CSE 461: TCP (part 1)

Ben Greenstein
Jeremy Elson

TA: Ivan Beschastnikh

Thanks to Tom Anderson and Ratul Mahajan for slides

Administrivia

Homework #3

- Optional, but at least 1-2 of these problems will serve as a model for a midterm exam question:
  Ch. 3: 15, 19. Ch. 4: 3, 10, 12, 13, 22

Midterm is on Wednesday, November 5

- HW1 and HW2 and Project 1 returned by Friday
- Covers all lectures and related text up to and including 4.2
- Today's and Monday's lectures will be on Final
- Short midterm review on Monday

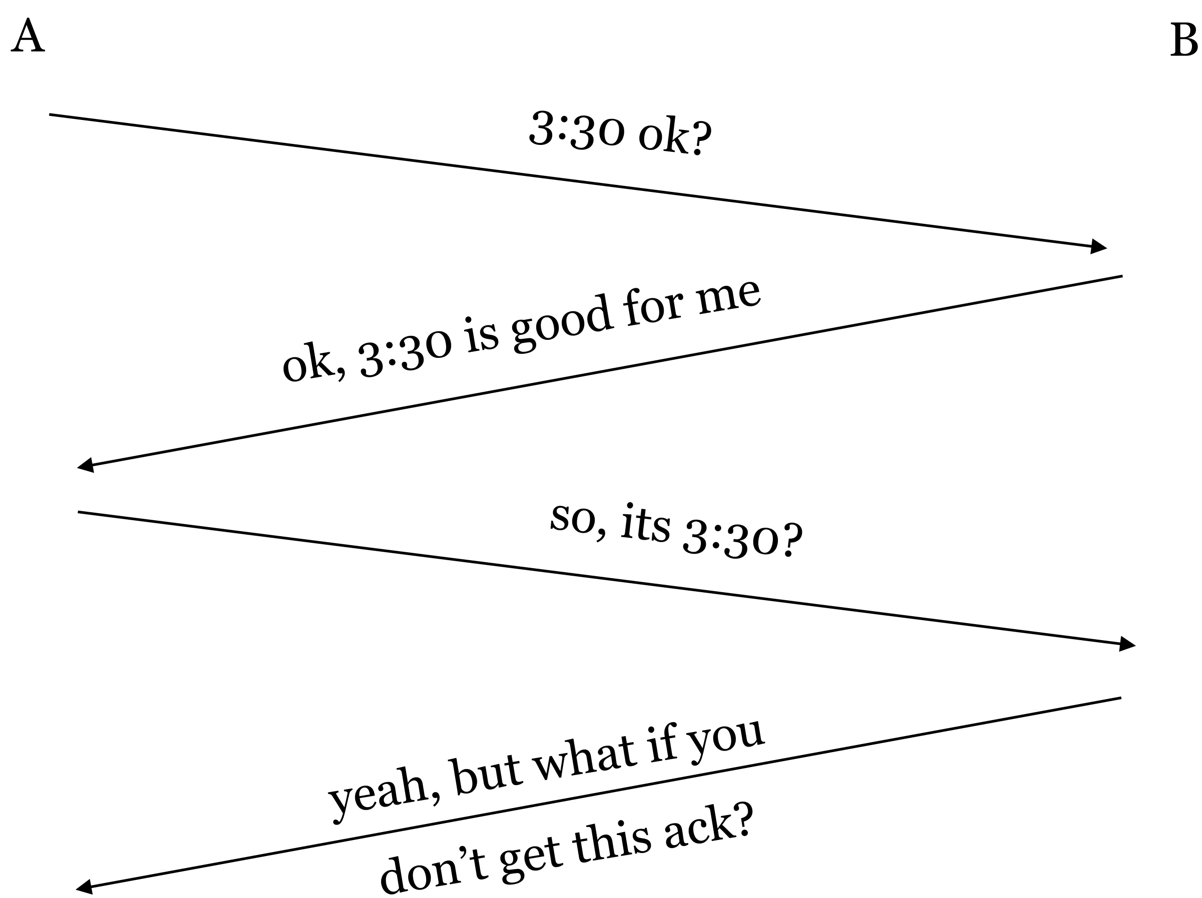
Project 2, part 2 due Friday, November 7

General's Paradox

Can we use messages and retries to synchronize two machines so they are guaranteed to do some operation at the same time?

- No. Why?

General's Paradox Illustrated



Consensus revisited

If distributed consensus is impossible, what then?

TCP can agree that destination received data

Transport Challenge

IP: routers can be arbitrarily bad

- packets can be lost, reordered, duplicated, have limited size & can be fragmented

TCP: applications need something better

- Reliable delivery, in order delivery, no duplicates, arbitrarily long streams of data, match sender/receiver speed, process-to-process

Reliable Transmission

How do we send packets reliably?

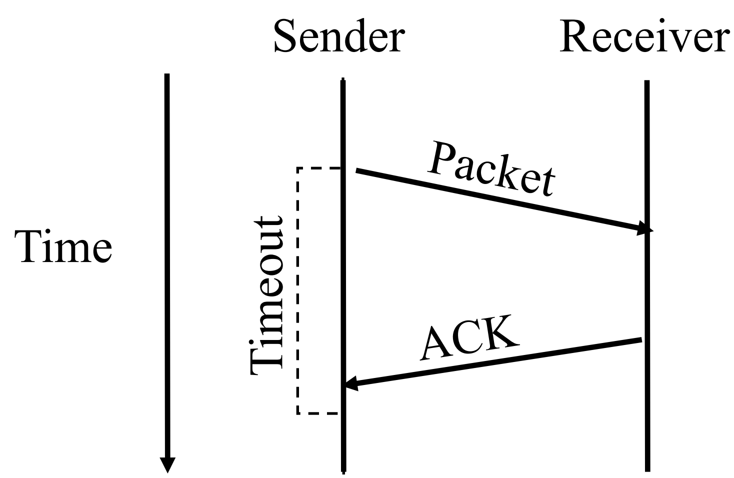
Two mechanisms

- Acknowledgements
- Timeouts

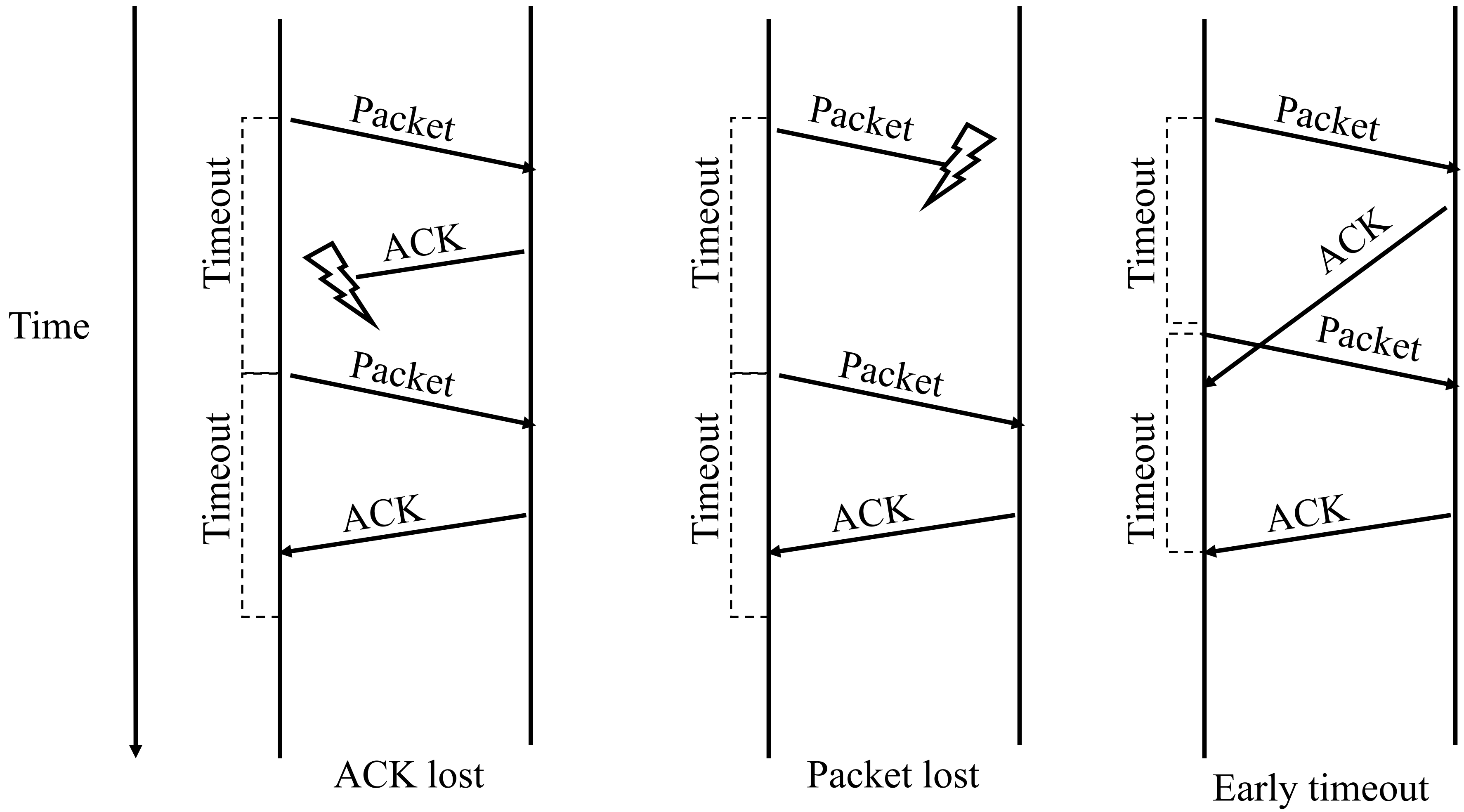
Simplest reliable protocol: Stop and Wait

Stop and Wait

- Send a packet, wait until ack arrives
  - retransmit if no ack within timeout
- Receiver acks each packet as it arrives



Recovering from error



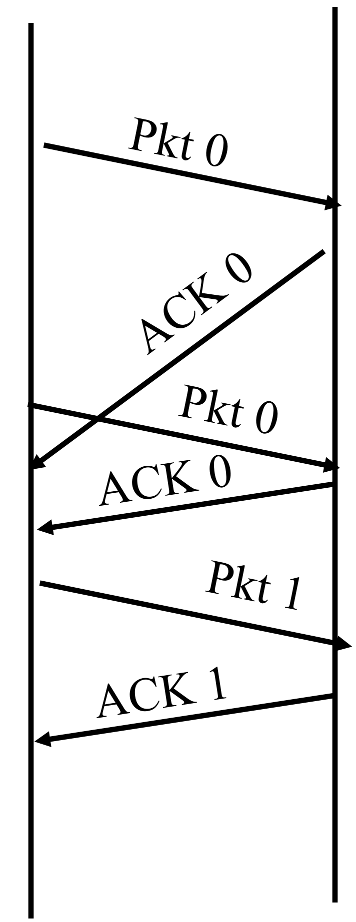
How can we recognize resends?

Use unique ID for each pkt

- for both packets and acks

How many bits for the ID?

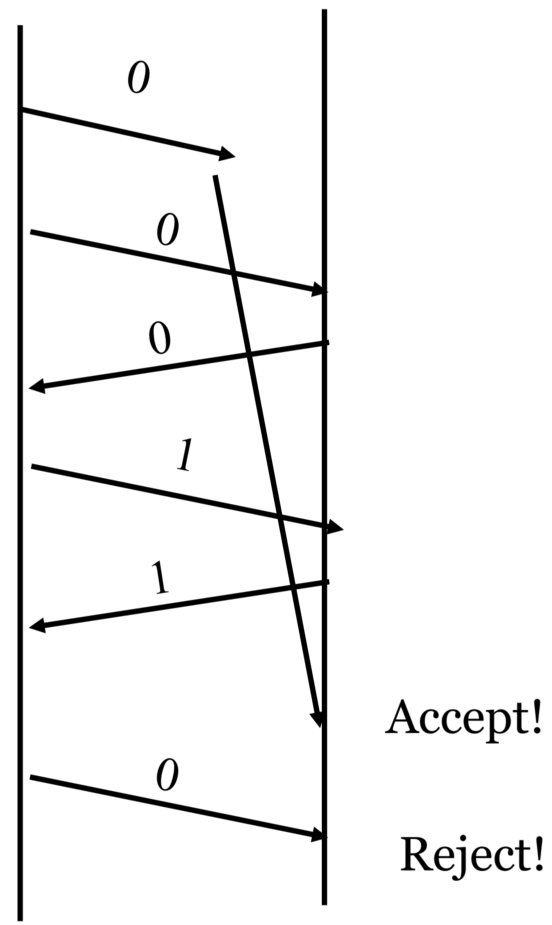
- For stop and wait, a single bit!
- assuming in-order delivery…



What if packets can be delayed?

Solutions?

- Never reuse an ID?
- Change IP layer to eliminate packet reordering?
- Prevent very late delivery?
  - IP routers keep hop count per pkt, discard if exceeded
  - ID's not reused within delay bound
- TCP won't work without some bound on how late packets can arrive!



What happens on reboot?

How do we distinguish packets sent before and after reboot?

- Can’t remember last sequence # used unless written to stable storage (disk or NVRAM)

Solutions?

- Restart sequence # at 0?
- Assume/force boot to take max packet delay?
- Include epoch number in packet (stored on disk)?
- Ask other side what the last sequence # was?

- TCP sidesteps this problem with random initial seq # (in each direction)

How do we keep the pipe full?

Unless the bandwidth*delay product is small, stop and wait can't fill pipe

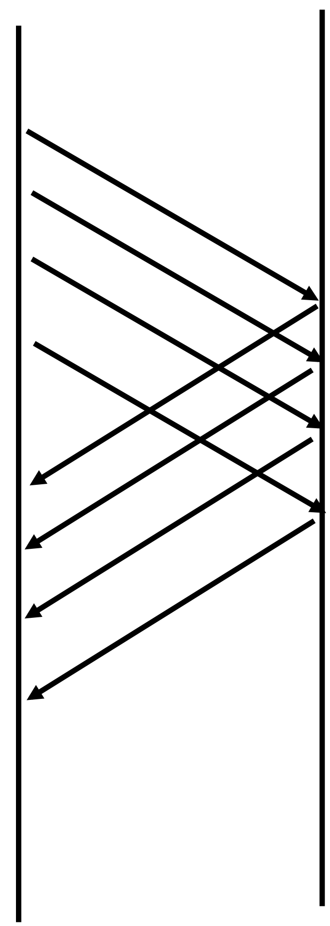
Solution: Send multiple packets without waiting for first to be acked

Reliable, unordered delivery:

- Send new packet after each ack
- Sender keeps list of unack'ed packets; resends after timeout
- Receiver same as stop&wait

How easy is it to write apps that handle out of order delivery?

- How easy is it to test those apps?



Sliding Window: Reliable, ordered delivery

Two constraints:

- Receiver can't deliver packet to application until all prior packets have arrived
- Sender must prevent buffer overflow at receiver

Solution: sliding window

- circular buffer at sender and receiver
  - packets in transit <= buffer size
  - advance when sender and receiver agree packets at beginning have been received
- How big should the window be?
  - bandwidth * round trip delay

Sender/Receiver State

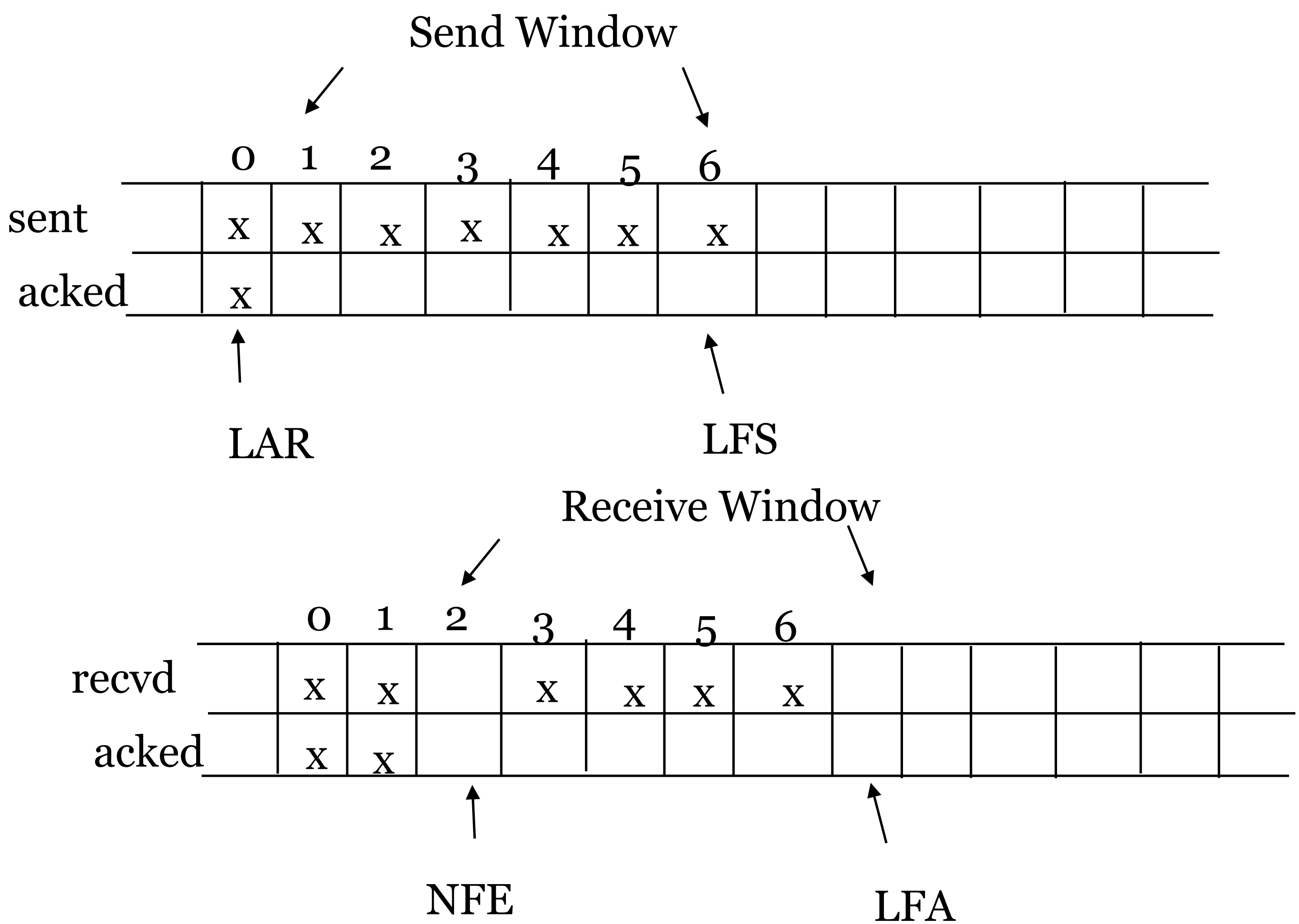
sender

- packets sent and acked (LAR = last ack recvd)
- packets sent but not yet acked
- packets not yet sent (LFS = last frame sent)

receiver

- packets received and acked (NFE = next frame expected)
- packets received out of order
- packets not yet received (LFA = last frame ok)

Sliding Window



What if we lose a packet?

Go back N (original TCP)

- receiver acks “got up through k” (“cumulative ack”)
- ok for receiver to buffer out of order packets
- on timeout, sender restarts from k+1

Selective retransmission (RFC 2018)

- receiver sends ack for each pkt in window
- on timeout, resend only missing packet

Can we shortcut timeout?

If packets usually arrive in order, out of order delivery is (probably) a packet loss

- Negative ack
  - receiver requests missing packet
- Fast retransmit (TCP)
  - receiver acks with NFE-1 (or selective ack)
  - if sender gets acks that don't advance NFE, resends missing packet

Sender Algorithm

Send full window, set timeout

On receiving an ack:

- if it increases LAR (last ack received)
    - send next packet(s)
        - -- no more than window size outstanding at once
- else (already received this ack)
    - if receive multiple acks for LAR, next packet may have been lost; retransmit LAR + 1 (and more if selective ack)

On timeout:

- resend LAR + 1 (first packet not yet acked)

Receiver Algorithm

On packet arrival:

- if packet is the NFE (next frame expected)
  - send ack
  - increase NFE
  - hand any packet(s) below NFE to application
- else if < NFE (packet already seen and acked)
  - send ack and discard // Q: why is ack needed?
- else (packet is > NFE, arrived out of order)
  - buffer and send ack for NFE – 1
    - -- signal sender that NFE might have been lost
    - -- and with selective ack: which packets correctly arrived

What if link is very lossy?

Wireless packet loss rates can be 10-30%

- end to end retransmission will still work
- will be inefficient, especially with go back N

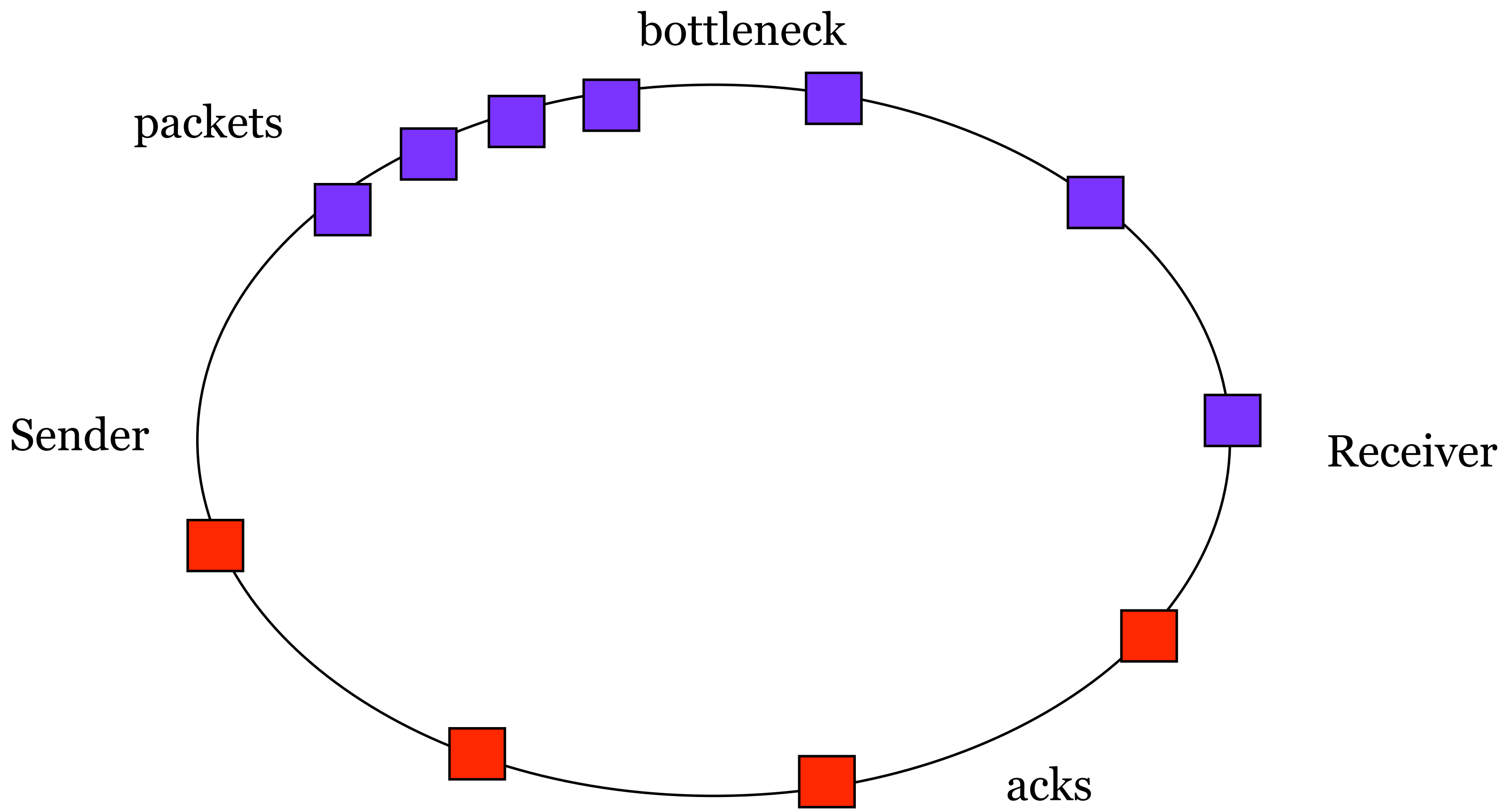
Solution: hop by hop retransmission

- performance optimization, not for correctness

End to end principle

- ok to do optimizations at lower layer
- still need end to end retransmission; why?

Avoiding burstiness: ack pacing



Window size = round trip delay * bit rate

How many sequence #'s?

Window size + 1?

- Suppose window size = 3
- Sequence space: 0 1 2 3 0 1 2 3
- send 0 1 2, all arrive
  - if acks are lost, resend 0 1 2
  - if acks arrive, send new 3 0 1

Window <= (max seq # + 1) / 2

How do we determine timeouts?

If timeout too small, useless retransmits

- can lead to congestion collapse (and did in 86)
- as load increases, longer delays, more timeouts, more retransmissions, more load, longer delays, more timeouts …
- Dynamic instability!

If timeout too big, inefficient

- wait too long to send missing packet

Timeout should be based on actual round trip time (RTT)

- varies with destination subnet, routing changes, congestion, …

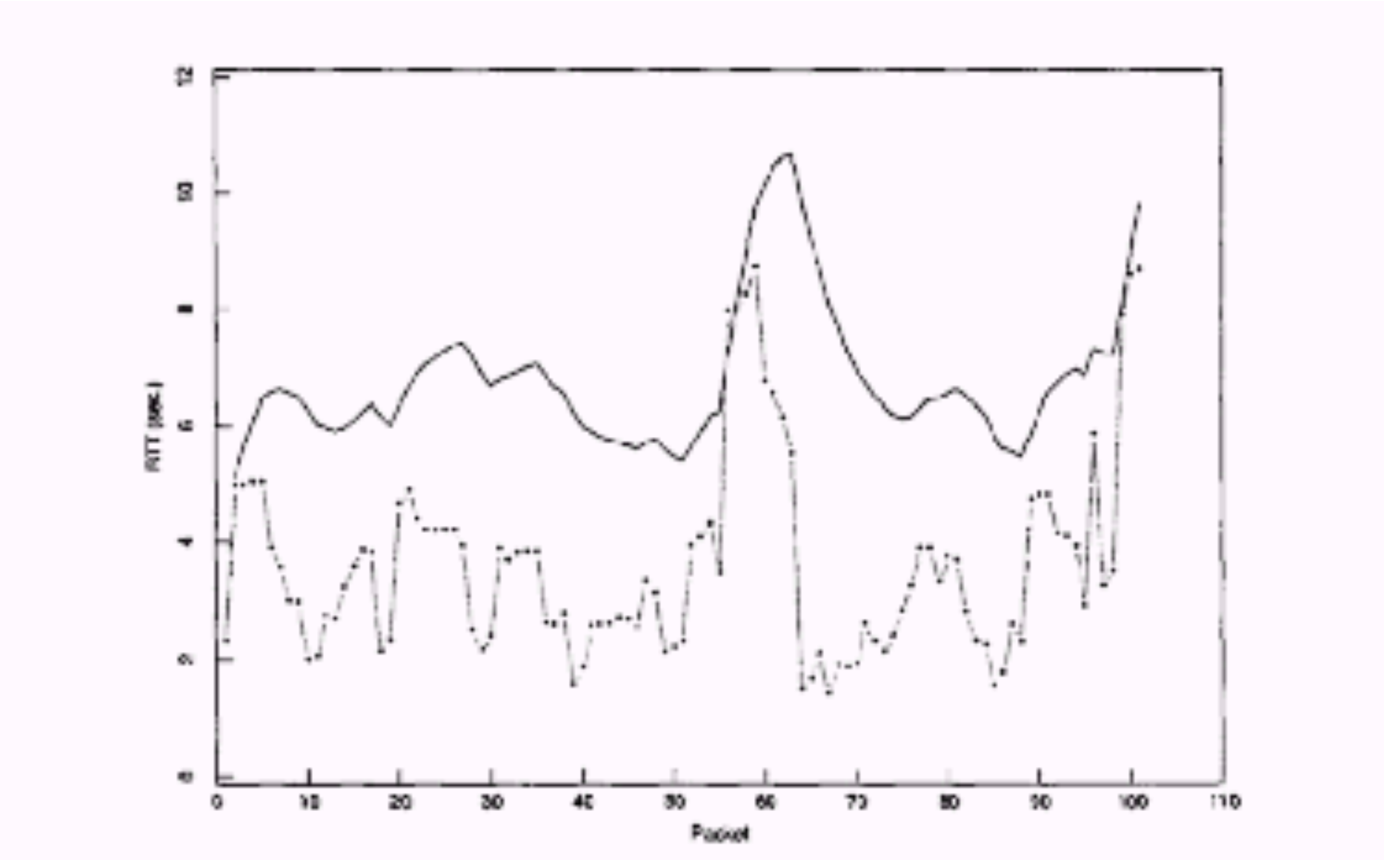
Estimating RTTs

Idea: Adapt based on recent past measurements

- For each packet, note time sent and time ack received
- Compute RTT samples and average recent samples for timeout
- EstimatedRTT = α x EstimatedRTT + (1 - α) x SampleRTT

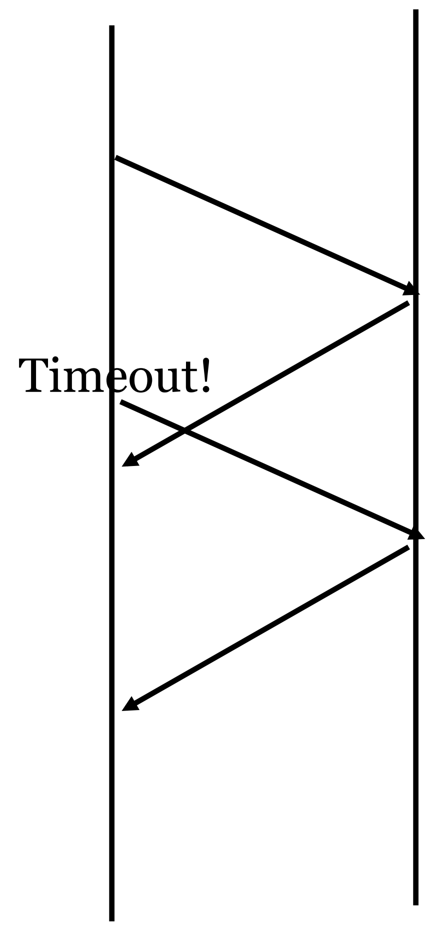
- This is an exponentially-weighted moving average (low pass filter) that smoothes the samples. Typically, α = 0.8 to 0.9.
- Set timeout to small multiple (2) of the estimate

Estimated Retransmit Timer



Retransmission ambiguity

- How do we distinguish first ack from retransmitted ack?
  - First send to first ack?
    - What if ack dropped?
  - Last send to last ack?
    - What if last ack dropped?
- Might never be able to fix too short a timeout!



Retransmission ambiguity: Solutions?

TCP: Karn-Partridge

- ignore RTT estimates for retransmitted pkts
- double timeout on every retransmission

Add sequence #'s to retransmissions (retry #1, retry #2, ...)

Modern TCP (RFC 1323): Add timestamp into packet header; ack returns timestamp

Jacobson/Karels Algorithm

Problem:

- Variance in RTTs gets large as network gets loaded
- Average RTT isn't a good predictor when we need it most

Solution: Track variance too.

- Difference = SampleRTT – EstimatedRTT
- EstimatedRTT = EstimatedRTT + (δ x Difference)
- Deviation = Deviation + δ(|Difference|- Deviation)
- Timeout = μ x EstimatedRTT + ϕ x Deviation
- In practice, δ = 1/8, μ = 1 and ϕ = 4

Estimate with Mean + Variance