Congestion Control

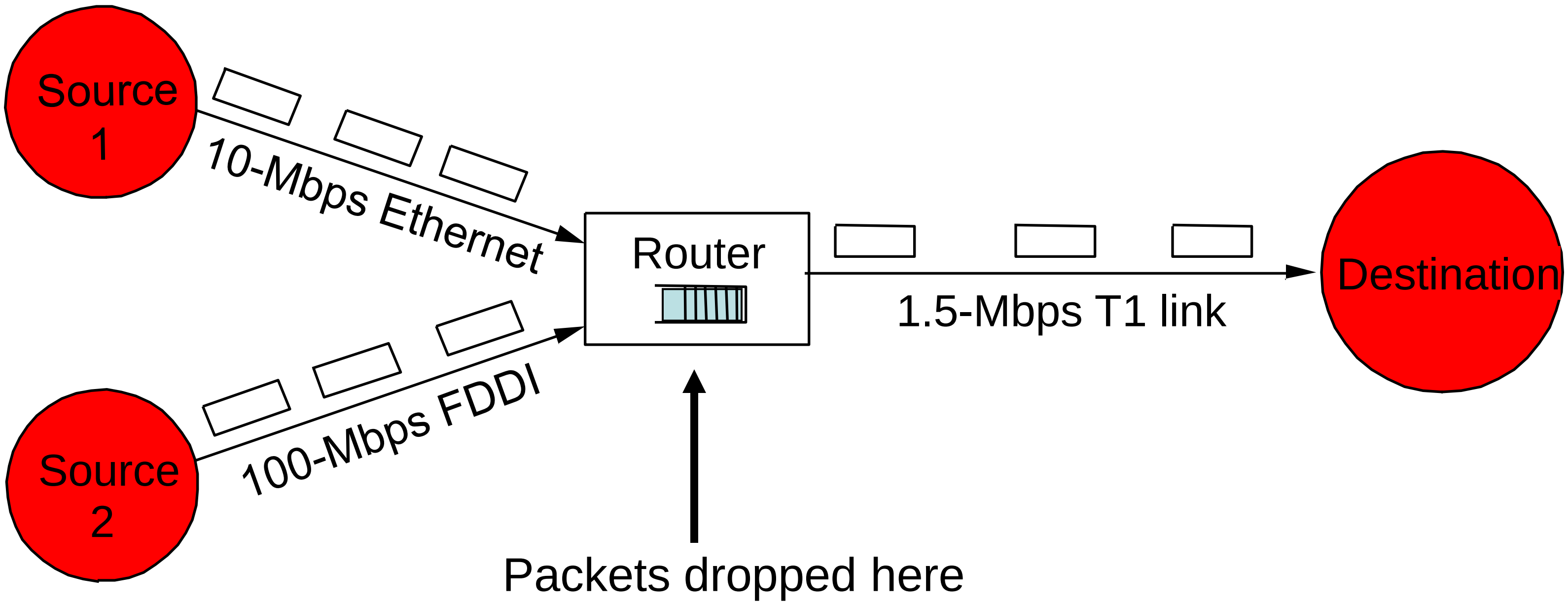
Tom Anderson

Bandwidth Allocation

How do we efficiently share network resources among billions of hosts?

- Congestion control
  - Sending too fast causes packet loss inside network -> retransmissions -> more load -> more packet losses -> …
  - Don’t send faster than network can accept
- Fairness
  - How do we allocate bandwidth among different users?
  - Each user should (?) get fair share of bandwidth

Congestion

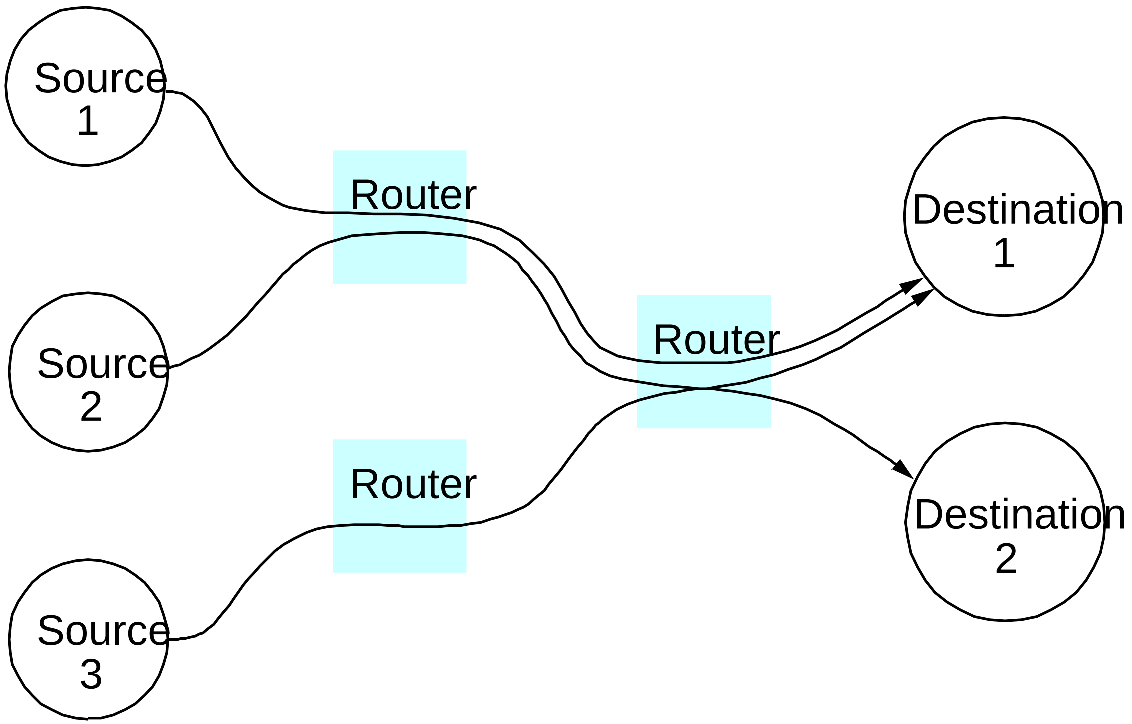


Buffer absorbs bursts when input rate > output

If sending rate is persistently > drain rate, queue builds

Dropped packets represent wasted work

Fairness



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

Chapter 6, Figure 2

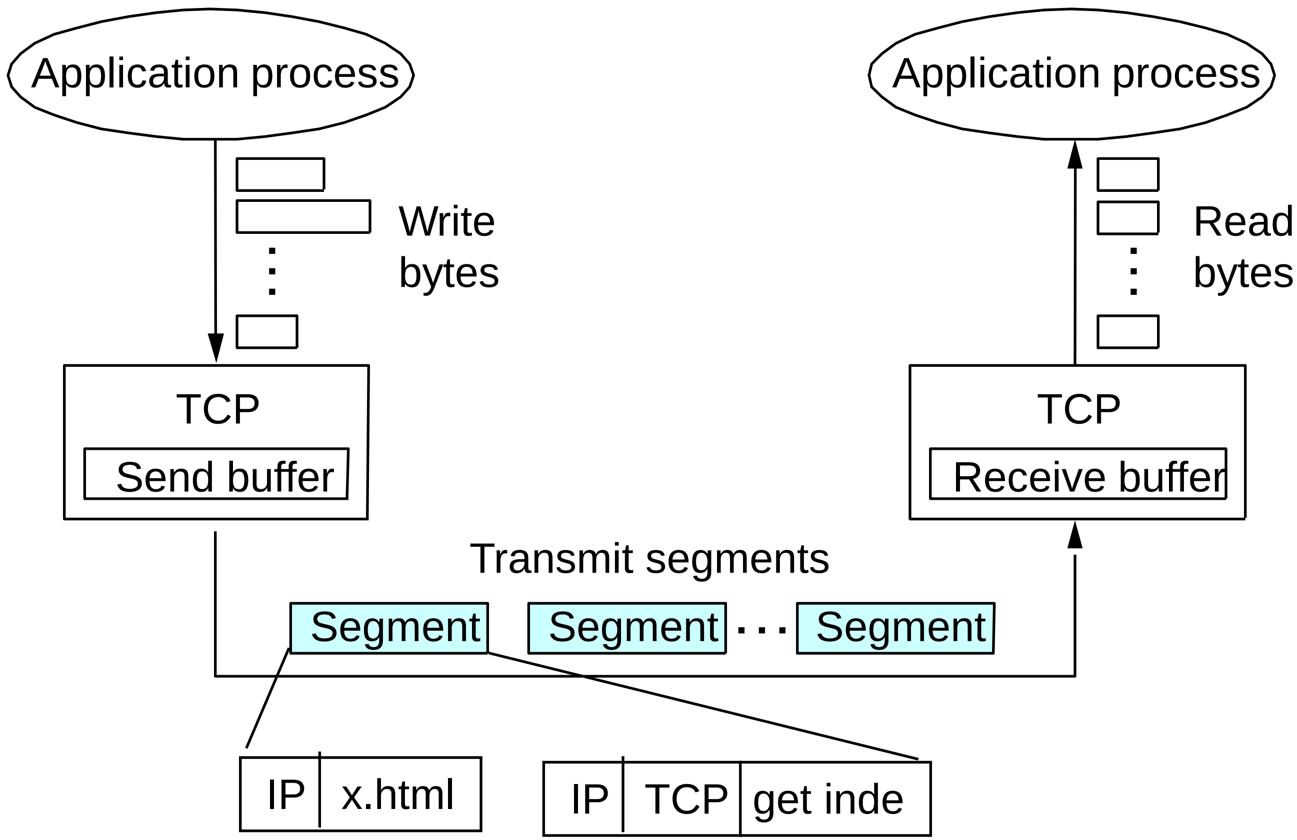
The Problem

Original TCP sent full window of data

When links become loaded, queues fill up, and this can lead to:

- C*ongestion collapse: w*hen round-trip time exceeds retransmit interval -- every packet is retransmitted many times
- Synchronized behavior: network oscillates between loaded and unloaded

TCP Delivery



TCP Sliding Window

Per-byte, not per-packet (why?)

- send packet says “here are bytes j-k”
- ack says “received up to byte k”

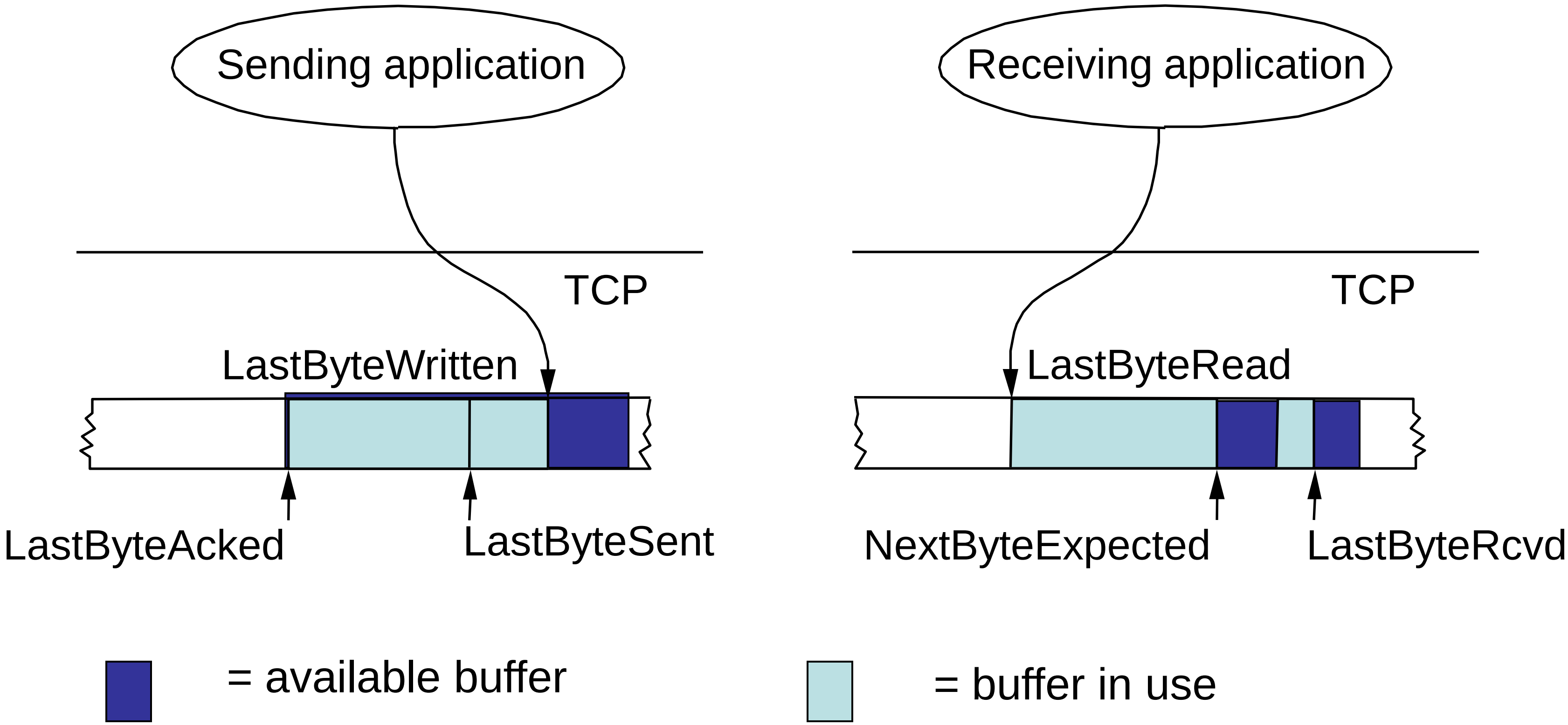
Send buffer >= send window

- can buffer writes in kernel before sending
- writer blocks if try to write past send buffer

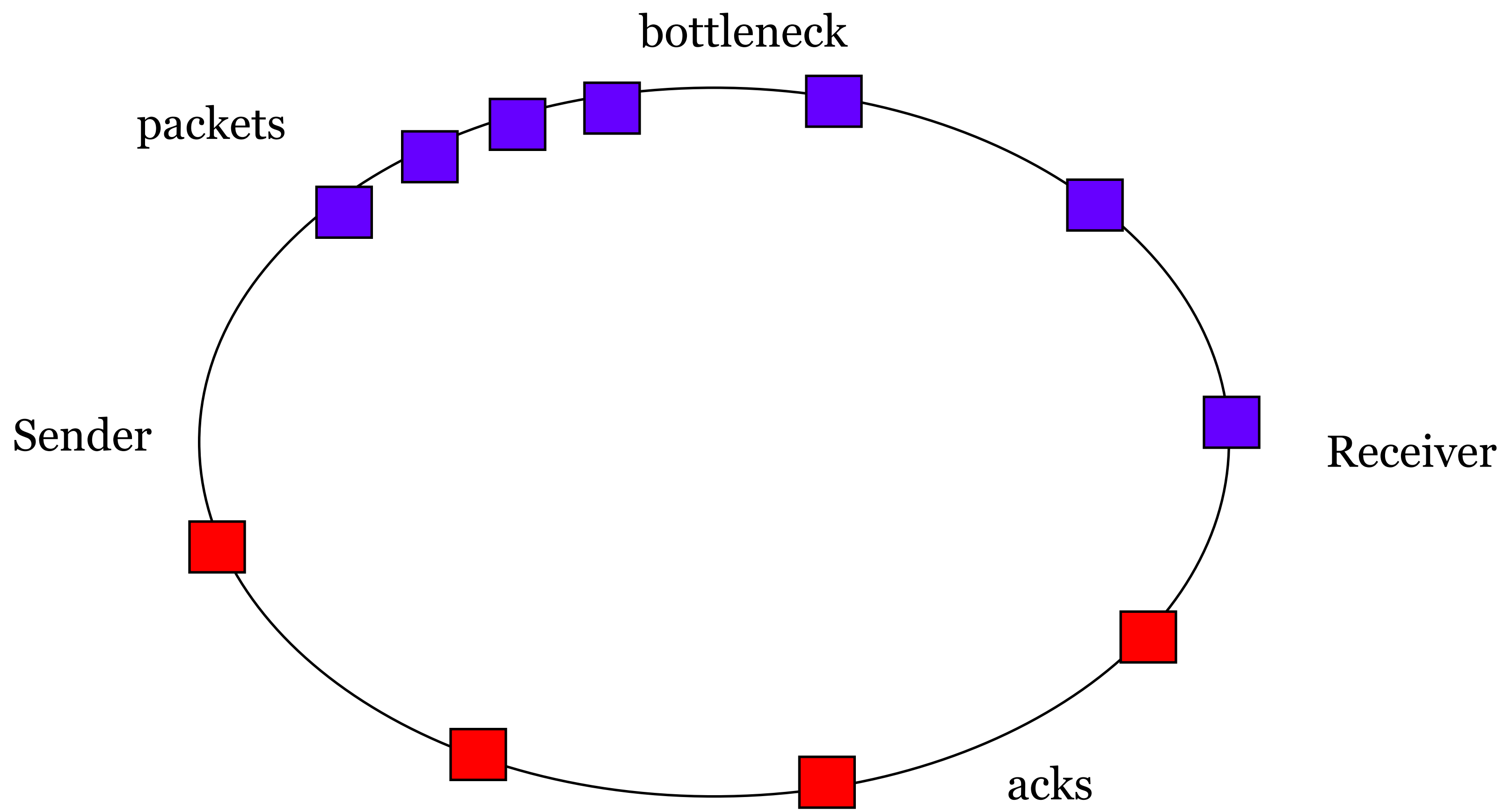
Receive buffer >= receive window

- buffer acked data in kernel, wait for reads
- reader blocks if try to read past acked data

Sender and Receiver Buffering



Avoiding burstiness: ack pacing



Window size = round trip delay * bit rate

The Problem

Original TCP sent full window of data

When links become loaded, queues fill up, and this can lead to:

- C*ongestion collapse: w*hen round-trip time exceeds retransmit interval -- every packet is retransmitted many times
- Synchronized behavior: network oscillates between loaded and unloaded

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

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, …

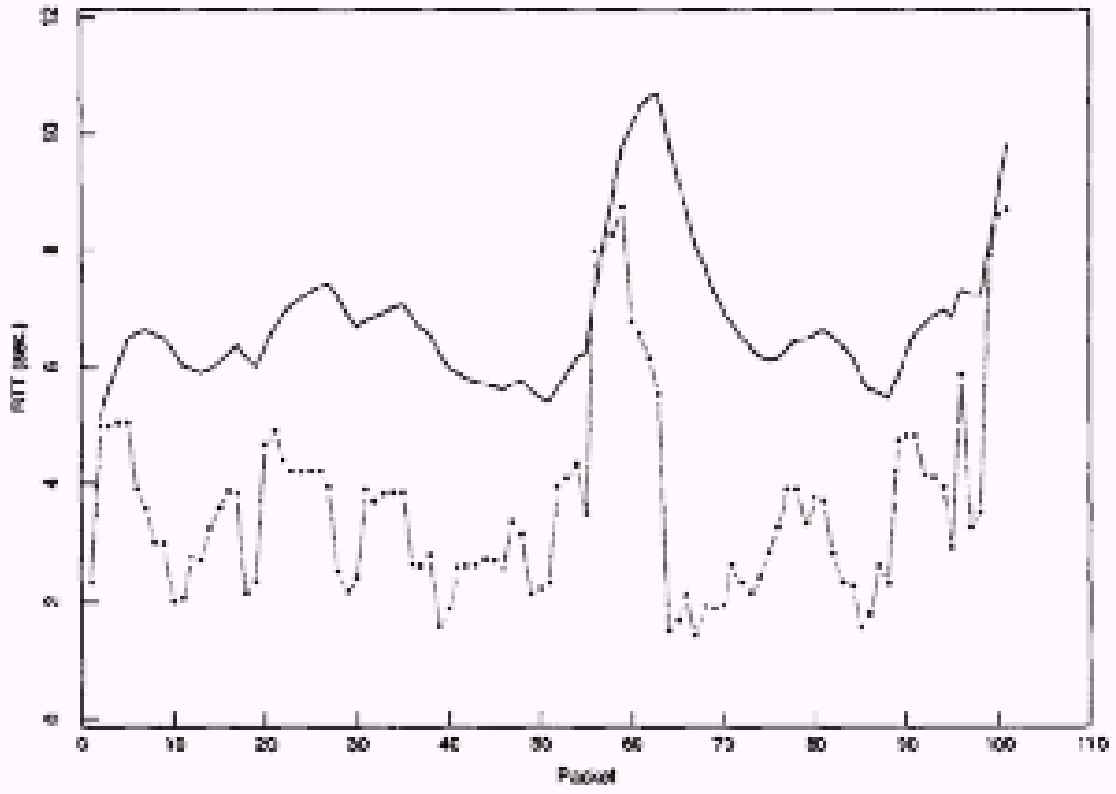
Estimating RTTs

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



Jacobson/Karels Algorithm

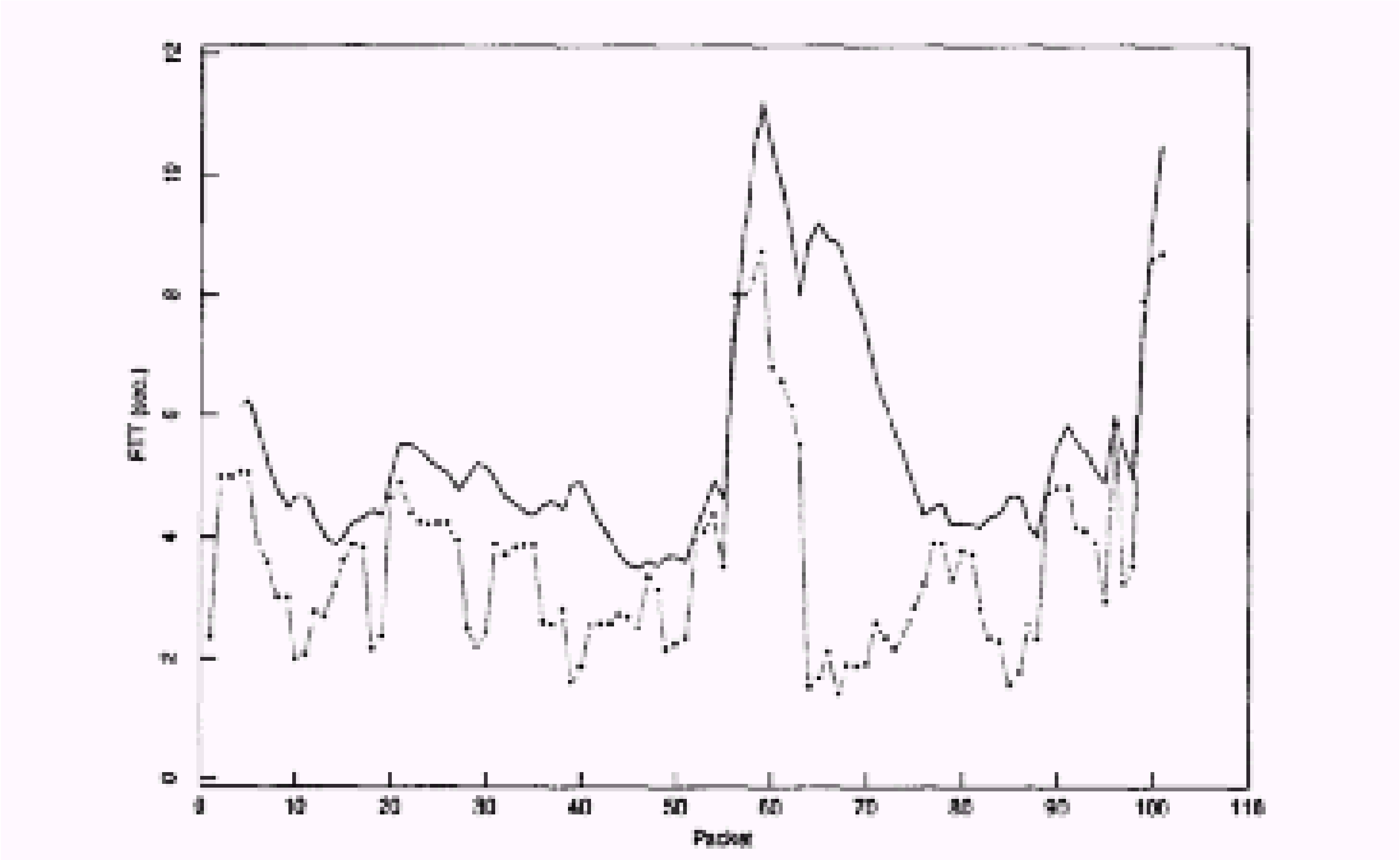
Problem:

- 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



Tracking the Bottleneck Bandwidth

Sending rate = window size/RTT

Multiplicative decrease

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

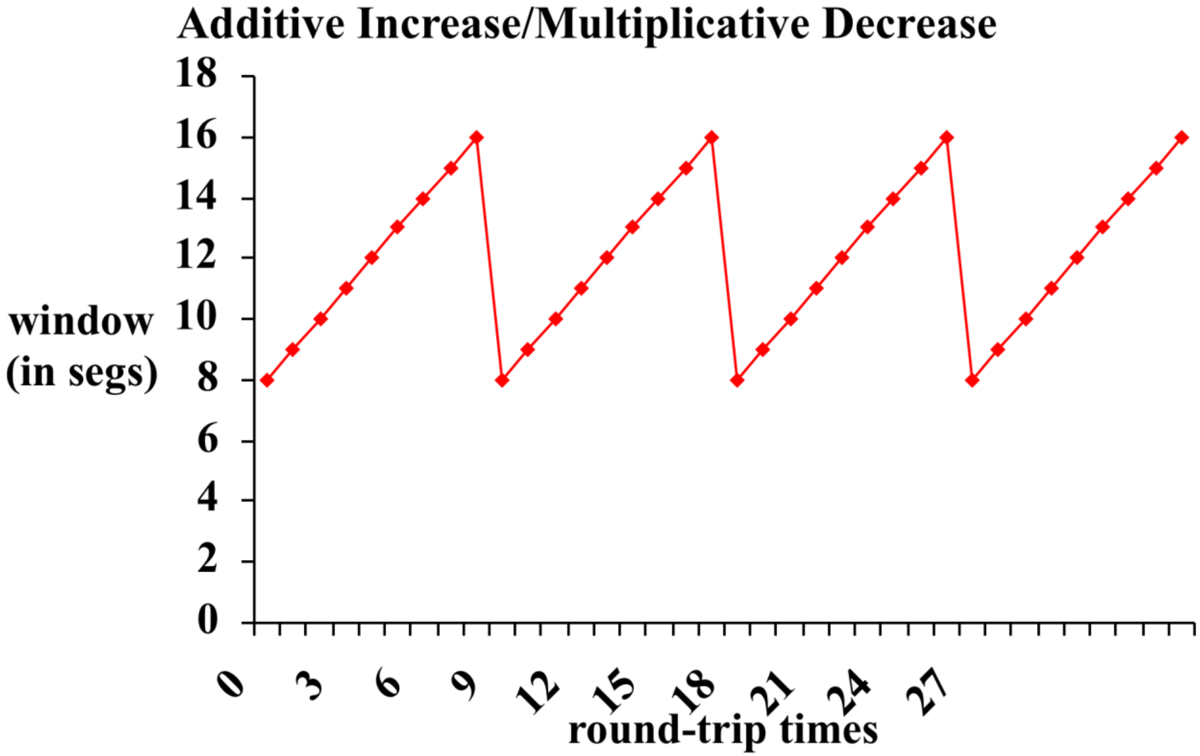
Additive increase

- Ack arrives => no drop => 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



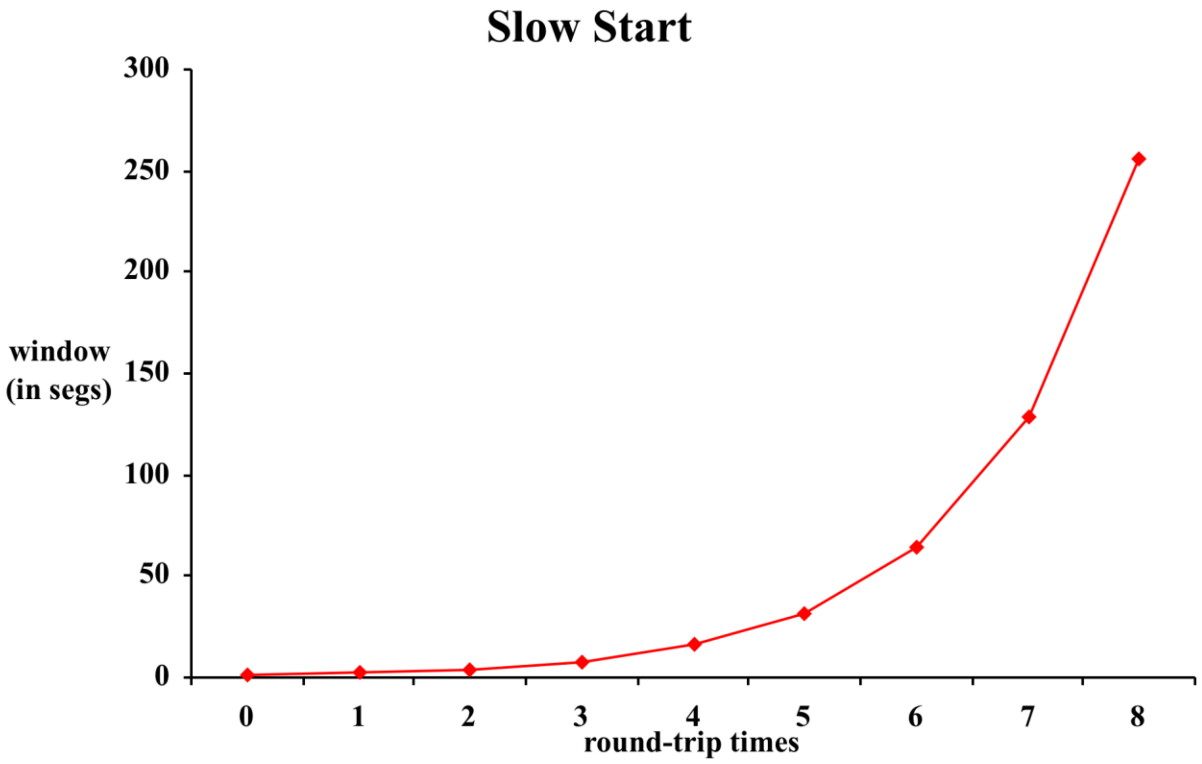
Slow start

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

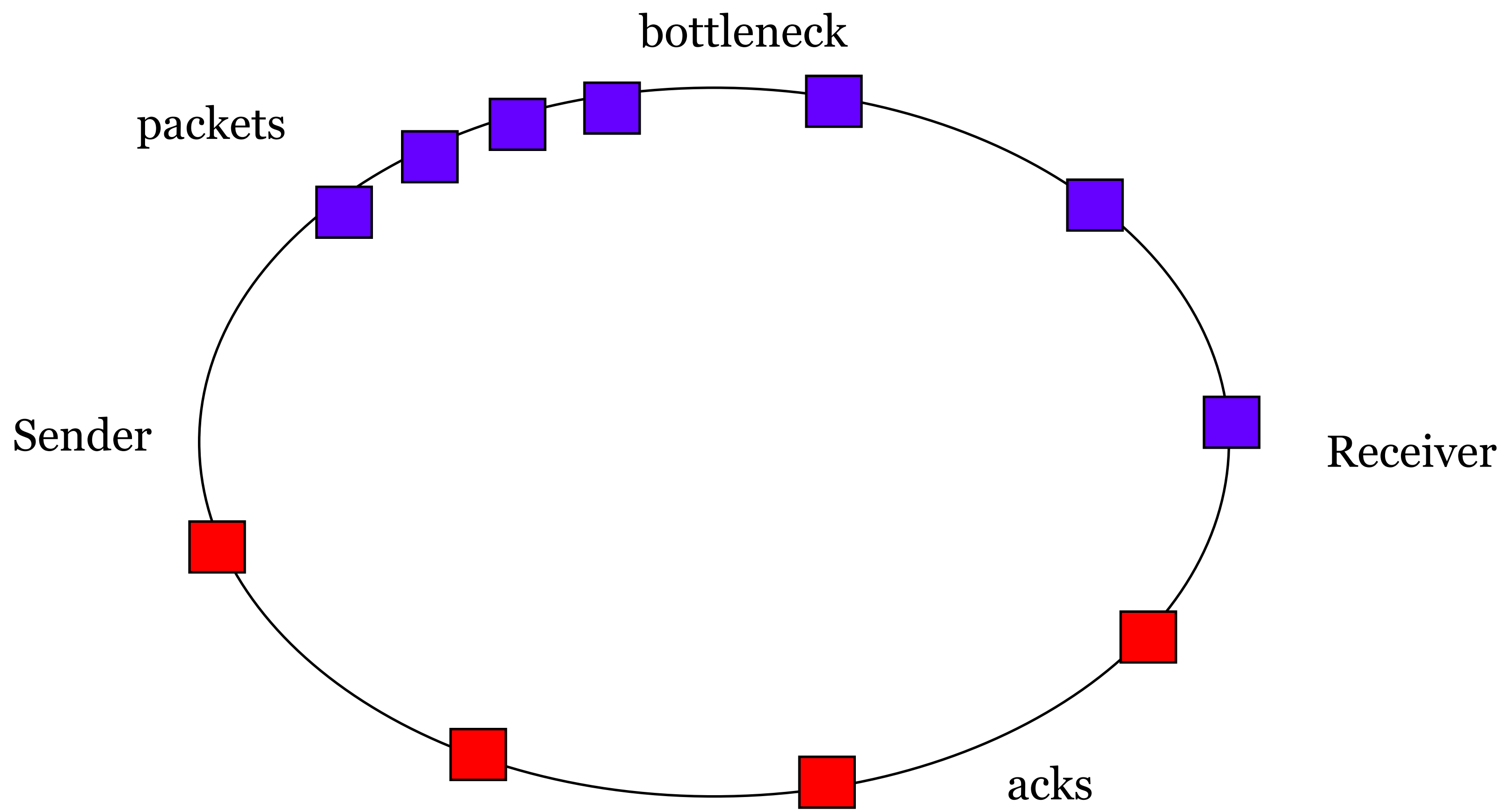
Slow Start Problems

Bursty traffic source
- will fill up router queues, causing losses for other flows
- solution: ack pacing

Slow start usually overshoots bottleneck
- will lose many packets in window
- solution: remember previous threshold

Short flows
- Can spend entire time in slow start!
- solution: persistent connections?

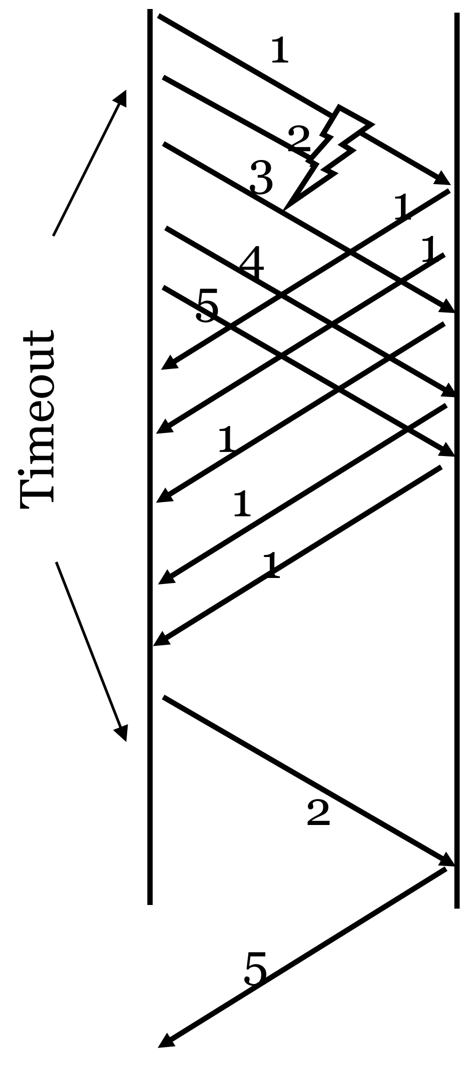
Avoiding burstiness: ack pacing



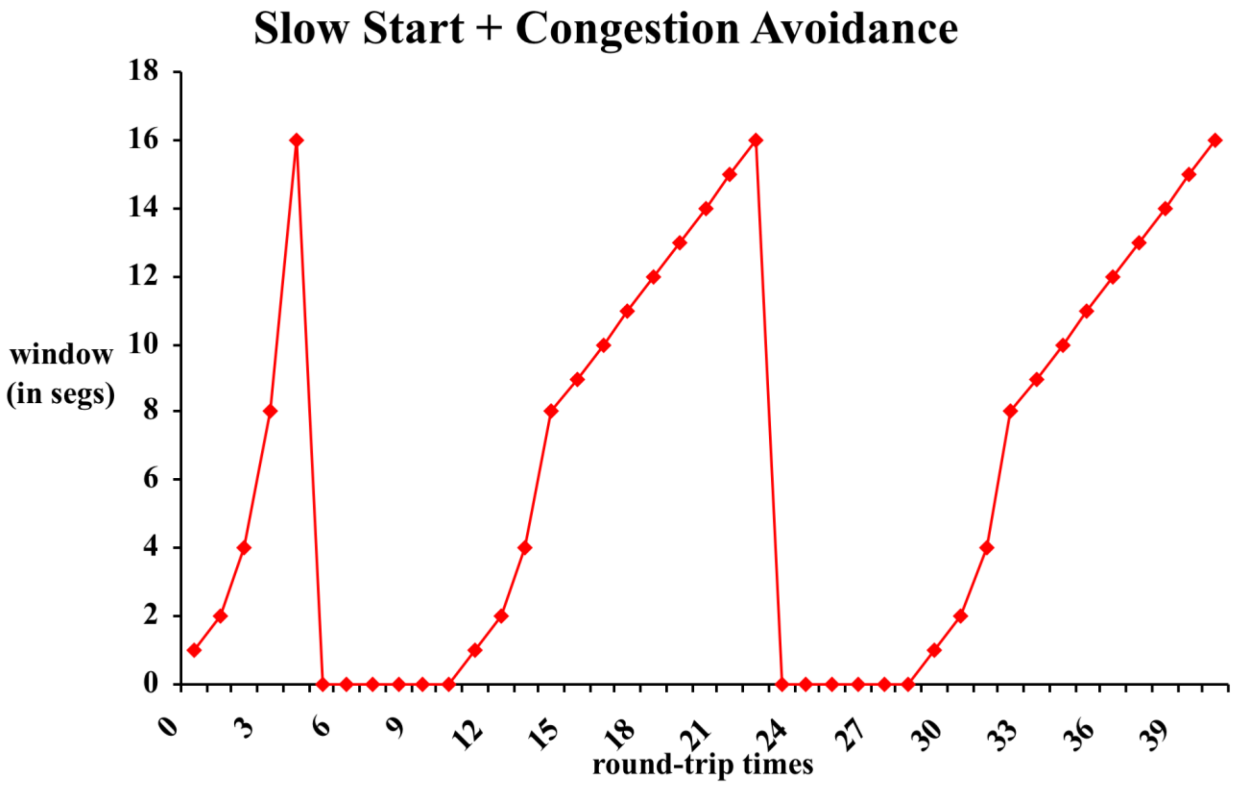
Window size = round trip delay * bit rate

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"



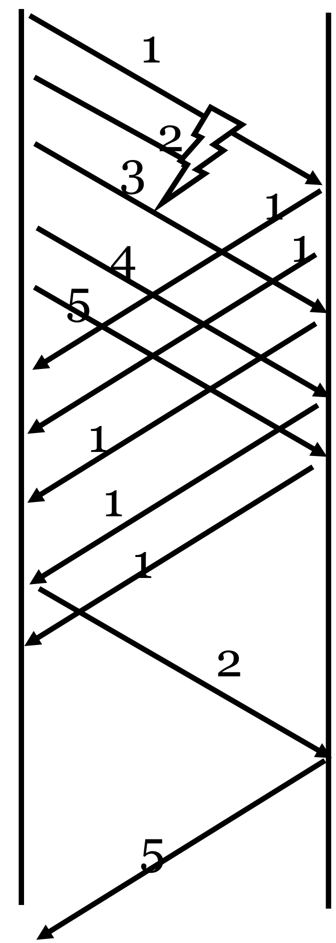
Putting It All Together



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

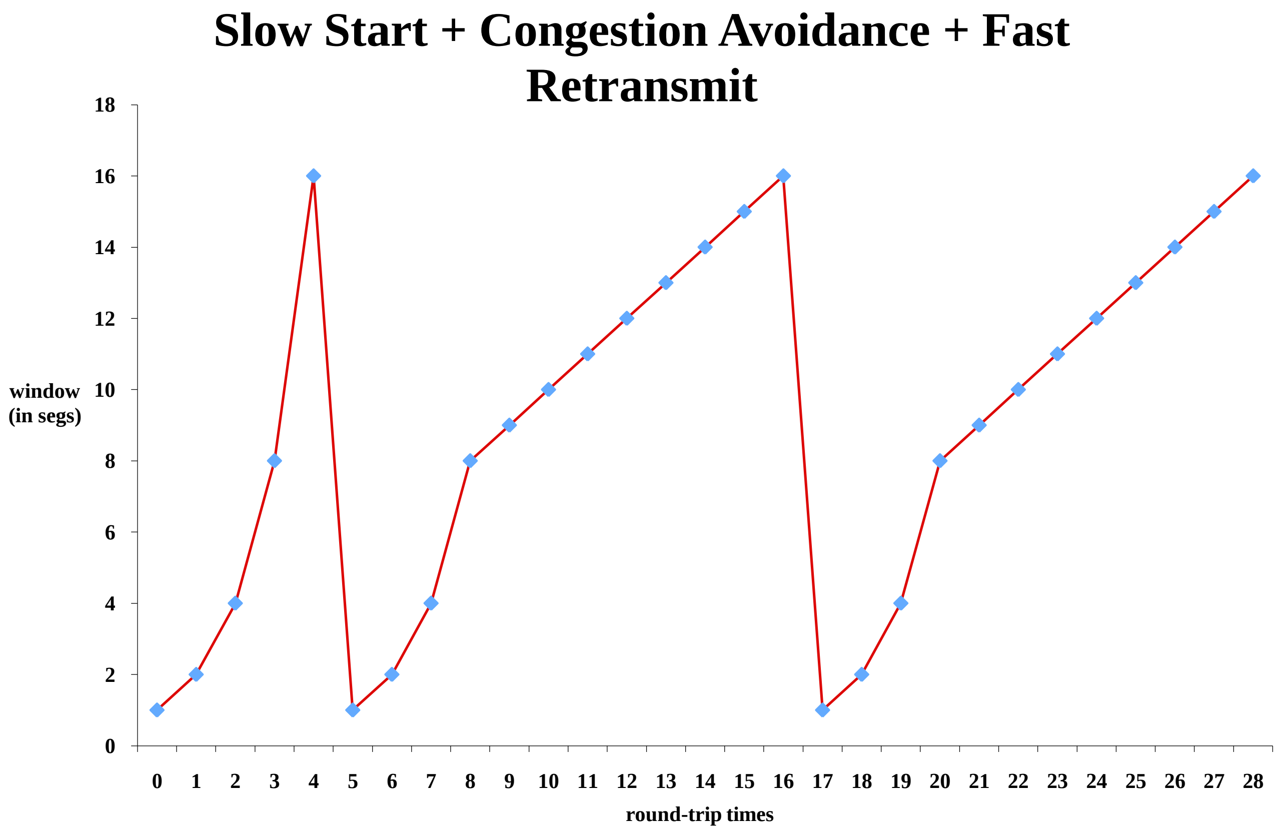
Doesn't work with small windows (e.g. modems)

- what if window size <= 3

Doesn't work if many packets are lost

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

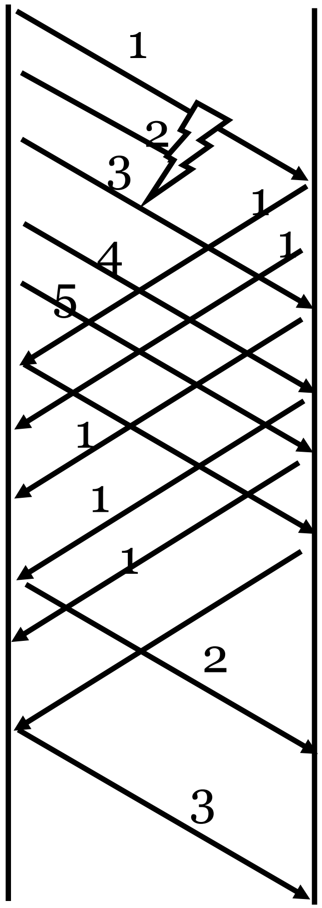
Fast Retransmit



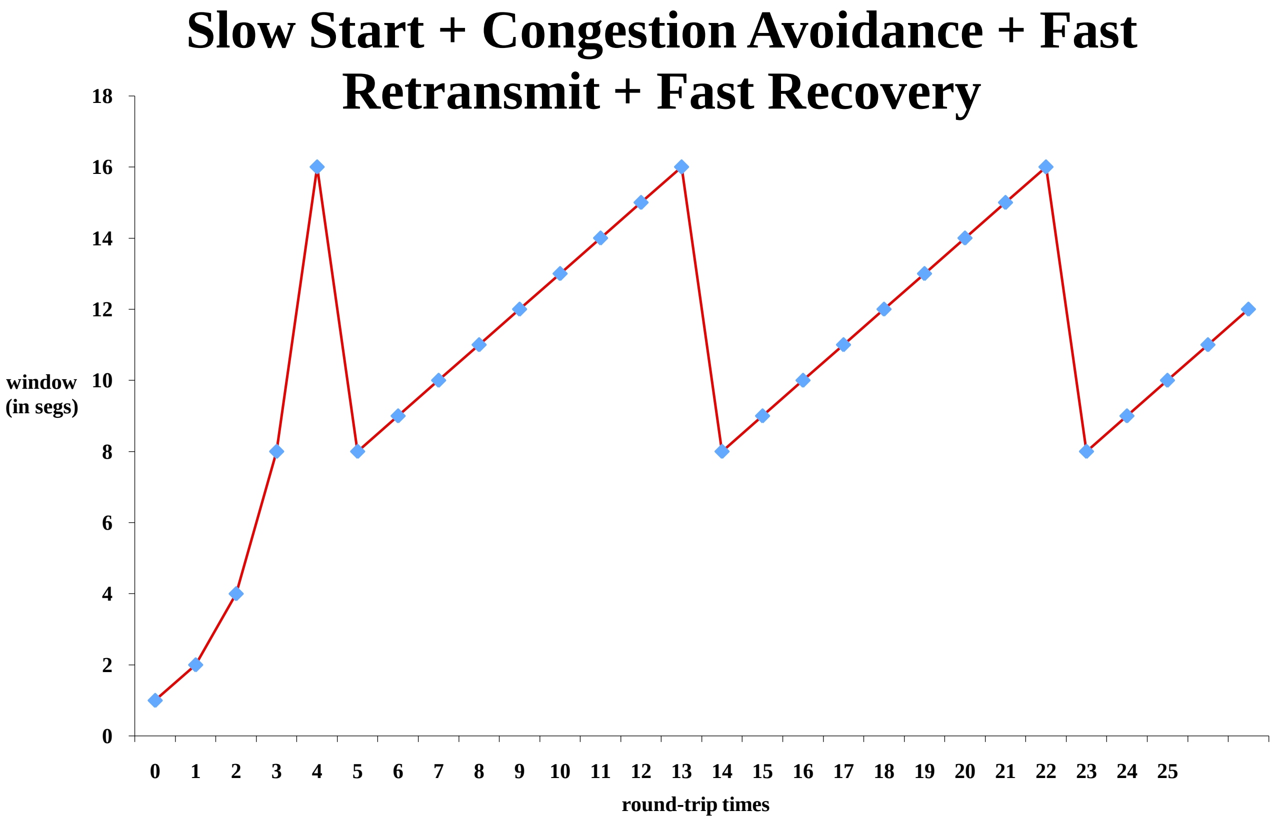
Regaining ack pacing limits performance

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



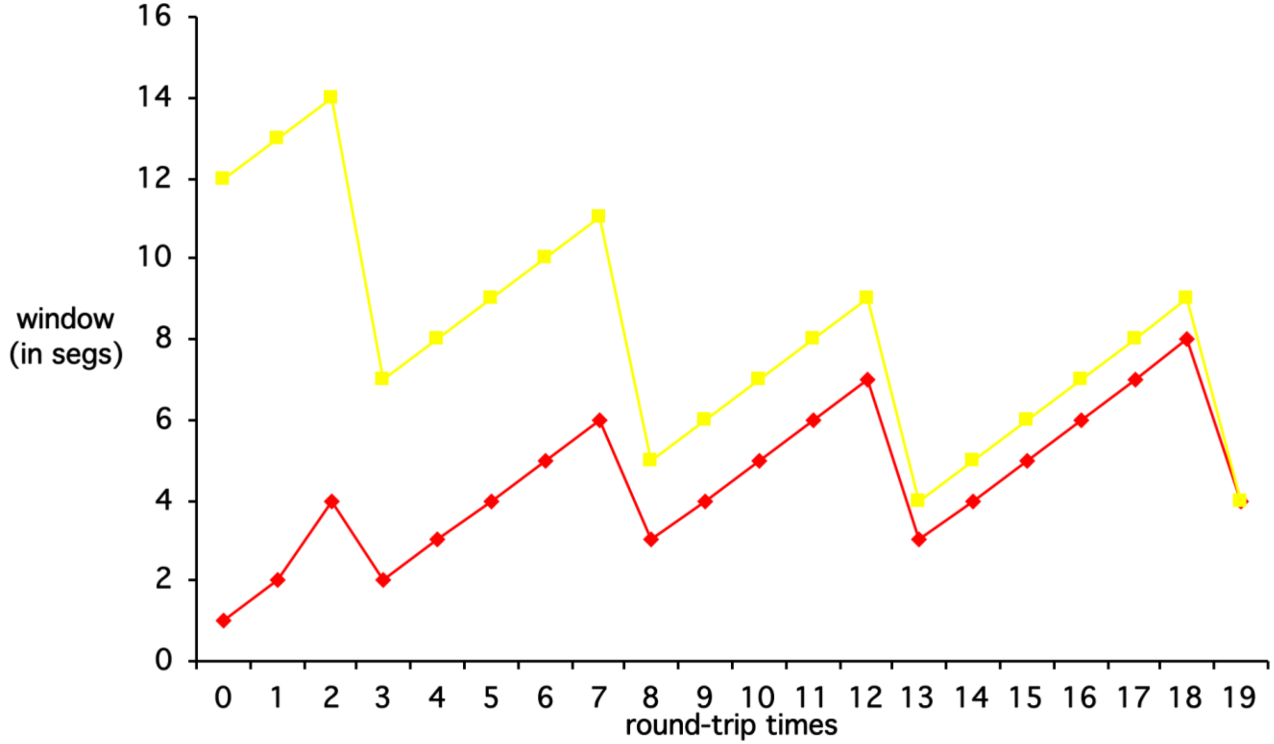
Fast Recovery



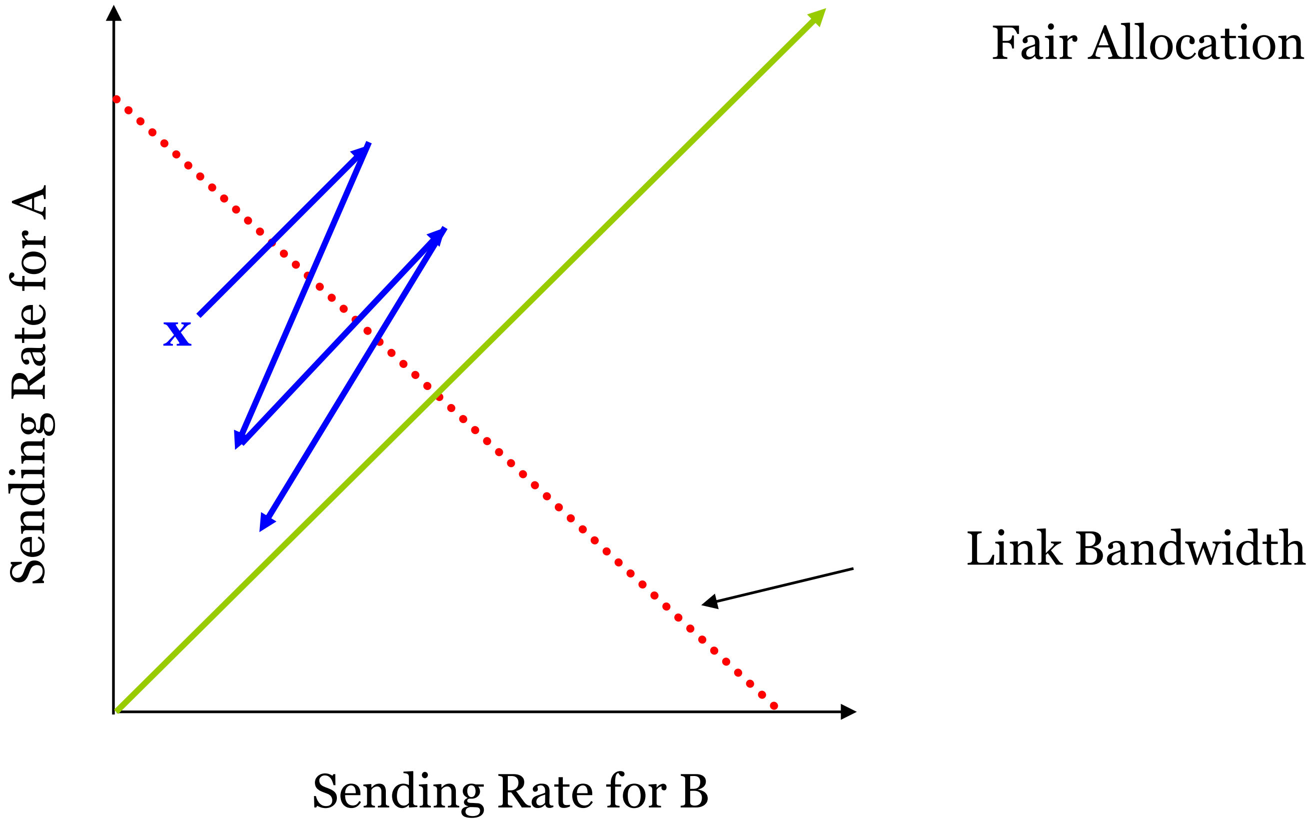
What if two TCPs share link?

Reach equilibrium independent of initial bw

- assuming equal RTTs, “fair” drops at the router

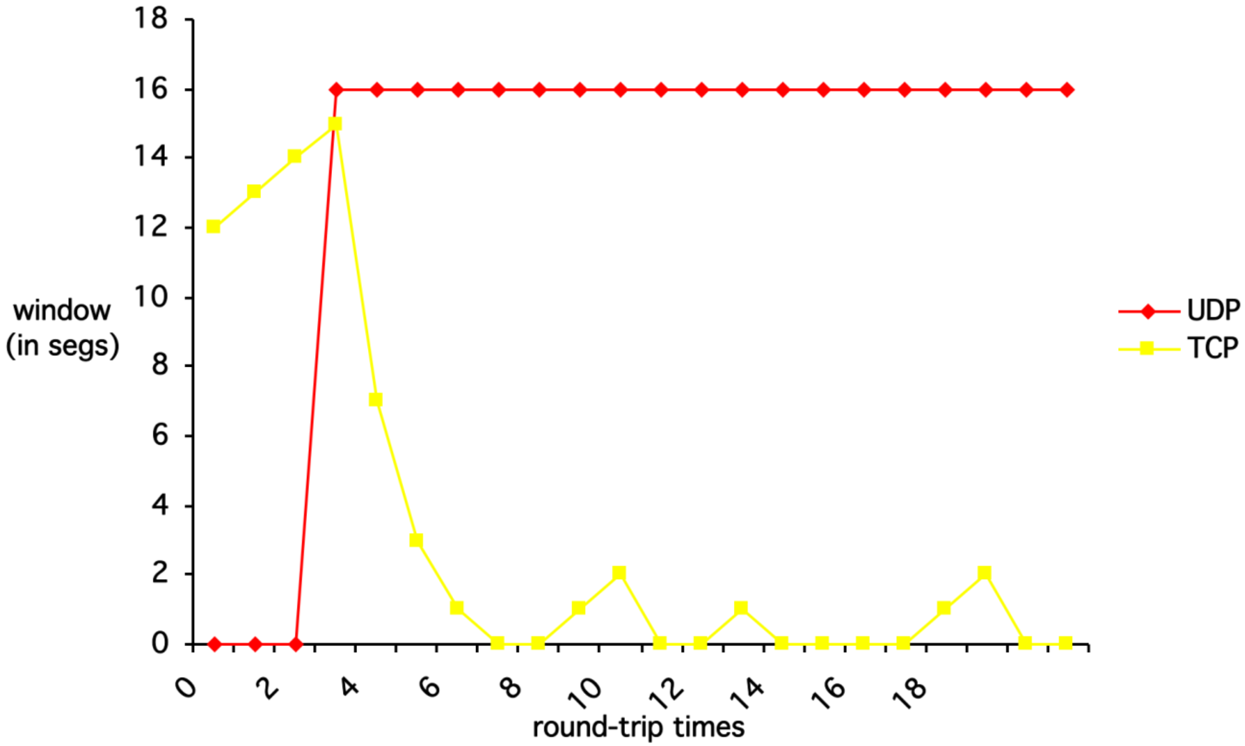


Equilibrium Proof



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

What if TCP connection is short?

Slow start dominates performance

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

Packet losses unreliable indicator

- can lose connection setup packet
- can get drop when connection near done
- signal unrelated to sending rate

In limit, have to signal every connection

- 50% loss rate as increase # of connections

Example: 10KB document

10Mb/s wifi,70ms RTT, 536 MSS

Ethernet ~ 10 Mb/s

64KB window, 70ms RTT ~ 7.5 Mb/s

can only use 10KB window ~ 1.2 Mb/s

5% drop rate ~ 275 Kb/s (steady state)

model timeouts ~ 228 Kb/s

slow start, no losses ~ 140 Kb/s

slow start, with 5% drop ~ 75 Kb/s

Other Issues

TCP over wireless

- High loss rate => ?

TCP in the data center

- Slow start => ?

TCP over 10 Gbps links

- Packet loss => ?

TCP and router buffer sizes

- Buffer = bw*delay; what happens to latency?

TCP and real-time delivery

- Competing flows drive system to overload

TCP Known to be Suboptimal

Small to moderate sized connections

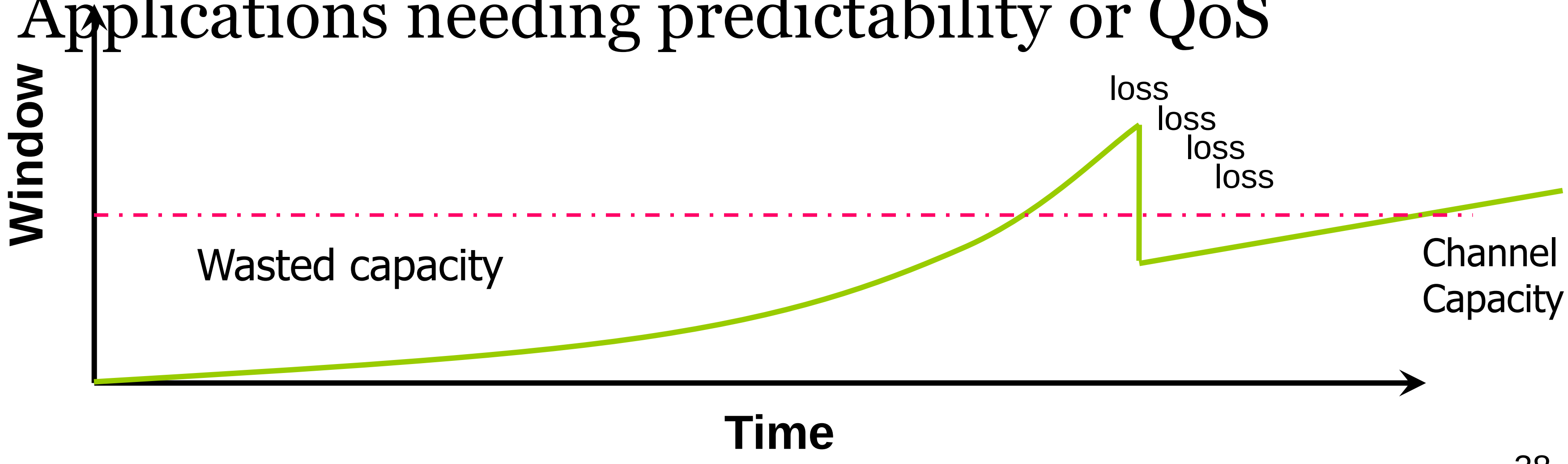
Intranets with low to moderate utilization

Wireless transmission loss

High bandwidth; high delay

Interactive applications

Applications needing predictability or QoS



Observation

Trivial to be optimal with help from the network; e.g., ATM rate control

- Hosts send bandwidth request into network
- Network replies with safe rate (min across links in path)

Can endpoint congestion control be near optimal with *no* change to the network?

- Assume: cooperating endpoints
- Router support only for isolation, not congestion control

PCP approach: directly emulate optimal router behavior!

Congestion Control Approaches

	Endpoint	Router Support
Try target rate for full RTT; if too fast, backoff	TCP, Vegas, RAP, FastTCP, Scalable TCP, HighSpeed TCP	DecBit, ECN, RED, AQM
Request rate from network; send at that rate	**PCP**	ATM, XCP, WFQ, RCP

PCP Goals

1. Minimize transfer time
2. Negligible packet loss, low queueing
3. Work conserving
4. Stability under extreme load
5. Eventual fairness

TCP achieves 3-5 (mostly)

PCP achieves all five (in the common case)

Probe Control Protocol (PCP)

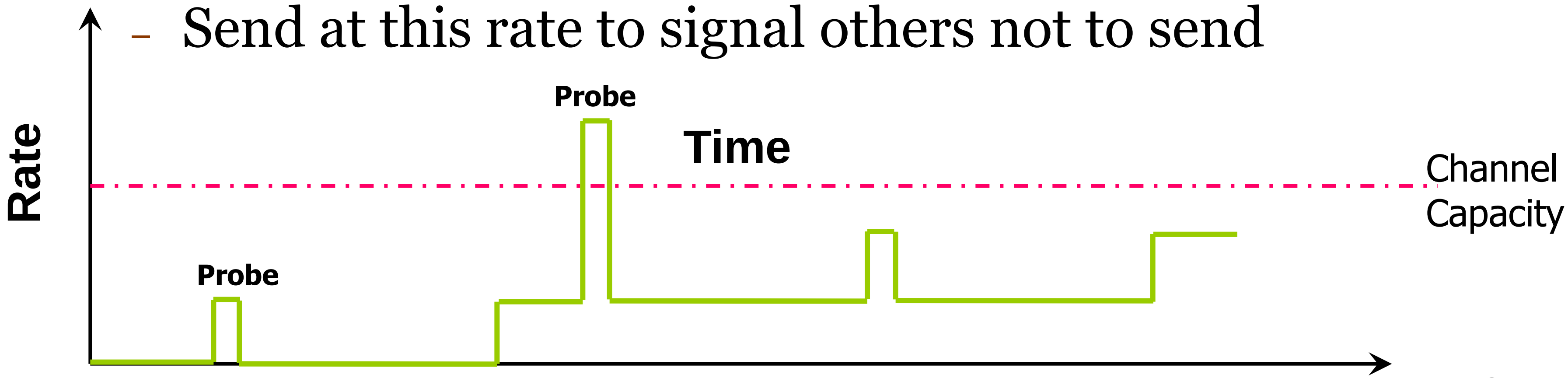
Probe for bandwidth using short burst of packets

- If bw available, send at the desired uniform rate (paced)
- If not, try again at a slower rate

Probe is a request

Successful probe sets the sending rate

- Send at this rate to signal others not to send

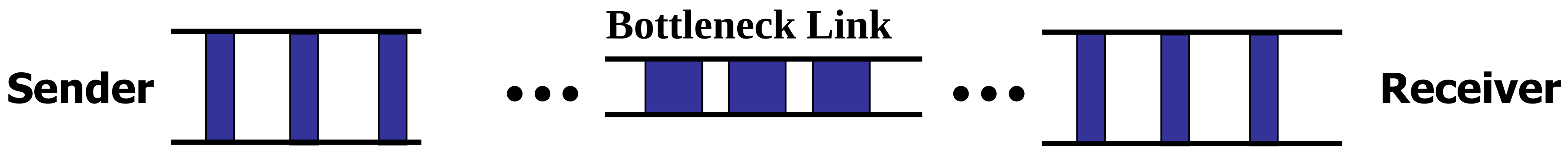


Probes

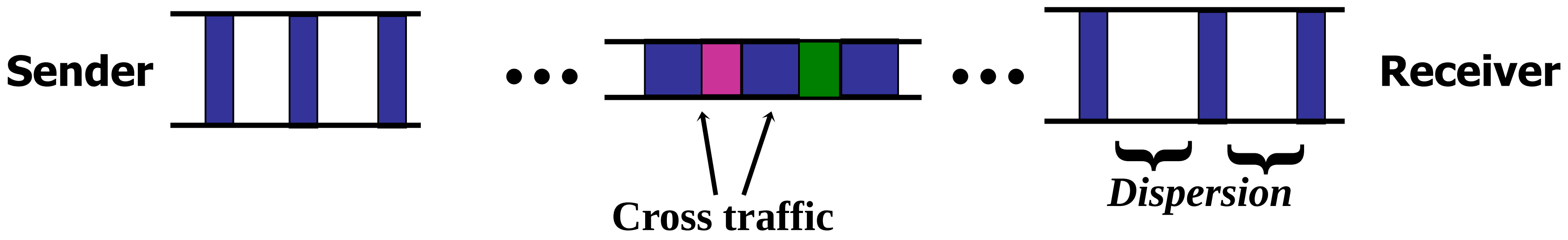
Send packet train spaced to mimic desired rate

Check packet dispersion at receiver

Successful probe:



Failed probe:



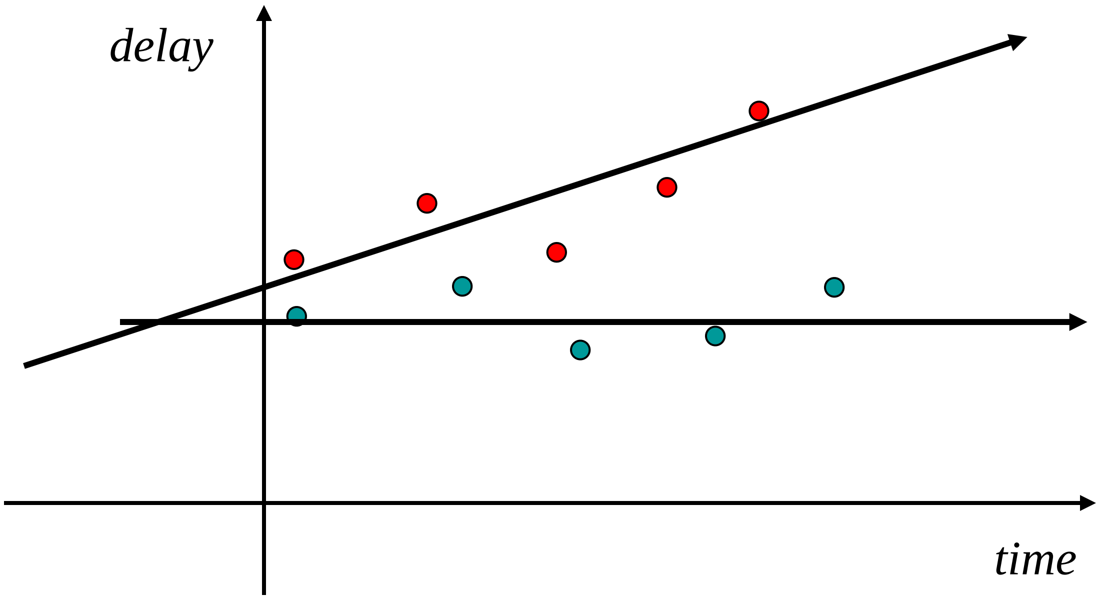
Probabilistic Accept

Randomly generate a slope consistent with the observed data

- same mean, variance as least squares fit

Accept if slope is not positive

Robust to small variations in packet scheduling



Rate Compensation

Queues can still increase:

- Failed probes, even if short, can add to queueing
- Simultaneous probes could allocate the same bw
- Probabilistic accept may decide probe was successful, without sufficient underlying available bandwidth

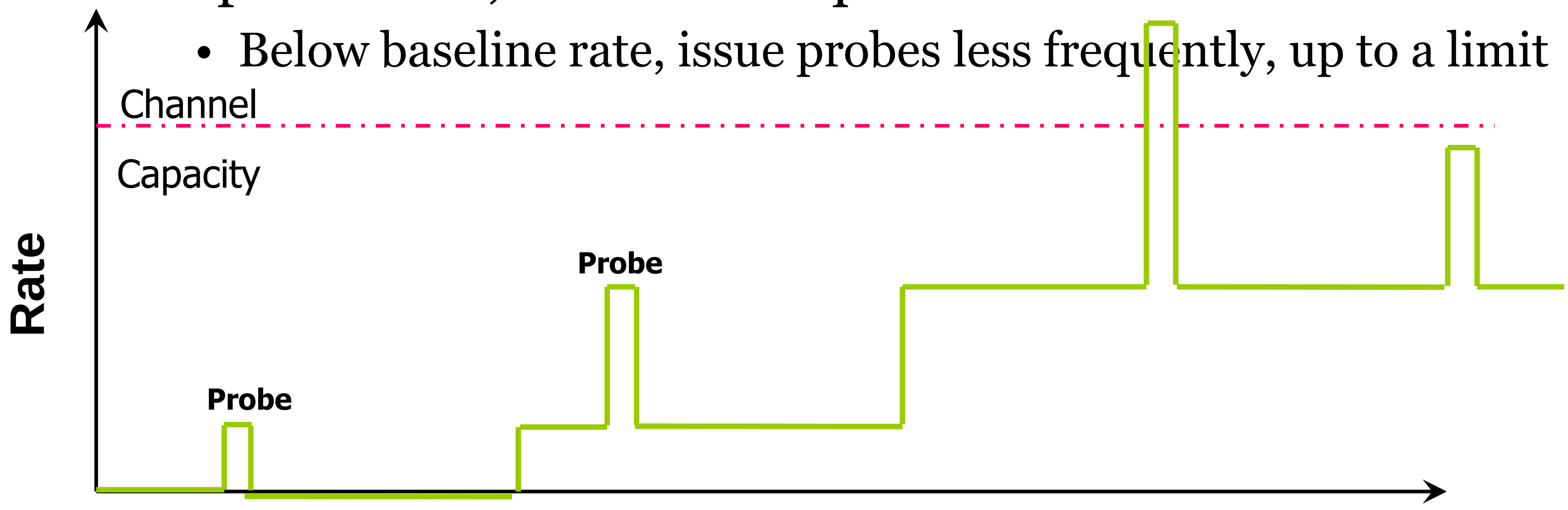
PCP solution

- Detect increasing queues by measuring packet latency and inter-packet delay
- Each sender decreases their rate proportionately, to eliminate queues within a single round trip
- Emulates AIMD, and thus provides eventual fairness

Binary Search

Base protocol: binary search for channel capacity

- Start with a baseline rate: One MSS packet per round-trip
- If probe succeeds, double the requested bandwidth
- If probe fails, halve the requested bandwidth
  - Below baseline rate, issue probes less frequently, up to a limit



History

Haven't we just reinvented TCP slow start?

- Still uses *O(log n)* steps to determine the bandwidth
- Does prevent losses, keeps queues small

Host keeps track of previous rate for each path

- Because probes are short, ok to probe using this history
- Currently: first try 1/3rd of previous rate
  - If prediction is inaccurate/accurate, we halve/double the initial probe rate

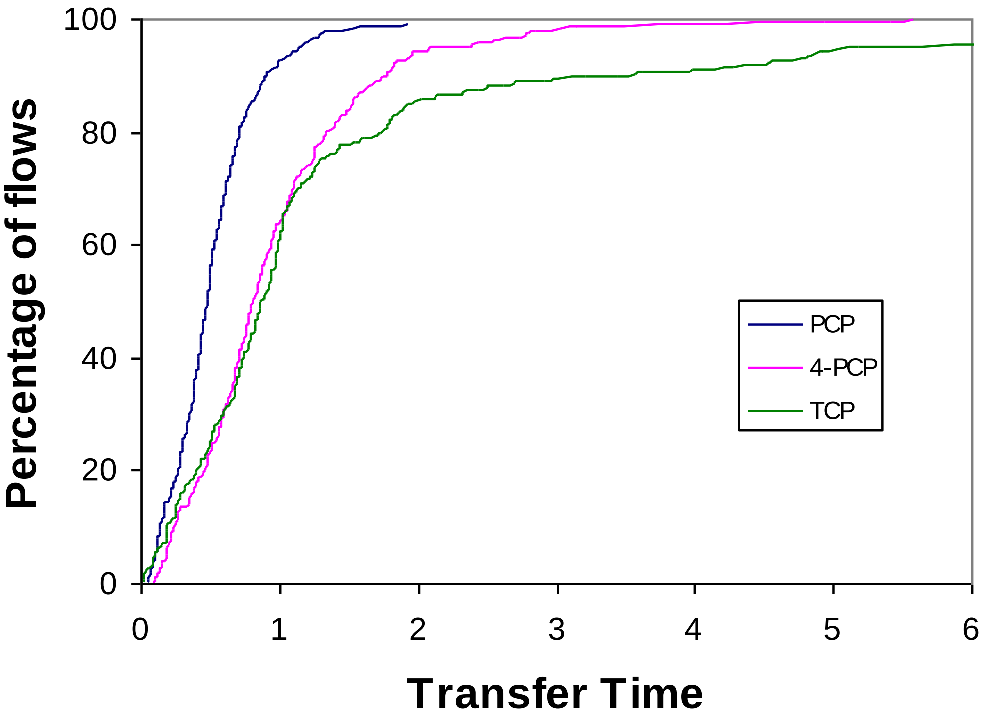
TCP Compatibility

- TCP increases its rate regardless of queue size
  - Should PCP keep reducing its rate to compensate?
- Solution: PCP becomes more aggressive in presence of non-responsive flows
  - If rate compensation is ineffective, reduce speed of rate compensation: "tit for tat"
  - When queues drain, revert to normal rate compensation
- Otherwise compatible at protocol level
  - Future work: PCP sender (receiver) induces TCP receiver (sender) to use PCP

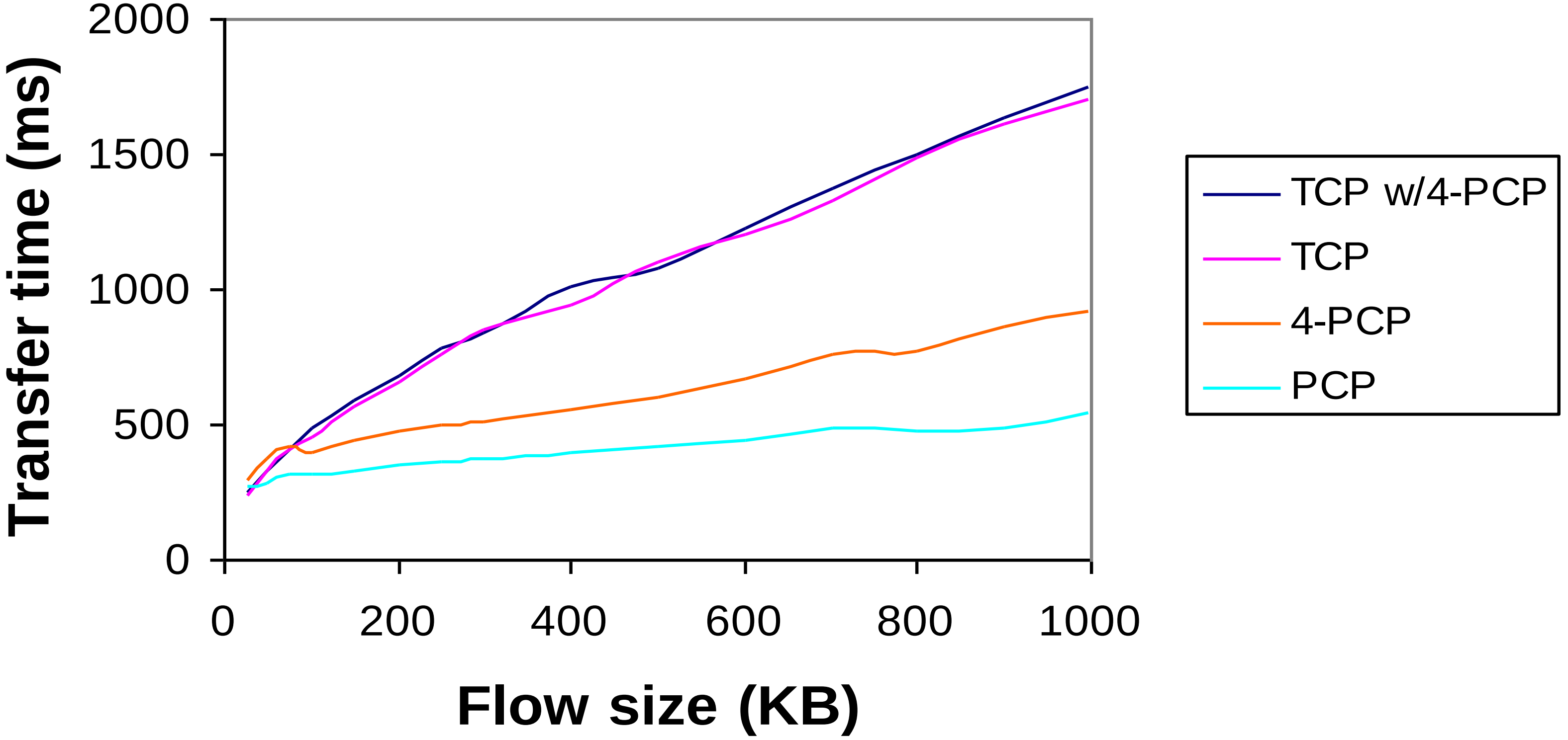
Performance

User-level implementation

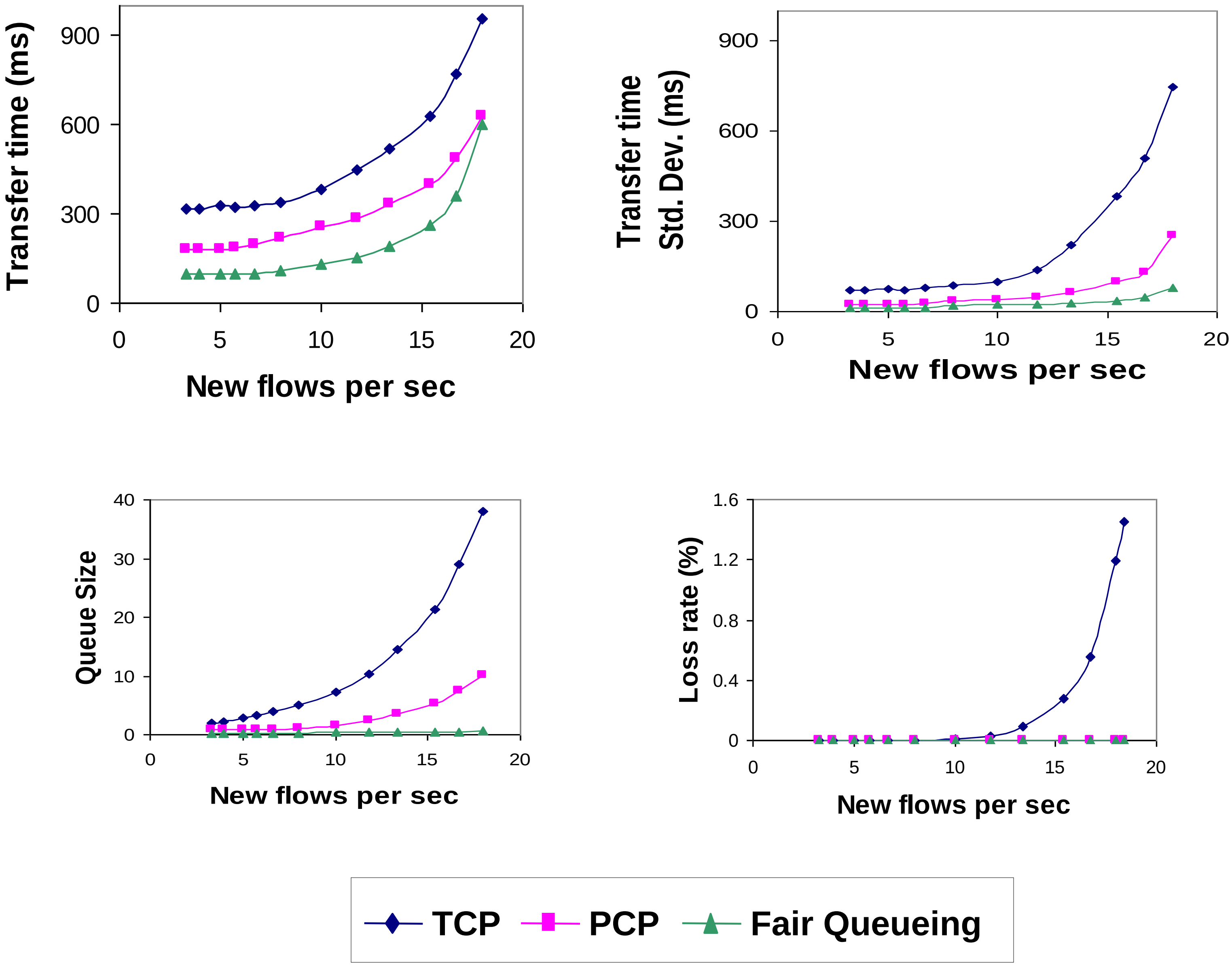
- 250KB transfers between every pair of RON nodes
- PCP vs. TCP vs. four concurrent PCP transmissions



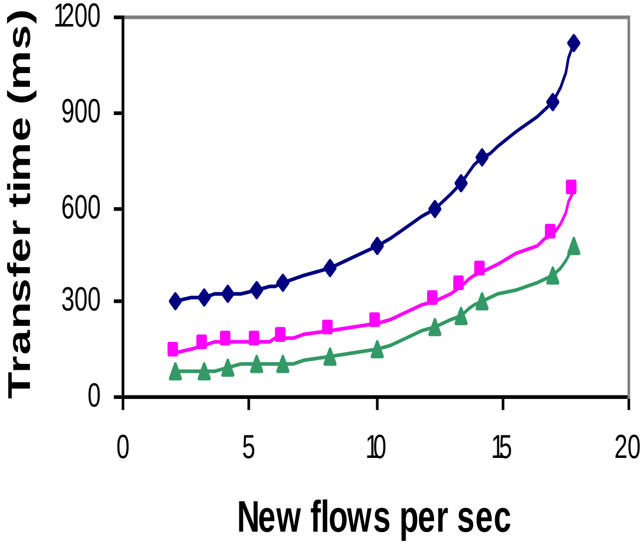
Is PCP Cheating?

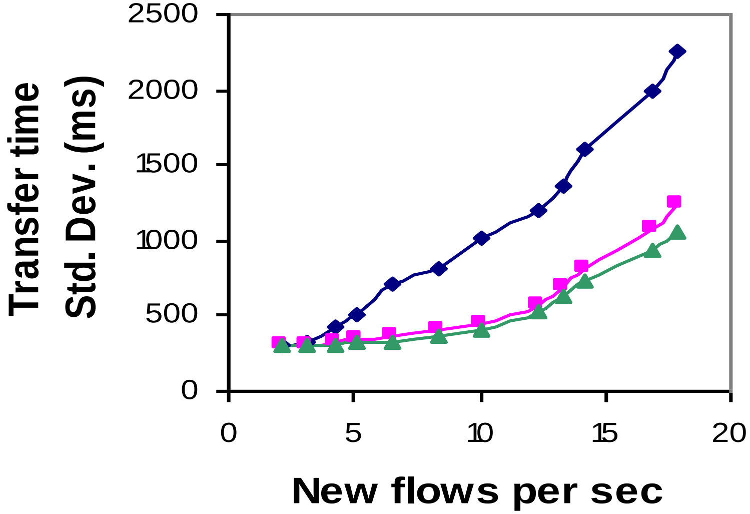


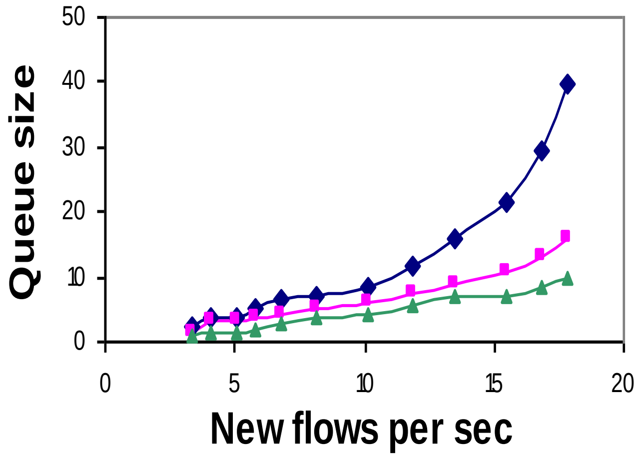
Simulation: Vary Offered Load

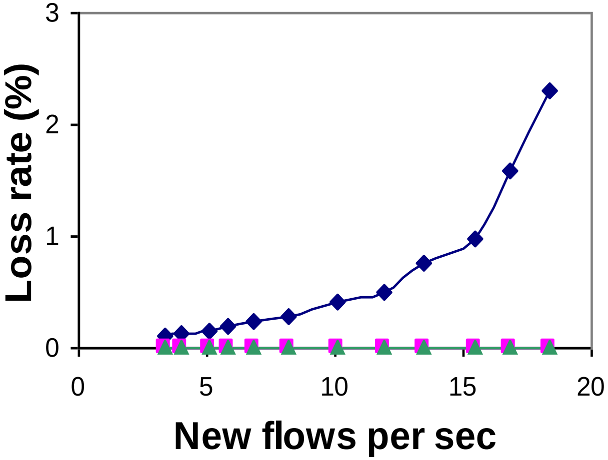


Simulation: Self-Similar Traffic



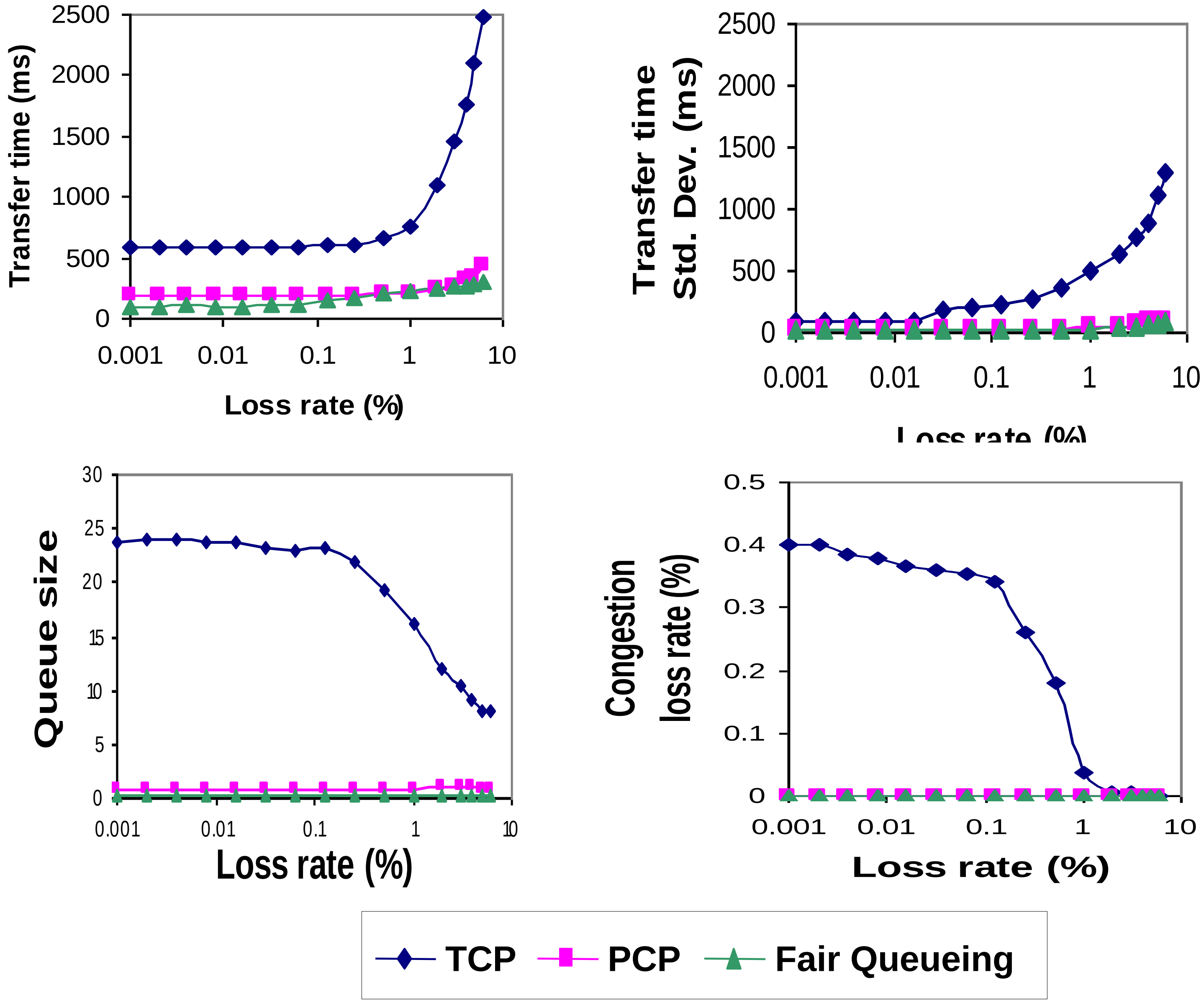




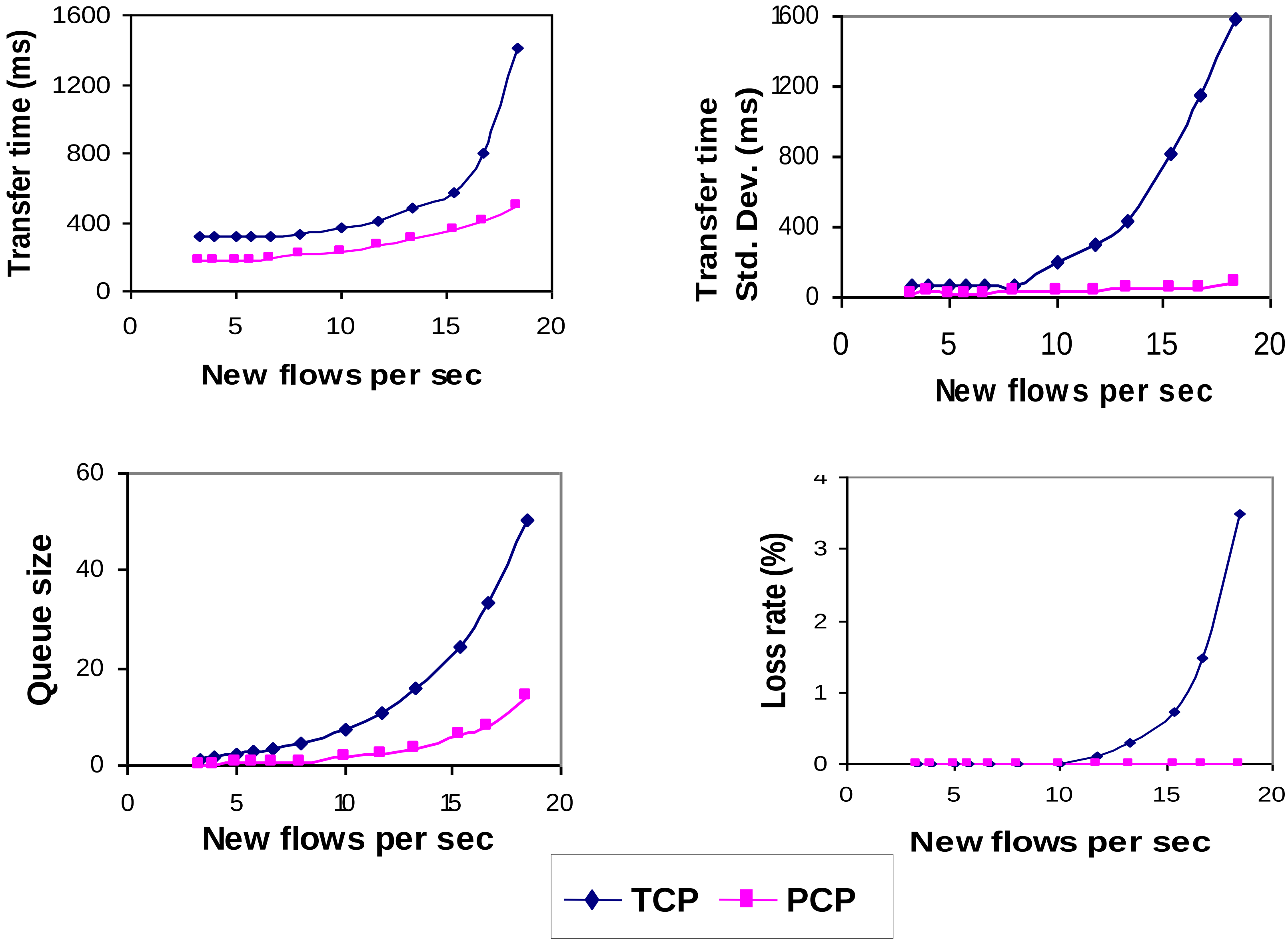


TCP PCP Fair Queueing

Simulation: Transmission Loss



Simulation: Fair-Queued Routers



Related Work

Short circuit TCP's slow-start: TCP Swift Start, Fast Start

Rate pacing: TCP Vegas, FastTCP, RAP

History: TCP Fast Start, MIT Congestion Manager

Delay-based congestion control: TCP Vegas, FastTCP

Available bandwidth: Pathload, Pathneck, IGI, Spruce

Separate efficiency & fairness: XCP

Summary

PCP: near optimal endpoint congestion control

- Emulates centralized control with no special support from network

Better than TCP for today's common case

- Most paths are idle and have predictable performance
- Most flows are short-lived

User-level and kernel implementation available:

http://www.cs.washington.edu/homes/arvind/pcp