CSE 461: TCP (part 3)

Ben Greenstein
Jeremy Elson

TA: Ivan Beschastnikh

Thanks Tom Anderson and Ratul Mahajan for slides

Administrivia

Project 2

- Part 3: Due Friday, 11/14, 11:59pm

Avoiding Small Packets

Nagle's algorithm (sender side):

- Only allow one outstanding segment smaller than the MSS
- A "self-clocking" algorithm
- But gets in the way for SSH etc. (TCP_NODELAY)

Delayed acknowledgements (receiver side)

- Wait to send ACK, hoping to piggyback on reverse stream
- But send one ACK per two data packets and use timeout on the delay
- Cuts down on overheads and allows coalescing
- Otherwise a nuisance, e.g, RTT estimation

Irony: how do Nagle and delayed ACKs interact?

- Consider a Web request

Bandwidth Allocation

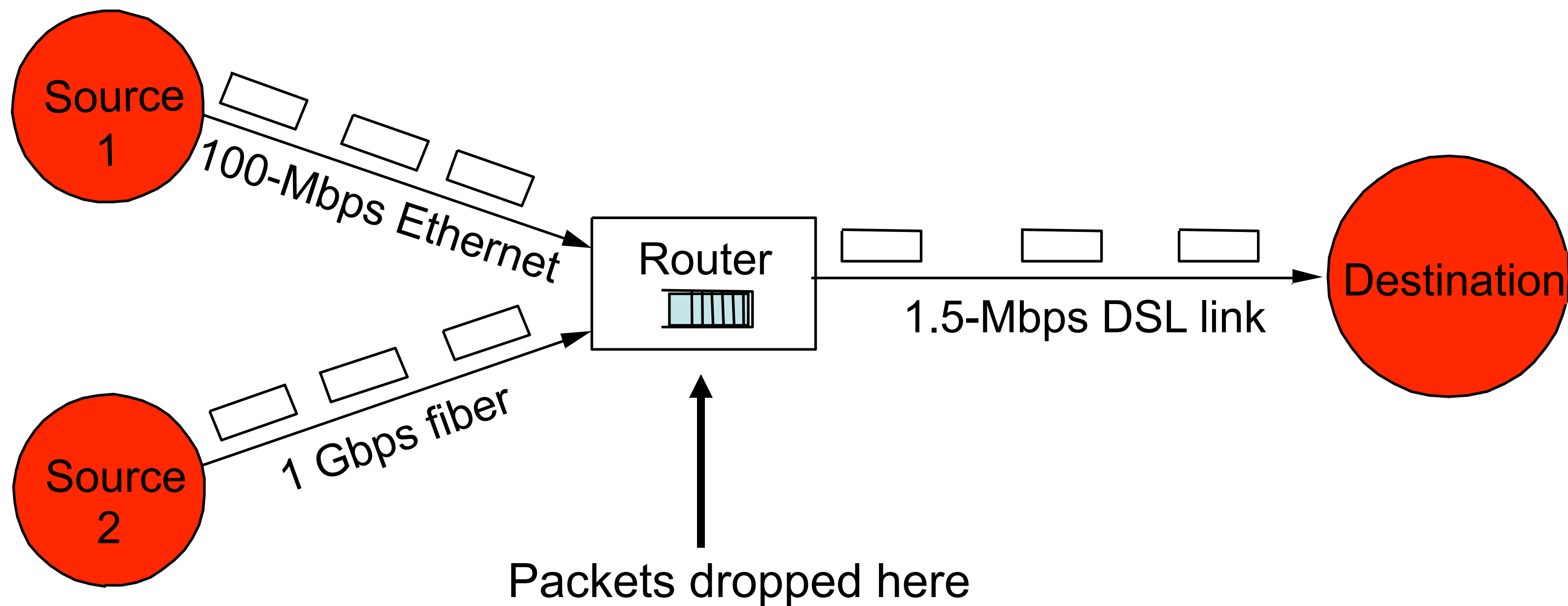
How fast should a host, e.g., a web server, send packets?

Two considerations:

- Congestion:
  - sending too fast will cause packets to be lost in the network
- Fairness:
  - different users should get their fair share of the bandwidth

Often treated together (e.g. TCP) but needn’t be.

Congestion



Buffer absorbs bursts when input rate > output

If sending rate is persistently > drain rate, queue builds

Dropped packets represent wasted work

Evaluating Congestion Control

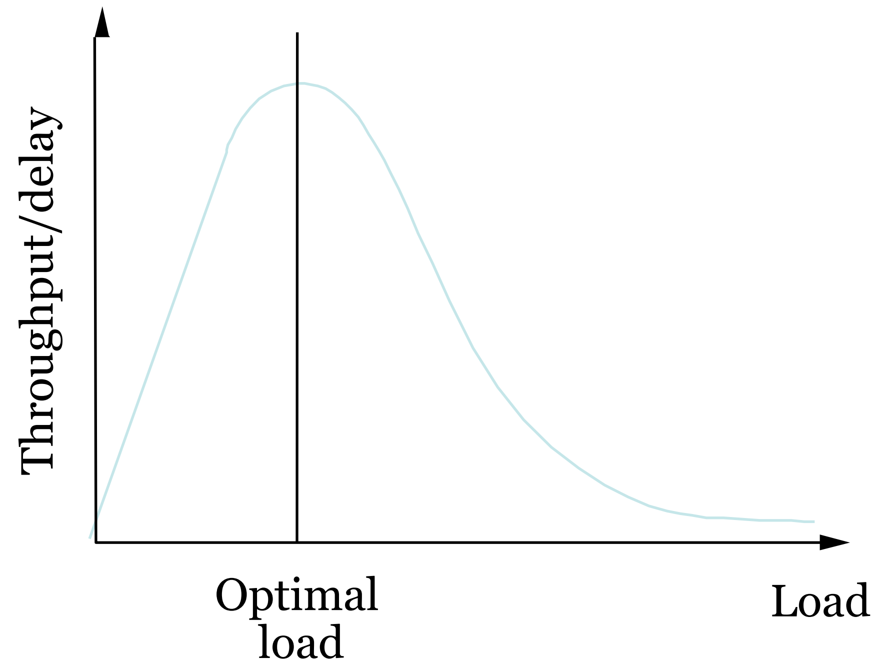
Power = throughput / delay

At low load, throughput goes up and delay remains small

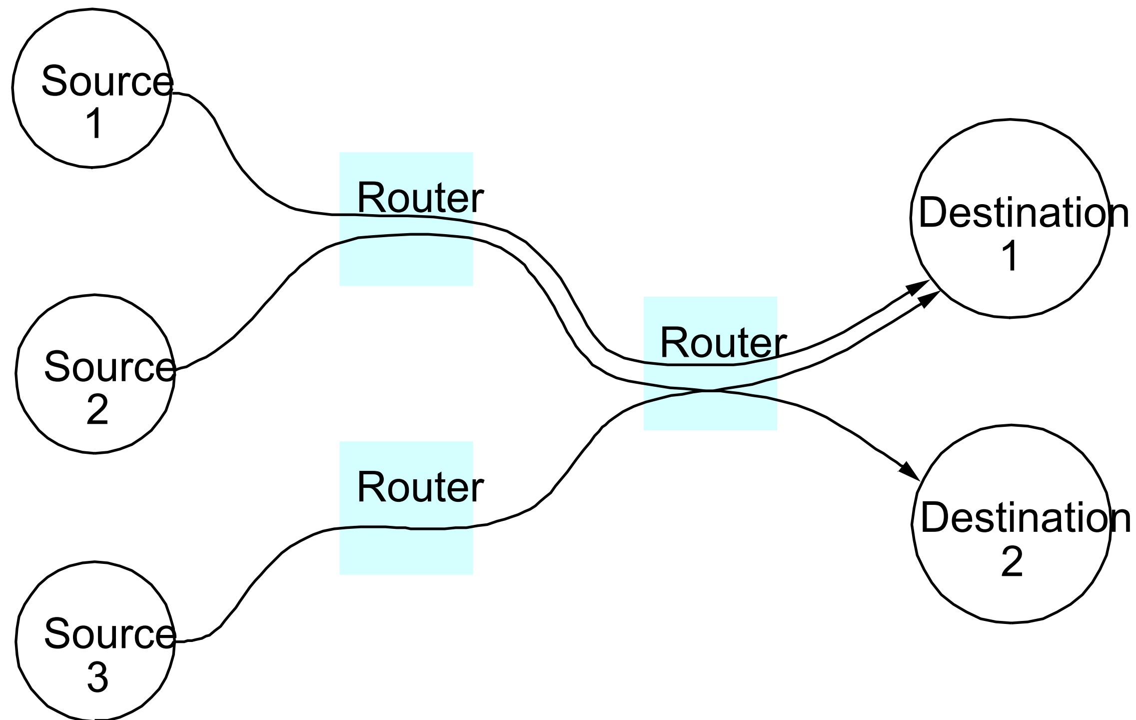
At moderate load, delay is increasing (queues) but throughput doesn't grow much

At high load, much loss and delay increases greatly due to retransmissions

Even worse, can oscillate!



Fairness



Each flow from a source to a destination should (?) get an equal share of the bottleneck link … depends on paths and other traffic

Chapter 6, Figure 2

Evaluating Fairness

First, need to define what is a fair allocation.

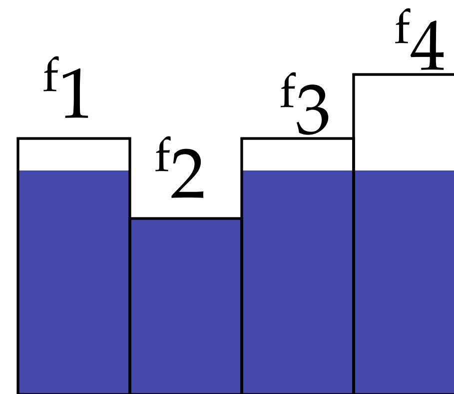
- Consider n flows, each wants a fraction f_i of the bandwidth

Min-max fairness:

- First satisfy all flows evenly up to the lowest f_i. Repeat with the remaining bandwidth.

Or proportional fairness

- Depends on path length …



Why is bandwidth allocation hard?

Given network and traffic, just work out fair share and tell the sources …

But:

- Demands come from many sources
- Needed information isn't in the right place
- Demands are changing rapidly over time
- Information is out-of-date by the time it's conveyed
- Network paths are changing over time

Designs affect Network services

TCP/Internet provides “best-effort” service

- Implicit network feedback, host controls via window.
- No strong notions of fairness

A network in which there are QOS (quality of service) guarantees

- Rate-based reservations natural choice for some apps
- But reservations are need a good characterization of traffic
- Network involvement typically needed to provide a guarantee

Former tends to be simpler to build, latter offers greater service to applications but is more complex.

Case Study: TCP

The dominant means of bandwidth allocation today

Internet meltdowns in the late 80s ("congestion collapse") led to much of its mechanism

- Jacobson's slow-start, congestion avoidance [sic], fast retransmit and fast recovery.

Main constraint was zero network support and de facto backwards-compatible upgrades to the sender

- Infer packet loss and use it as a proxy for congestion

We will look at other models later …

TCP Before Congestion Control

- Just use a fixed size sliding window!
  - Will under-utilize the network or cause unnecessary loss
- Congestion control dynamically varies the size of the window to match sending and available bandwidth
  - Sliding window uses minimum of cwnd, the congestion window, and the advertised flow control window
- The big question: how do we decide what size the window should be?

TCP Congestion Control

Goal: efficiently and fairly allocate network bandwidth

- Robust RTT estimation
- Additive increase/multiplicative decrease
  - oscillate around bottleneck capacity
- Slow start
  - quickly identify bottleneck capacity
- Fast retransmit
- Fast recovery

Tracking the Bottleneck Bandwidth

Sending rate = window size/RTT

Multiplicative decrease

- Timeout => dropped packet => sending too fast => cut window size in half
  - and therefore cut sending rate in half

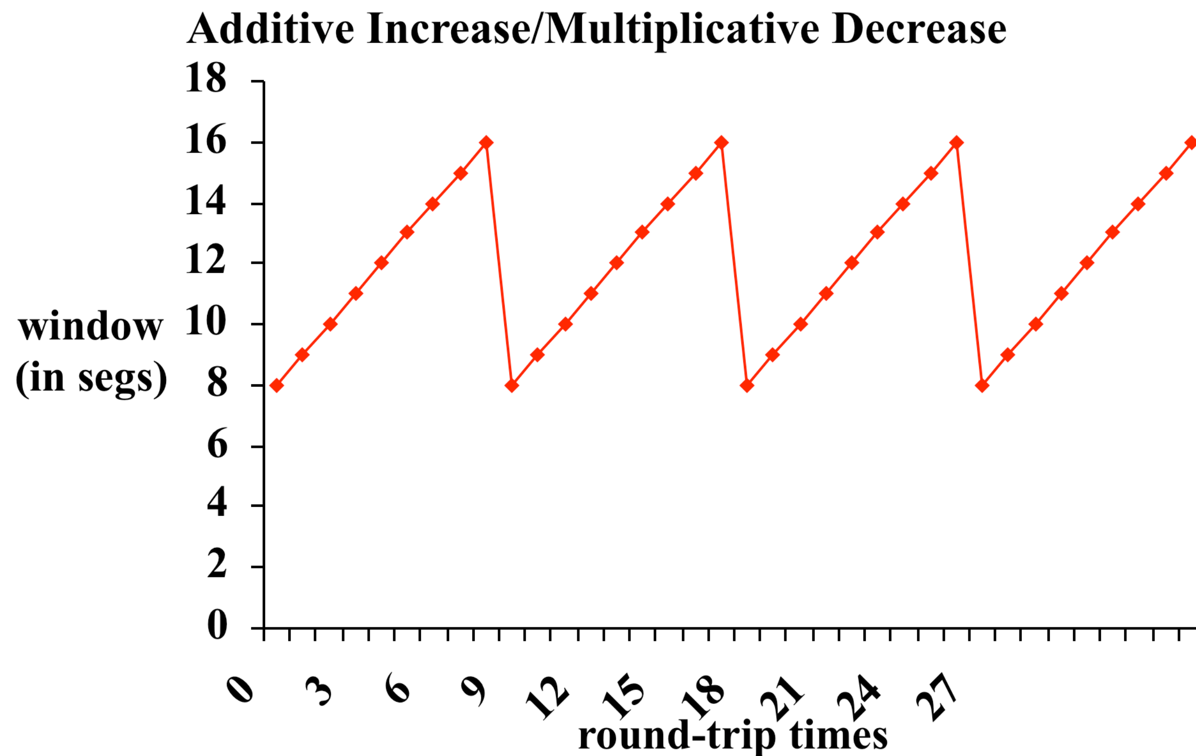
Additive increase

- Ack arrives => no drop => sending too slow => increase window size by one packet/window
  - and therefore increase sending rate a little

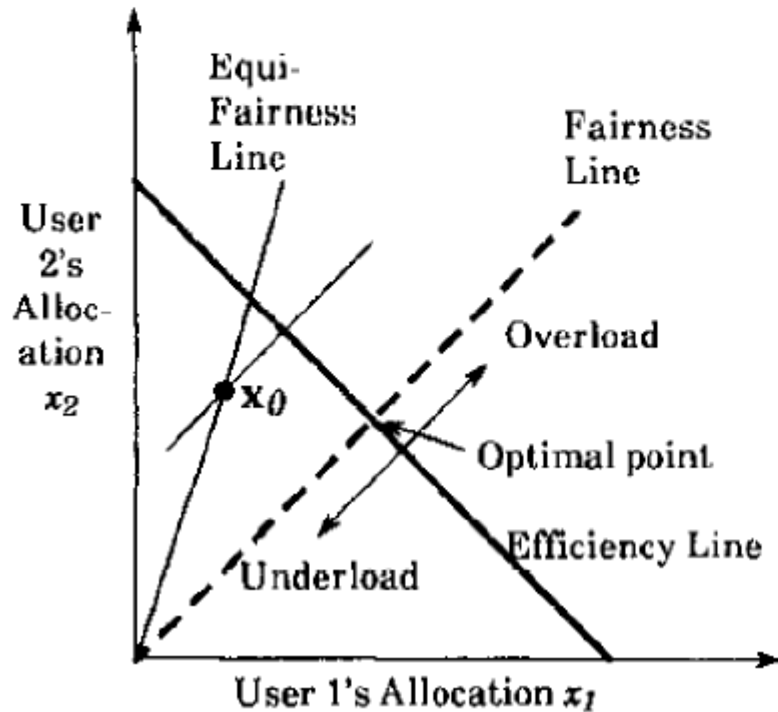
TCP "Sawtooth"

Oscillates around bottleneck bandwidth

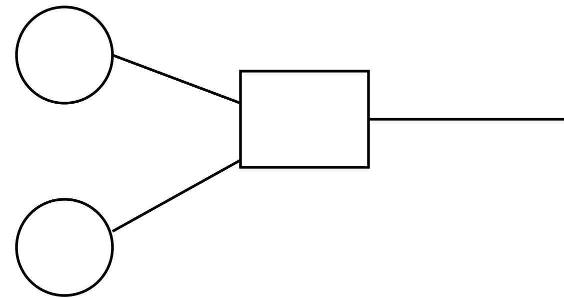
- adjusts to changes in competing traffic



Why AIMD?



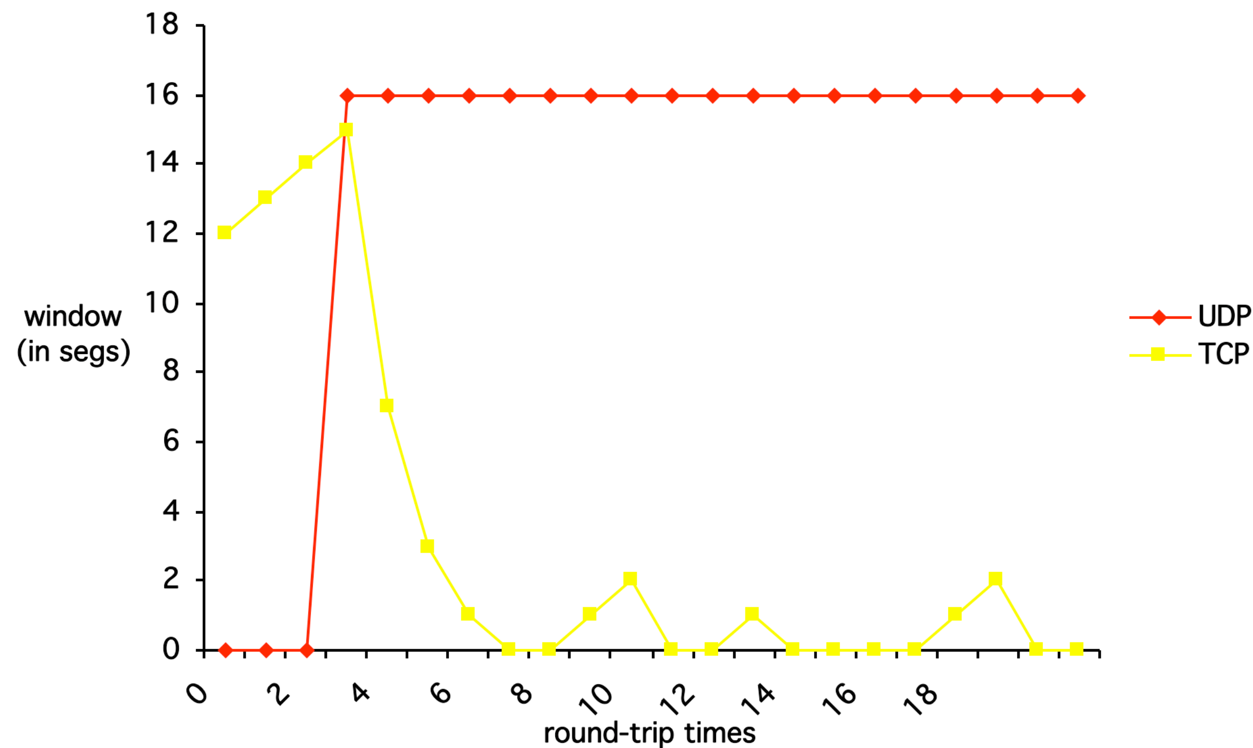
Two users competing for bandwidth:



Consider the sequence of moves from AIMD, AIAD, MIMD, MIAD.

What if TCP and UDP share link?

Independent of initial rates, UDP will get priority! TCP will take what's left.



What if two different TCP implementations share link?

If cut back more slowly after drops => will grab bigger share

If add more quickly after acks => will grab bigger share

Incentive to cause congestion collapse!

- Many TCP “accelerators”
- Easy to improve perf at expense of network

One solution: enforce good behavior at router

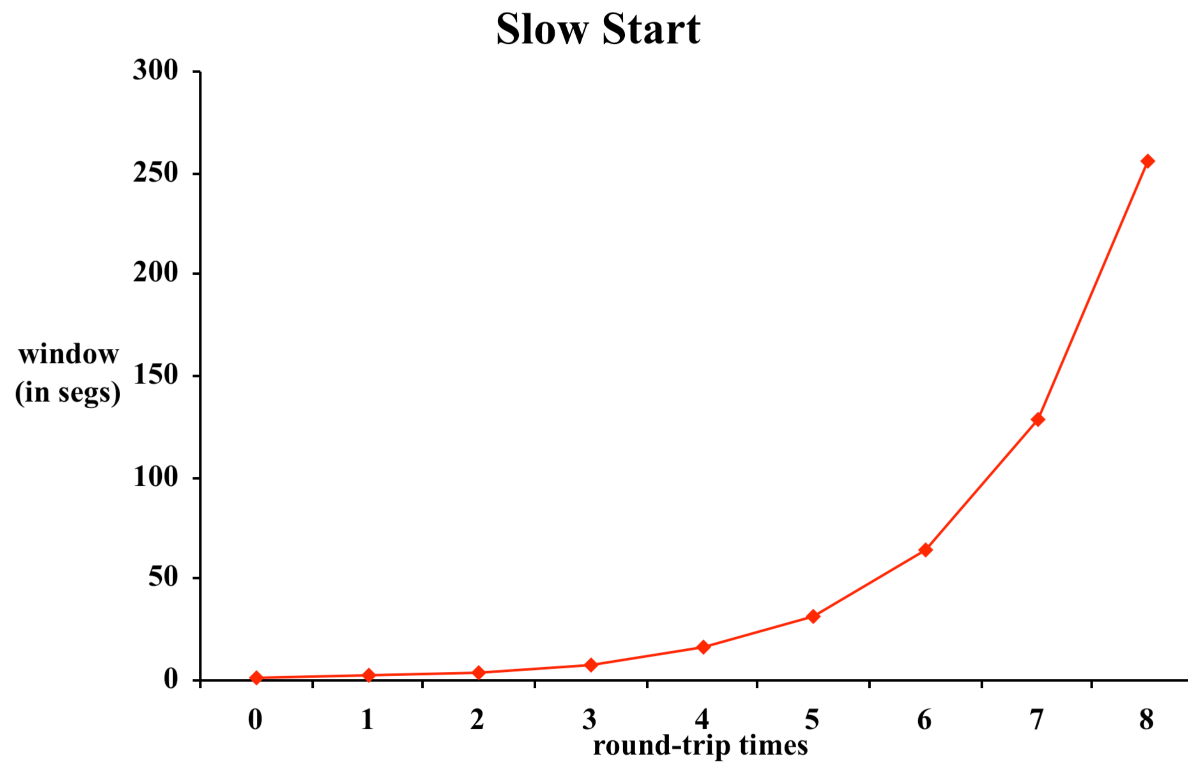
Slow start

How do we find bottleneck bandwidth?

- Start by sending a single packet
  - start slow to avoid overwhelming network
- Multiplicative increase until get packet loss
  - quickly find bottleneck
- Remember previous max window size
  - shift into linear increase/multiplicative decrease when get close to previous max ~ bottleneck rate
  - called "congestion avoidance"

Slow Start

Quickly find the bottleneck bandwidth



TCP Mechanics Illustrated

Source

Router

Dest

100 Mbps

0.9 ms latency

10 Mbps

0 latency

Slow Start vs. Delayed Acks

Recall that acks are delayed by 200ms to wait for application to provide data

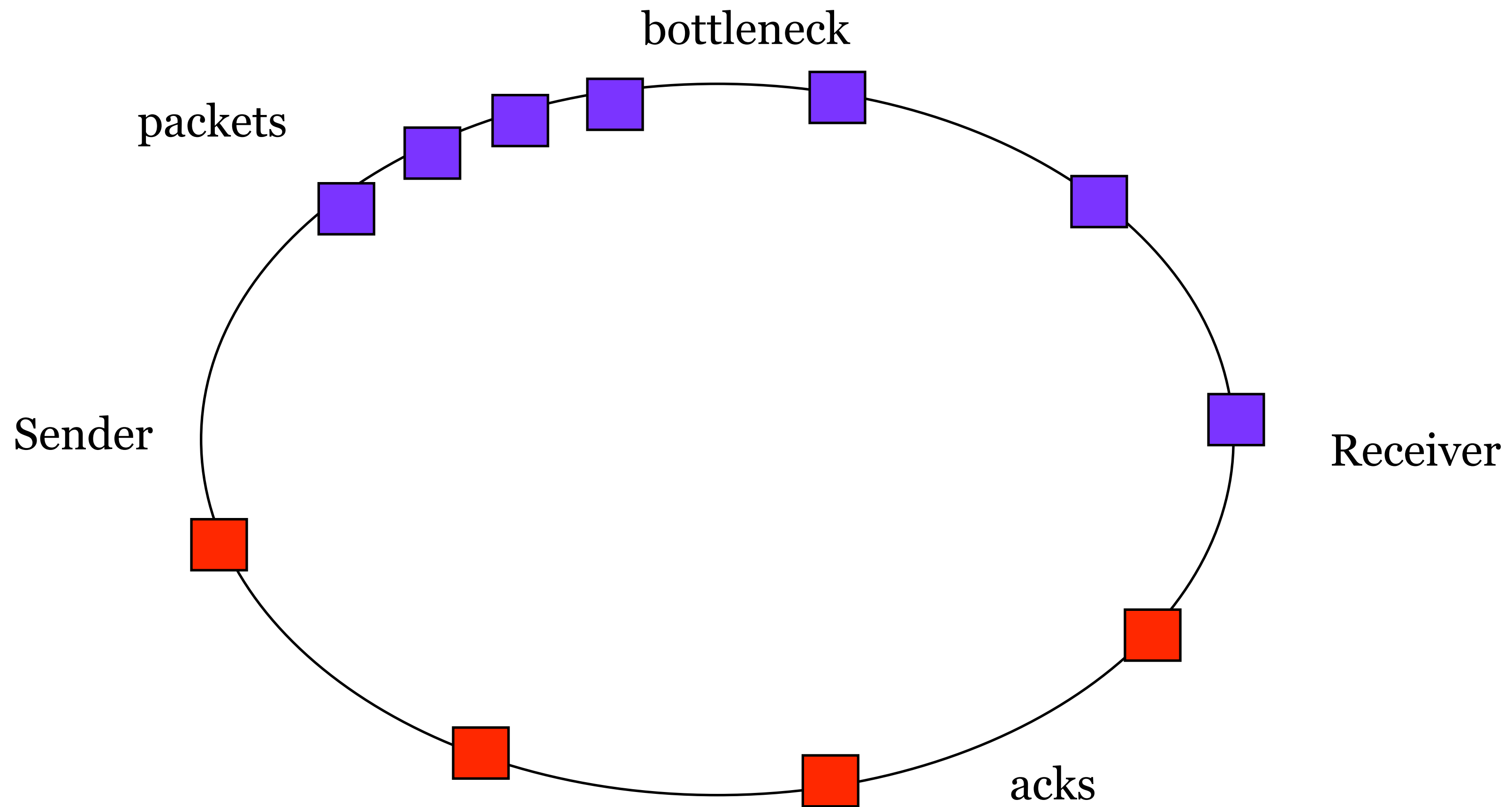
But (!) TCP congestion control triggered by acks

- if receive half as many acks => window grows half as fast

Slow start with window = 1

- ack will be delayed, even though sender is waiting for ack to expand window

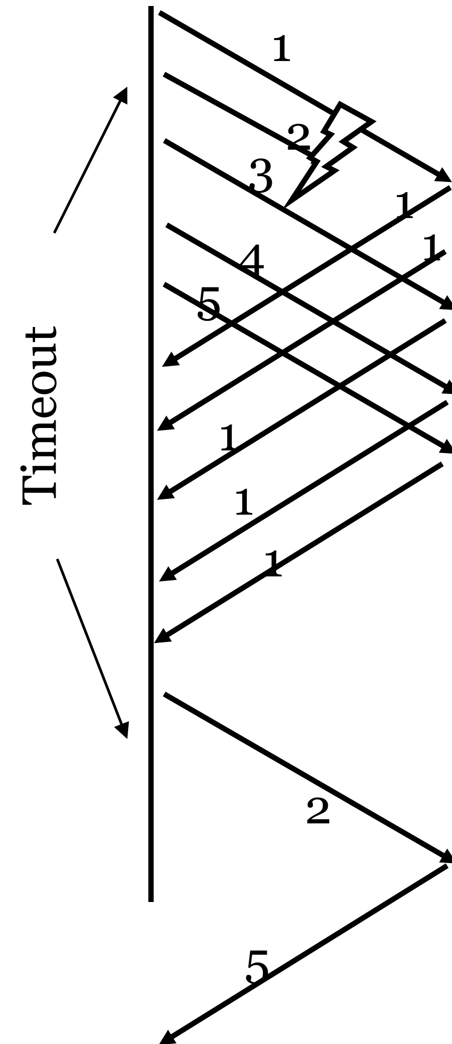
Avoiding burstiness: ack pacing



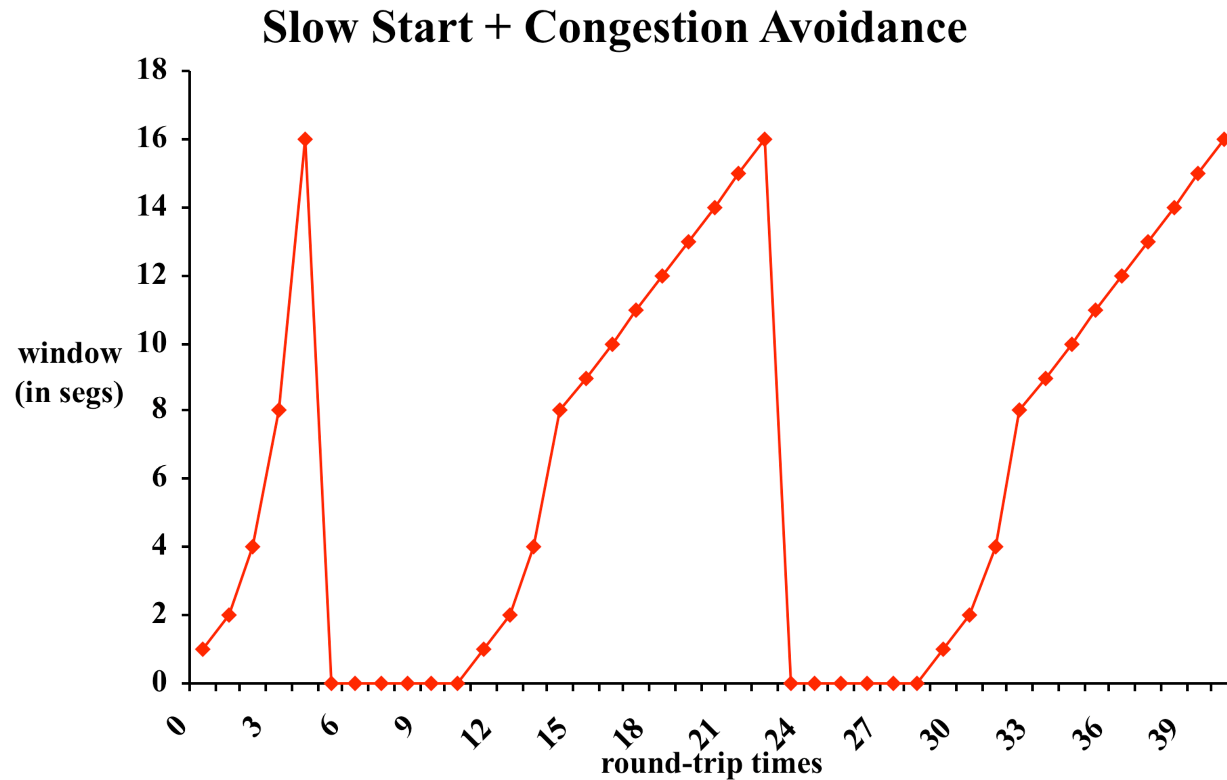
Window size = round trip delay * bit rate

Ack Pacing After Timeout

- Packet loss causes timeout, disrupts ack pacing
  - slow start/additive increase are *designed* to cause packet loss
- After loss, use slow start to regain ack pacing
  - switch to linear increase at last successful rate
  - "congestion avoidance"



Putting It All Together



Timeouts dominate performance!

Fast Retransmit

Can we detect packet loss without a timeout?

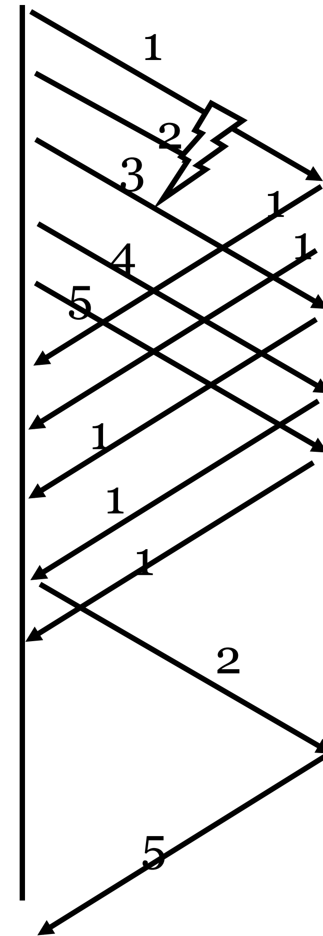
- Receiver will reply to each packet with an ack for last byte received in order

Duplicate acks imply either

- packet reordering (route change)
- packet loss

TCP Tahoe

- resend if sender gets three duplicate acks, without waiting for timeout



Fast Retransmit Caveats

Assumes in order packet delivery

- Recent proposal: measure rate of out of order delivery; dynamically adjust number of dup acks needed for retransmit

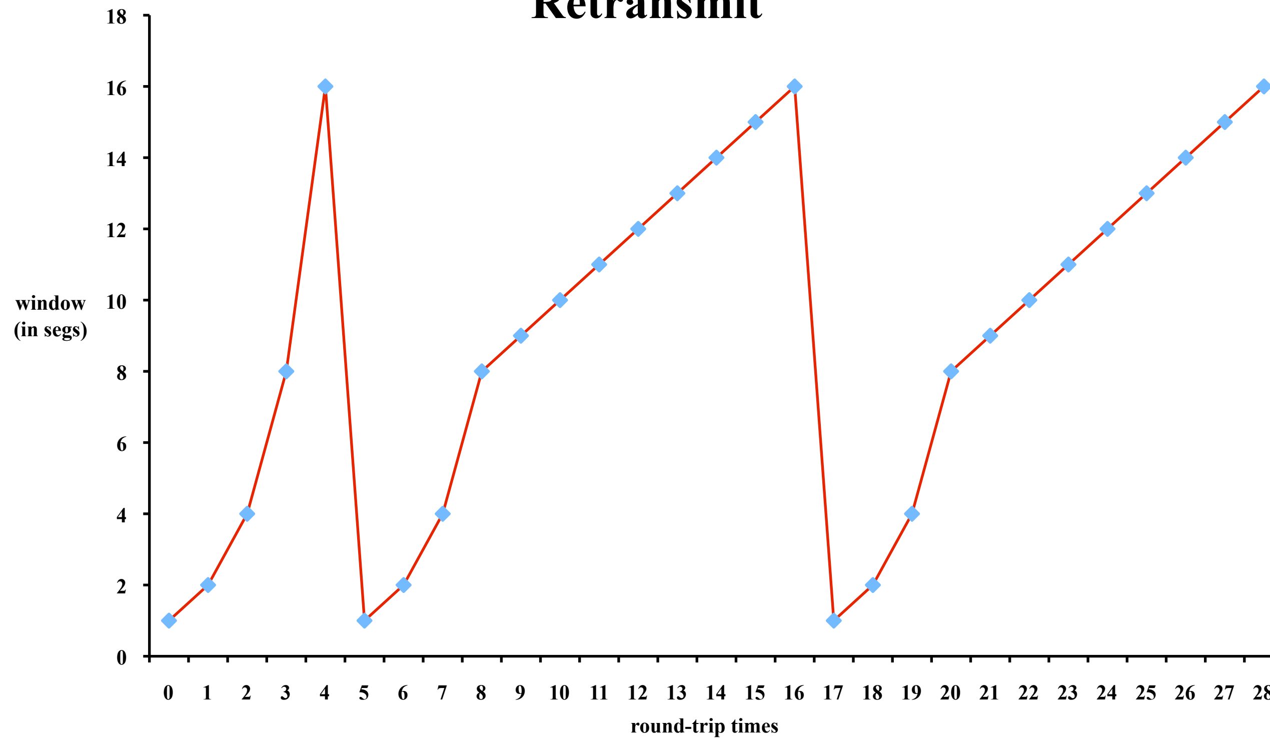
Doesn't work with small windows (e.g. modems)

- what if window size <= 3

Doesn't work if many packets are lost

- example: at peak of slow start, might lose many packets

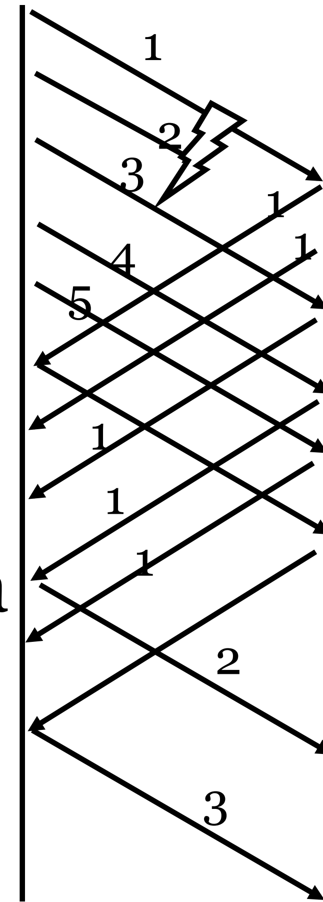
Fast Retransmit



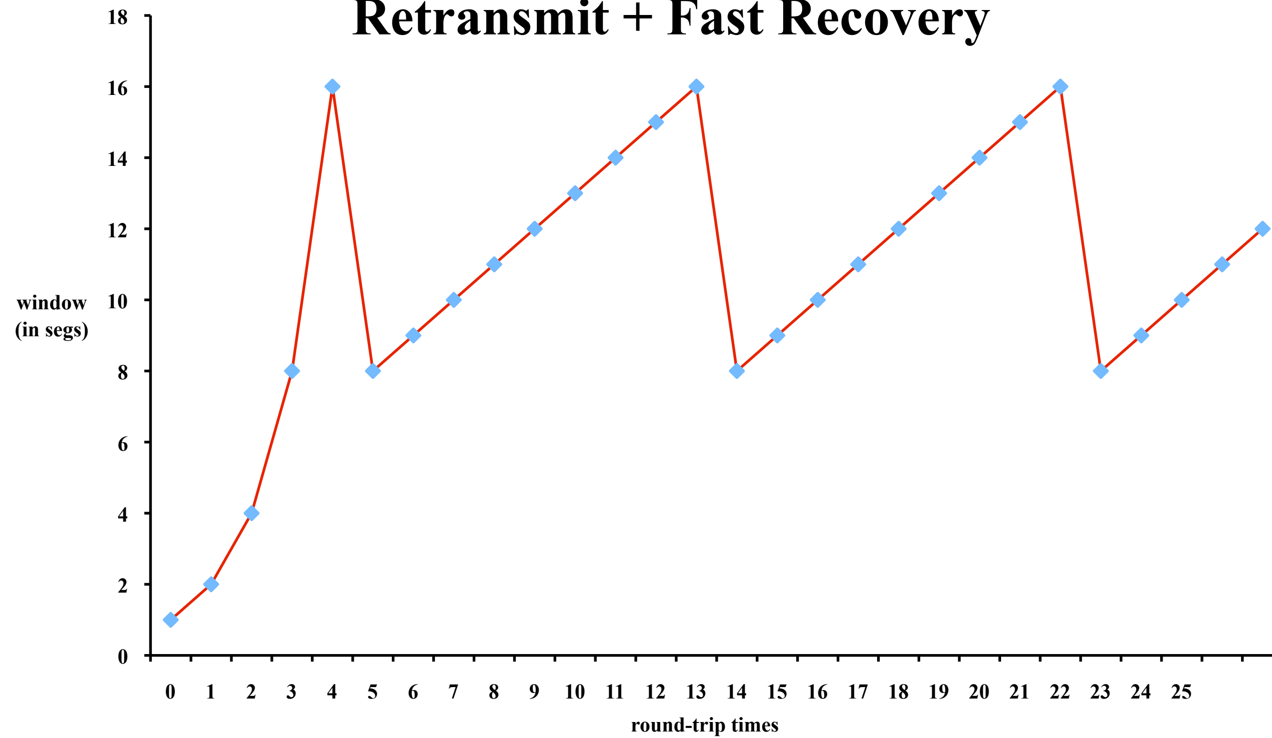
Regaining ack pacing limits performance

Fast Recovery

- Use duplicate acks to maintain ack pacing
  - duplicate ack => packet left network
  - after loss, send packet after every other acknowledgement
- Doesn't work if lose many packets in a row
  - fall back on timeout and slow start to reestablish ack pacing



Fast Recovery



TCP Performance (Steady State)

Bandwidth as a function of

- RTT?
- Loss rate?
- Packet size?
- Receive window?

TCP over 10Gbps Pipes

What's the problem?

How might we fix it?

TCP over Wireless

What's the problem?

How might we fix it?

What if TCP connection is short?

Slow start dominates performance

- What if network is unloaded?
- Burstiness causes extra drops

Packet losses unreliable indicator for short flows

- can lose connection setup packet
- Can get loss when connection near done
- Packet loss signal unrelated to sending rate

In limit, have to signal congestion (with a loss) on every connection

- 50% loss rate as increase # of connections

Example: 100KB transfer

100Mb/s Ethernet,100ms RTT, 1.5MB MSS

Ethernet ~ 100 Mb/s

64KB window, 100ms RTT ~ 6 Mb/s

slow start (delayed acks), no losses ~ 500 Kb/s

slow start, with 5% drop ~ 200 Kb/s

Steady state, 5% drop rate ~ 750 Kb/s

Improving Short Flow Performance

- Start with a larger initial window
  - RFC 3390: start with 3-4 packets
- Persistent connections
  - HTTP: reuse TCP connection for multiple objects on same page
  - Share congestion state between connections on same host or across host
- Skip slow start?
- Ignore congestion signals?

Misbehaving TCP Receivers

On server side, little incentive to cheat TCP

- Mostly competing against other flows from same server

On client side, high incentive to induce server to send faster

- How?