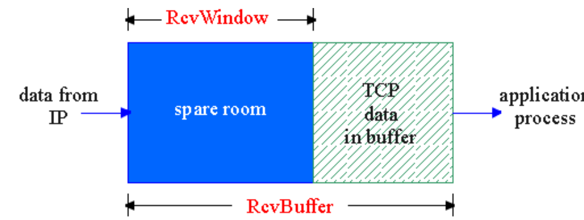
eecs489 COMPUTER NETWORKS

Lecture 30:
Flow Control, Reliable Delivery

TCP Flow Control

The receiver side of a TCP connection maintains a receiver buffer:

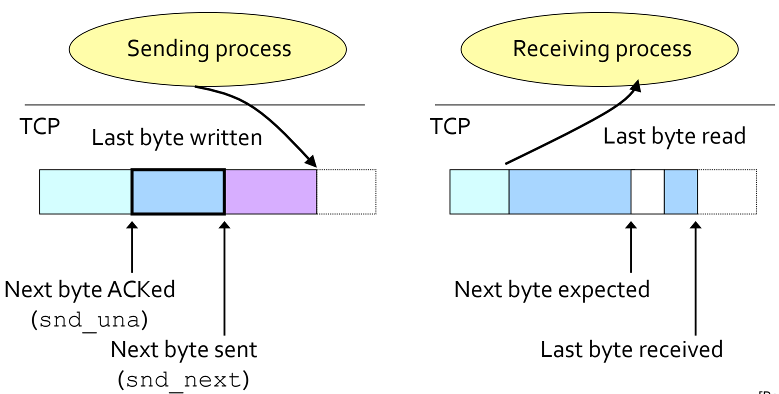


application process may be slow at reading from the buffer

Flow control ensures that sender won't overflow receiver's buffer by transmitting too much, too fast

Sliding Window

TCP uses sliding window flow control: allows a larger amount of data "in flight" than has been acknowledged
- allows sender to get ahead of the receiver
- but not too far ahead



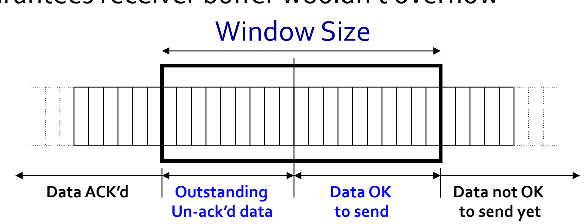
[Rexford]

TCP Receiver Window

Receiver window size (`rwnd`)
- amount that can be sent without acknowledgment
- receiver can buffer this amount of data

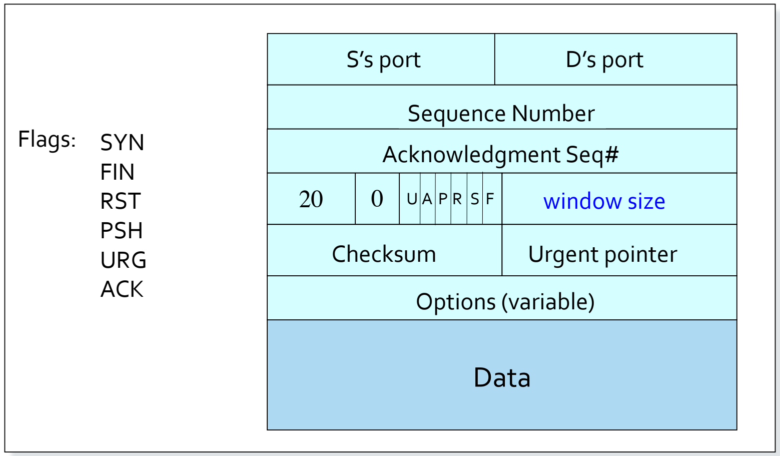
Receiver continually advertises buffer space available to sender by including the current value of `rwnd` in TCP header

Sender limits unACKed data to `rwnd`
⇒ guarantees receiver buffer wouldn't overflow



[Rexford]

TCP Header with `rwnd`



[Rexford]

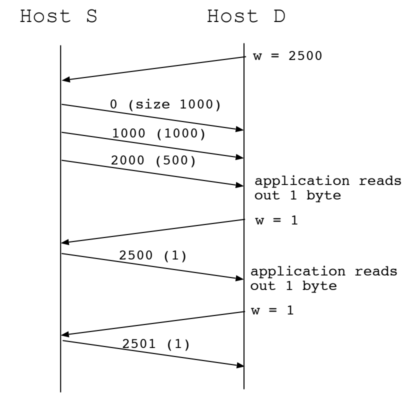
TCP Flow Control Problems

Two flow-control problems:

1. receiver too slow (silly-window syndrome)
2. sender's data comes in small amount (Nagle's algorithm)

Silly-window syndrome:

receiver window opens only by a small amount, hence sender can send only a small amount of data at a time

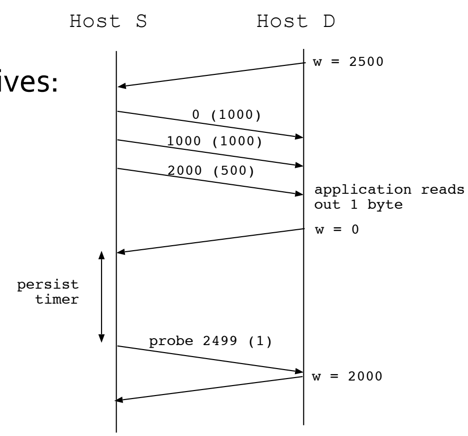


Why is this not good?

1. packet header overhead
2. small packets cause more interrupts at busy receiver

Solution to Silly-window Syndrome

Don't advertise window until it opens "significantly" (> ½*MSS or ½*`rwnd`)



Implementation alternatives:

- ACK with `rwnd`=0: sender probes after persistence timer goes off
- delayed ACK, but
  - delayed not more than 500 ms
  - or ACK every other segment

  (Why?)

Characteristics of Interactive Applications

User sends only a small amount of data, e.g., instant messaging, sends one character at a time

Problem: 40-byte header for every byte sent!

Solution: "clumping," sender clumps data together, i.e., sender waits for a "reasonable" amount of time before sending

How long is "reasonable"?

Nagle Algorithm

- send first segment immediately
- accumulate data until ACK returns, or
- up to ½ sender window or ½ MSS

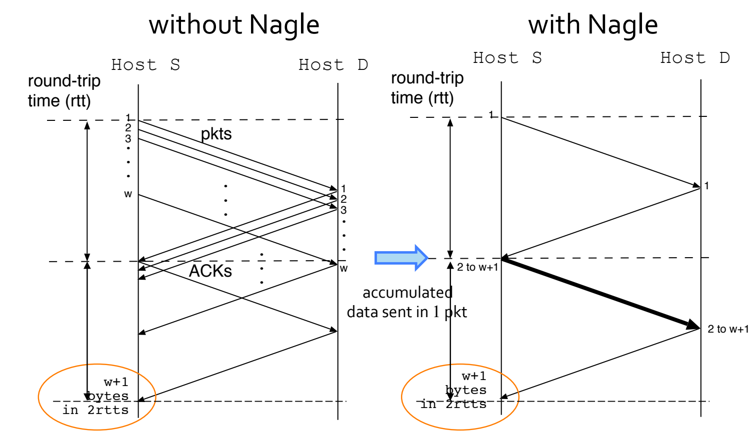
Advantages:

- bulk transfer is not held up
- data sent as fast as network can deliver (see next slide)

Can be disabled by `setsockopt(TCP_NODELAY)`

Nagle Algorithm

Nagle sends data as fast as network can deliver:



TCP Error Recovery

Sender:

- maintains only one active timer, for snd_una, restarting the timer after retransmission

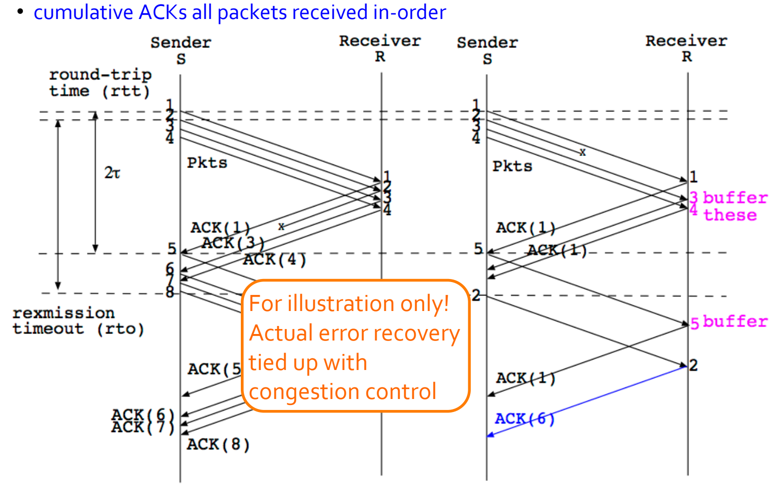
Receiver:

- cumulative ACKs all packets received in-order
- out-of-order packets repeat the last ACK
- buffers out-of-order packets
- error recovery on TCP is actually more complicated because it's tied up with congestion control, but it still relies on retransmission timeout for correctness

TCP Go-back-N with Buffering

Receiver:

- buffers out-of-order packets
- cumulative ACKs all packets received in-order



Retransmission Timeout

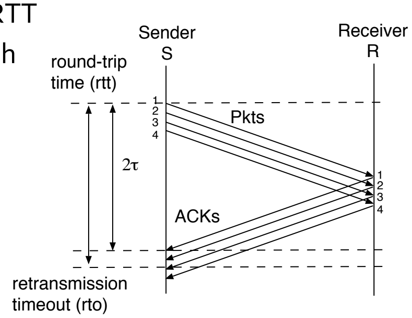
ARQ depends on retransmission to achieve reliability: sender sets a timeout waiting for an ACK

Retransmission timeout (RTO) computed from round-trip time (RTT)

- expects ACK to arrive after an RTT
- but on the Internet, RTT of a path varies over time, due to:
  - route changes
  - congestion

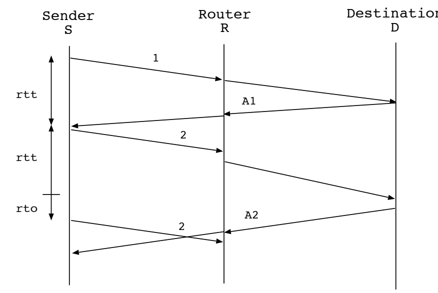
Varying RTT complicates the computation of:

1. retransmission timeout (RTO)
2. optimal sender's window size

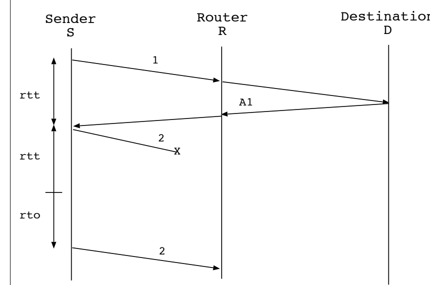


Implications of Bad RTO

RTO too small: unnecessary retransmissions:



RTO too big: lower throughput:



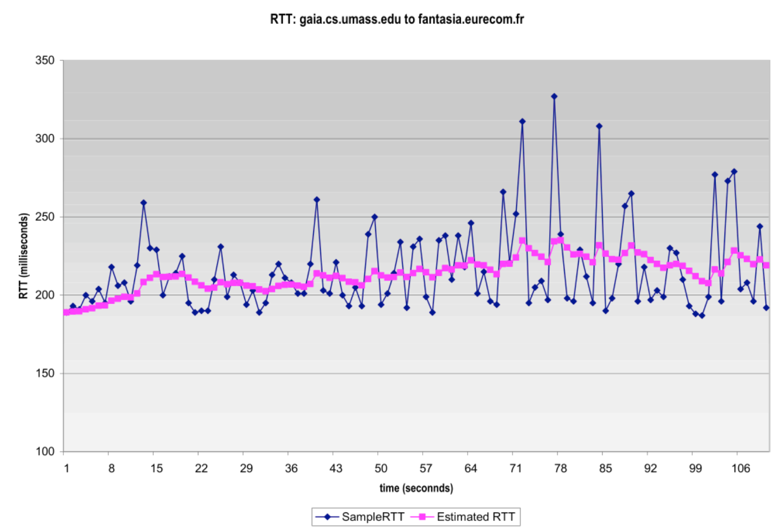
Estimating RTT

RTO must adapt to actual and current RTT

Estimate the RTT by watching returning ACKs

- compute a smoothed estimate by keeping a running average of the RTTs (a.k.a. Exponentially Weighted Moving Average (EWMA))
- estimated_RTT' $= \alpha$ * estimated_RTT $+ (1 - \alpha)$ * sample_RTT where
  - sample_RTT: time between when a segment is transmitted and when its ACK is received
  - α is the weight:
    - $\alpha \rightarrow 1$: each sample changes the estimate only a little bit
    - $\alpha \rightarrow 0$: each sample influences the estimate heavily
    - α is typically $7/8$ ($1 - 1/2^3$, which allows for fast implementation (3 right shifts))

Example RTT Estimation

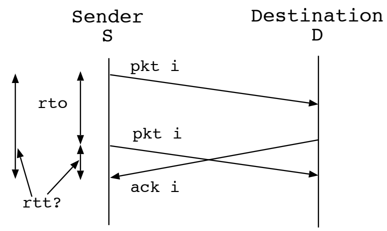


How to Compute RTO?

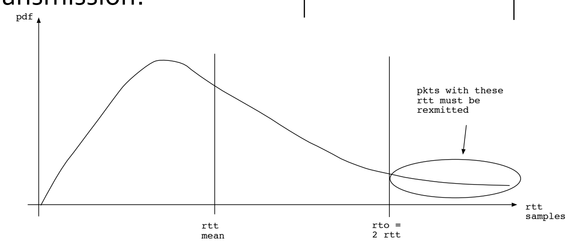
First try: RTO $= \beta$ RTT, with β typically set to 2 or 3

Two problems:

1. an ACK acknowledges receipt of data, is not an ACK for transmission: which packet to associate with an ACK in the case of retransmission?



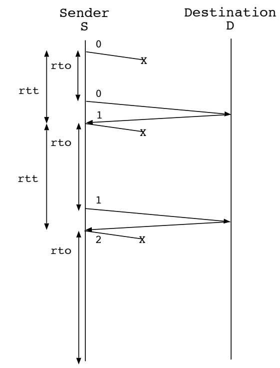
2. RTTs spread too wide



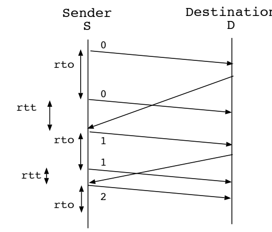
ACK Ambiguity

Which retransmitted packet to associate with an ACK?

1. original packet: RTO can grow unbounded



2. retransmitted packet: RTO shrinks



There is a feedback loop between RTO computation and RTT estimate

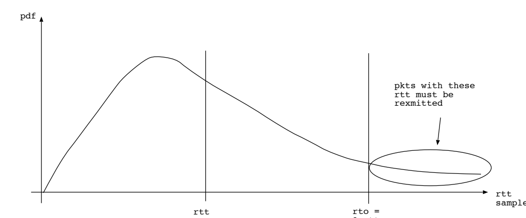
ACK Ambiguity: Karn's Algorithm

Karn's algorithm:

- adjust RTT estimate only from non-retransmitted samples
- however, ignoring retransmissions could lead to insensitivity to long delays
- so, back off RTO upon retransmission: $\text{RTO}_{\text{new}} = \gamma\, \text{RTO}_{\text{old}}$, γ typically $= 2$

RTT Spread Too Wide

RTT estimate computed using EWMA only considers the mean, doesn't take variance into account



Jacobson's algorithm:

- estimate deviation (D) of sample_RTT
  - $D_{\text{new}} = \alpha D_{\text{old}} + (1 - \alpha)\,|\text{sample\_RTT} - \text{estimated\_RTT}|$
- compute new estimated_RTT as usual
- take the deviation in sample_RTT (D) into account when computing RTO
  - RTO = estimated_RTT + $4D$

Timers Used in TCP

1. `TIME_WAIT`: 2*MSL
2. persistence timer
3. RTO
4. keep-alive timer: probe the other side if connection has been idle for "too long"
   - may be turned on/off
   - idle period may be set using `setsockopt()`