresh = win/2;
    CongWin = 1;
3 Duplicate Acks:
    ssthresh = win/2;
    CongWin = ssthresh;
```

```
while (next < unack + win)
    transmit next packet;

where   win = min(CongWin,
                  flow_win);
```



The big picture

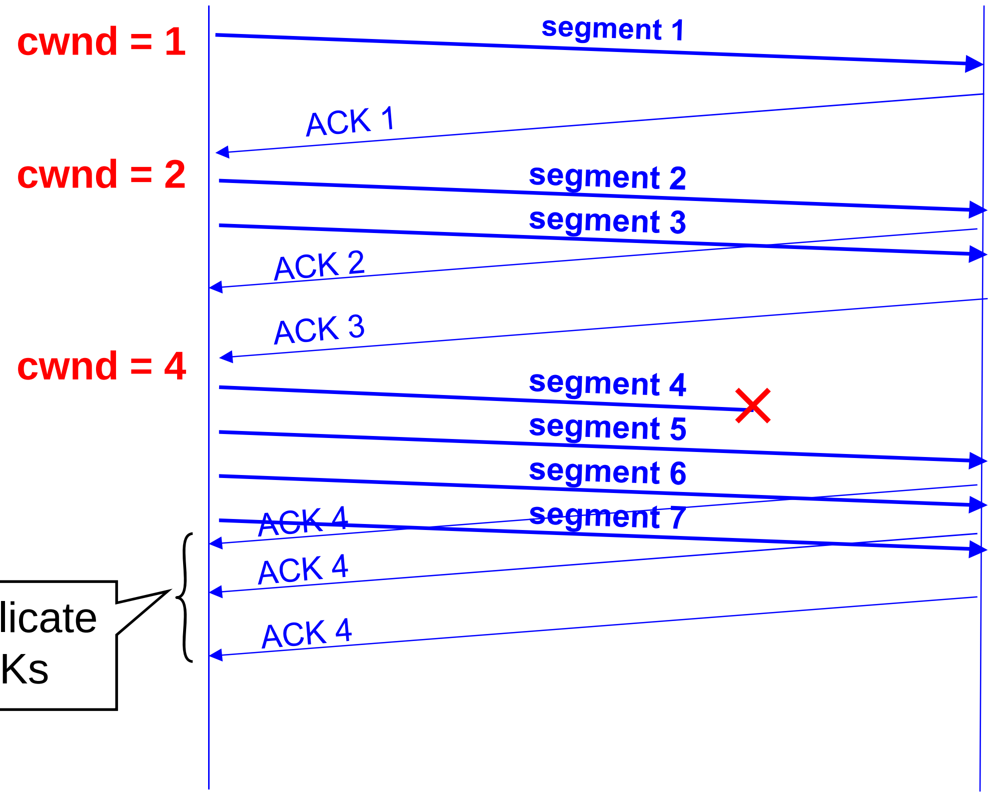


Packet Loss Detection

- Wait for Retransmission Time Out (RTO)
- What's the problem with this?
- Because RTO is performance killer
- In BSD TCP implementation, RTO is usually more than 1 second
  - the granularity of RTT estimate is 500 ms
  - retransmission timeout is at least two times of RTT
- Solution: Don't wait for RTO to expire

Fast Retransmit

- Resend a segment after 3 duplicate ACKs
  - remember a duplicate ACK means that an out-of sequence segment was received

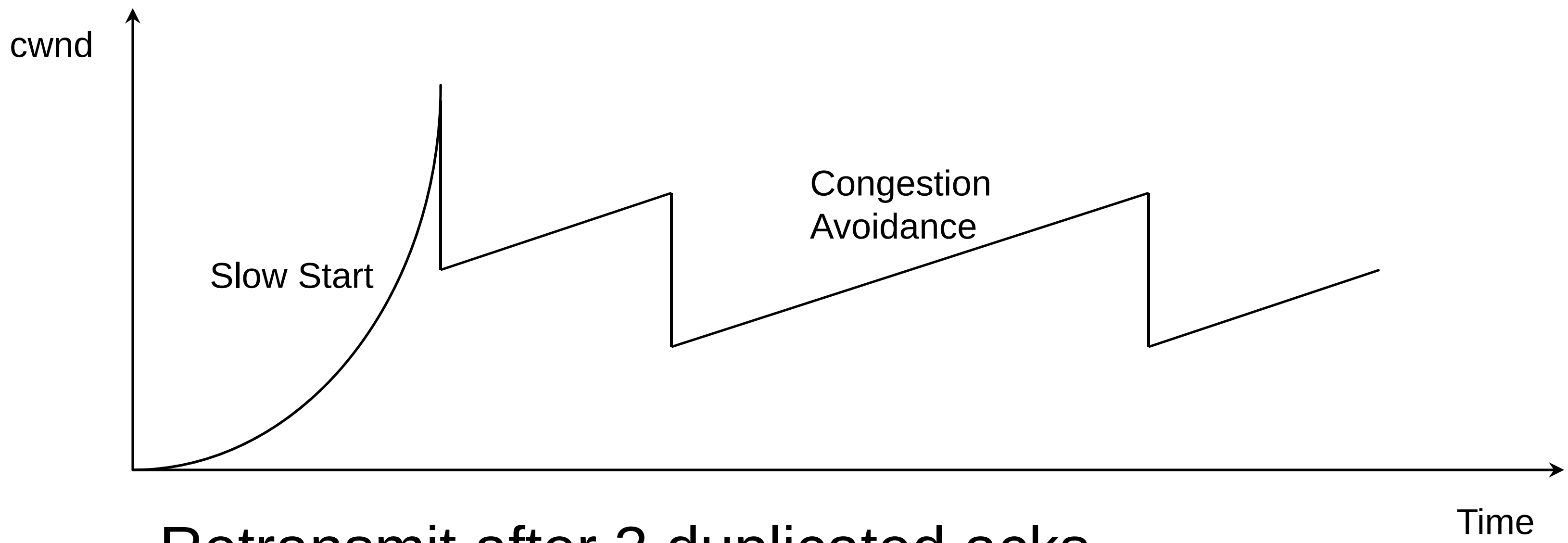


- Notes:
  - duplicate ACKs due to packet reordering!
  - if window is small don't get duplicate ACKs!

Fast Recovery

- After a fast-retransmit set *cwnd* to *ssthresh/2*
  - i.e., don't reset *cwnd* to 1
- But when RTO expires still do *cwnd* = 1
- Fast Retransmit and Fast Recovery → implemented by TCP Reno; most widely used version of TCP today

Fast Retransmit and Fast Recovery



- Retransmit after 3 duplicated acks
  - prevent expensive timeouts
- No need to slow start again
- At steady state, *cwnd* oscillates around the optimal window size.

- TCP
  - Round trip time estimation

TCP Round Trip Time and Timeout

Q: how to set TCP timeout value?

- longer than RTT
- too short: premature timeout
  - unnecessary retransmissions
- too long: slow reaction to segment loss

Q: how to estimate RTT?

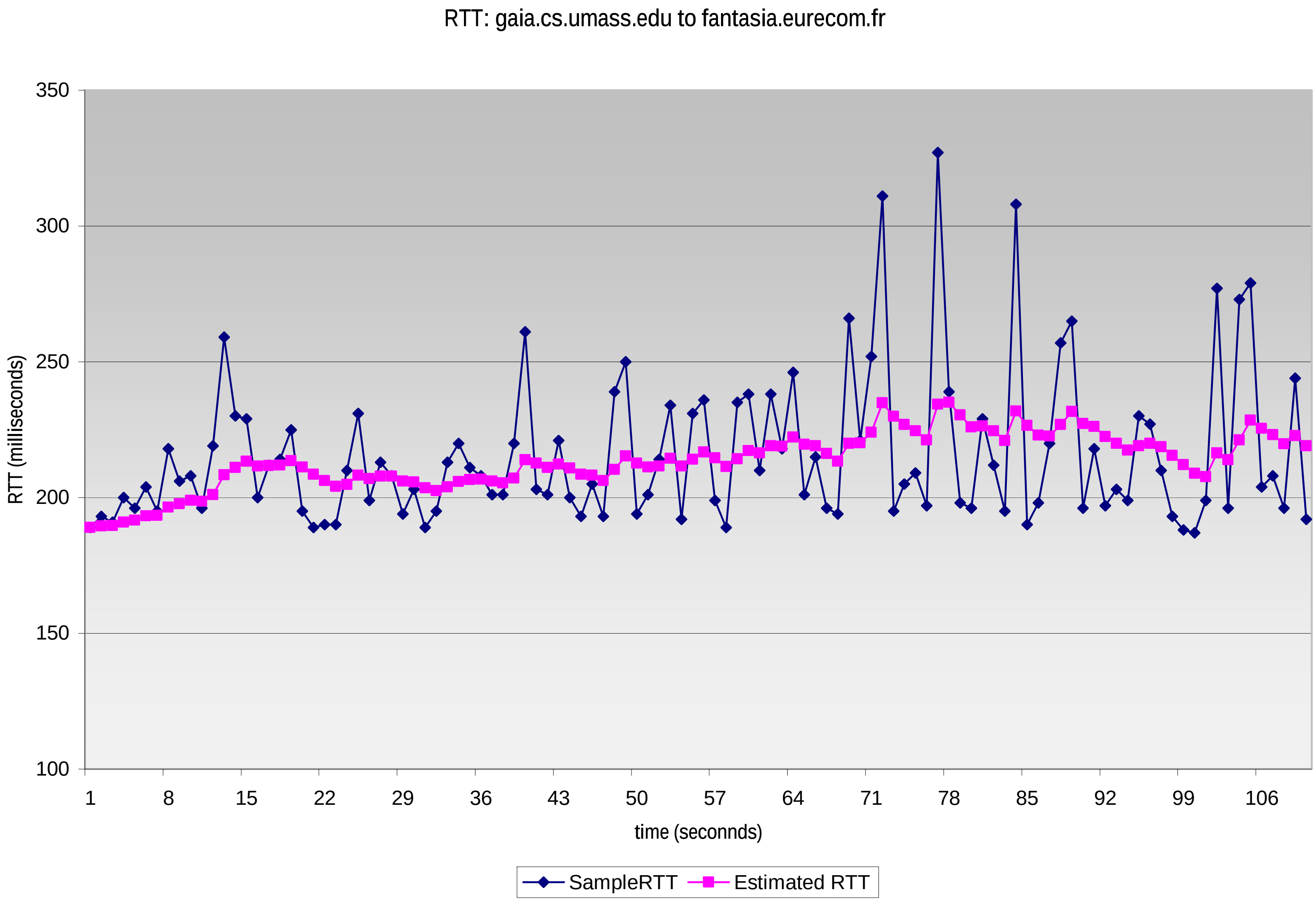
- **`SampleRTT`**: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- **`SampleRTT`** will vary, want estimated RTT “smoother”
  - average several recent measurements, not just current **`SampleRTT`**

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

- **`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
```

Congestion Control Summary

- Architecture: end system detects congestion and slow down
- Starting point:
  - slow start/congestion avoidance
    - packet drop detected by retransmission timeout RTO as congestion signal
  - fast retransmission/fast recovery
    - packet drop detected by three duplicate acks
- Router support
  - Binary feedback scheme: explicit signaling
    - Today Explicit Congestion Notification [RF99]

Reflection

- TCP implicitly assumes that all sources need to cooperate
- What are implications?