

Lecture 22:

Buffering/Scheduling Cont. & Congestion Control

CSE 123: Computer Networks

Aaron Schulman



Overview



- Finish **Router-based** buffering and scheduling
- Introduce **Host-based** congestion control



Fair Queuing

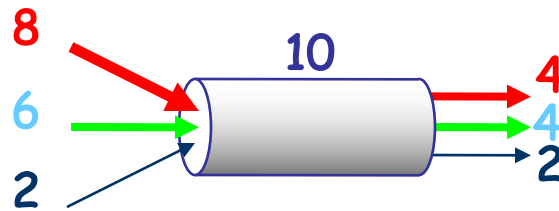
- Maintain a queue for each flow
 - ◆ What is a flow?
 - » We approximate this with the **5 tuple**
 - » (IP src, IP dst, port src, port dst, protocol)
- Implements **max-min fairness**: each flow receives $\min(r_i, f)$, where
 - ◆ r_i – flow arrival rate
 - ◆ f – link fair rate (see next slide)
- **Weighted Fair Queuing** (WFQ) – associate a weight with each flow to divvy bandwidth up non-equally



Fair Rate Computation

- If link congested, compute f such that

$$\sum_i \min(r_i, f) = C$$



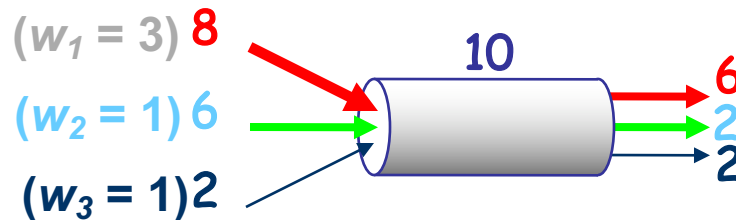
$f = 4$:
 $\min(8, 4) = 4$
 $\min(6, 4) = 4$
 $\min(2, 4) = 2$



Weighted Fair Queuing

- Associate a weight w_i with each flow i
- If link congested, compute f such that

$$\sum_i \min(r_i, f \times w_i) = C$$



$f = 2$:

$$\begin{aligned} \min(8, 2 \times 3) &= 6 \\ \min(6, 2 \times 1) &= 2 \\ \min(2, 2 \times 1) &= 2 \end{aligned}$$

Flow i is guaranteed to be allocated a rate $\geq w_i \cdot C / (\sum_k w_k)$



If $\sum_k w_k \leq C$, flow i is guaranteed to be allocated a rate $\geq w_i$

WFQ Scheduling: Fluid Flow

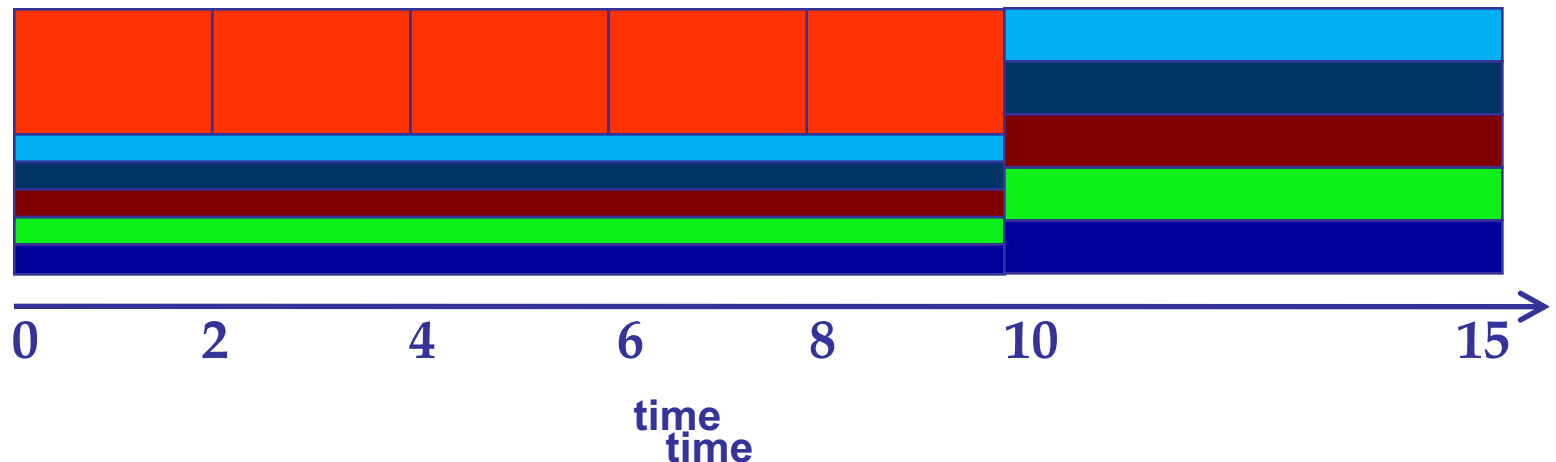
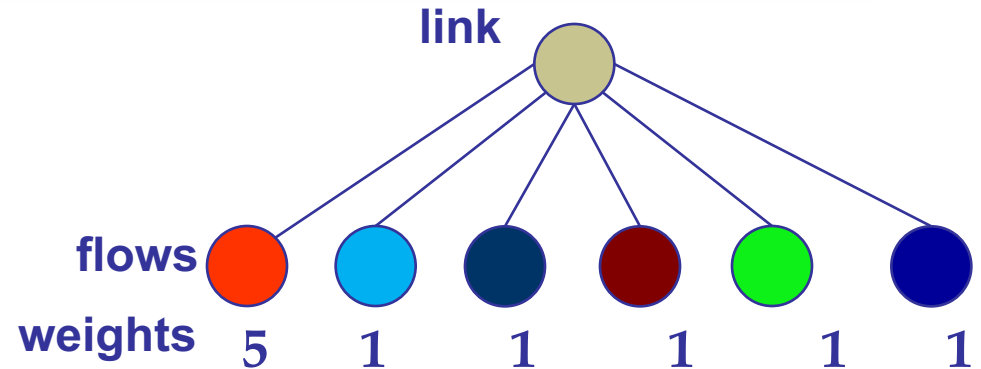


- Simplification: Flows can be served one bit at a time
- WFQ can be implemented using bit-by-bit weighted round robin
 - ◆ During each round from each flow that has data to send, send a number of bits equal to the flow's weight



Fluid Flow Example

- **Red flow** has packets backlogged between time 0 and 10
- Other flows have packets continuously backlogged
- All packets have the same size



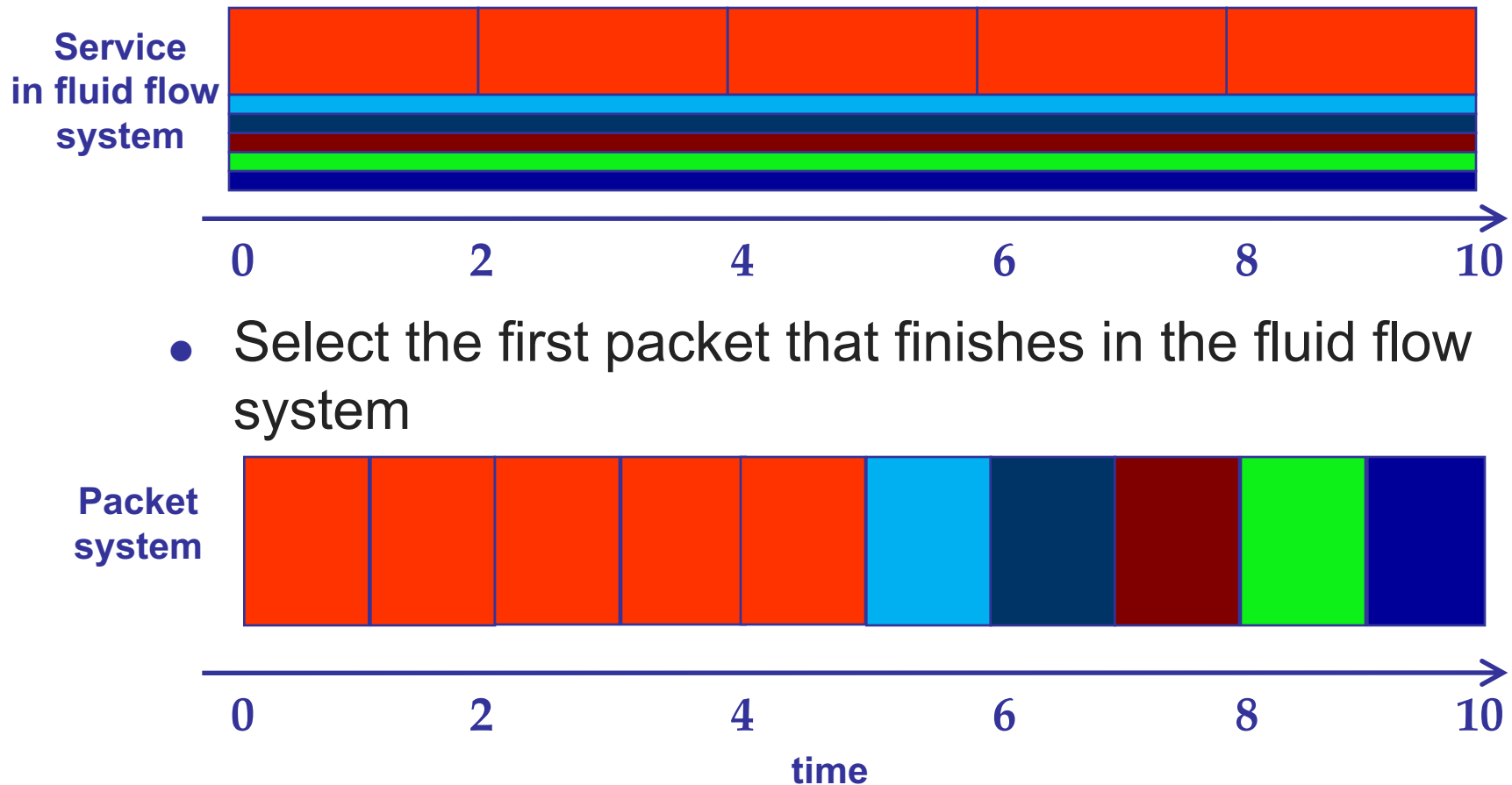
Packet-Based Implementation



- Packet (real) system can't do bit-by-bit splitting: packet transmission cannot be preempted. Why?
- Issue: flows can get more bandwidth by sending bigger packets
- Solution: serve packets in the order in which they would have finished being transmitted in the fluid flow system



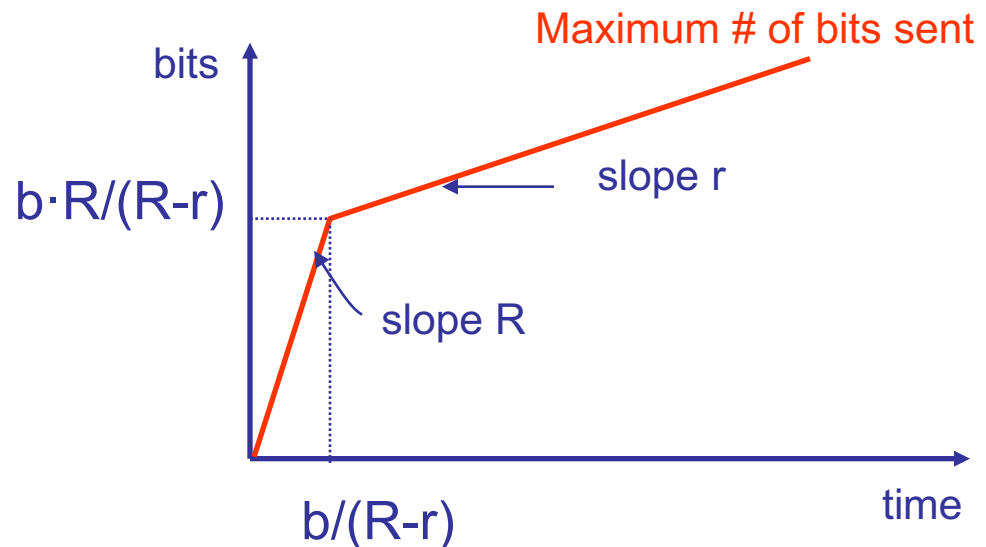
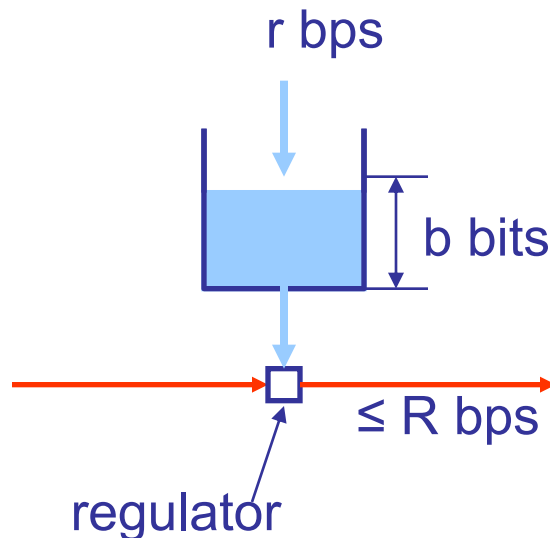
Packet-Based Example



Token Bucket Basics (another way to do scheduling)



- Parameters
 - ◆ r – average rate, i.e., rate at which tokens fill the bucket
 - ◆ b – bucket depth (limits size of burst)
 - ◆ R – maximum link capacity or peak rate (optional parameter)
- A bit can be transmitted only when a token is available



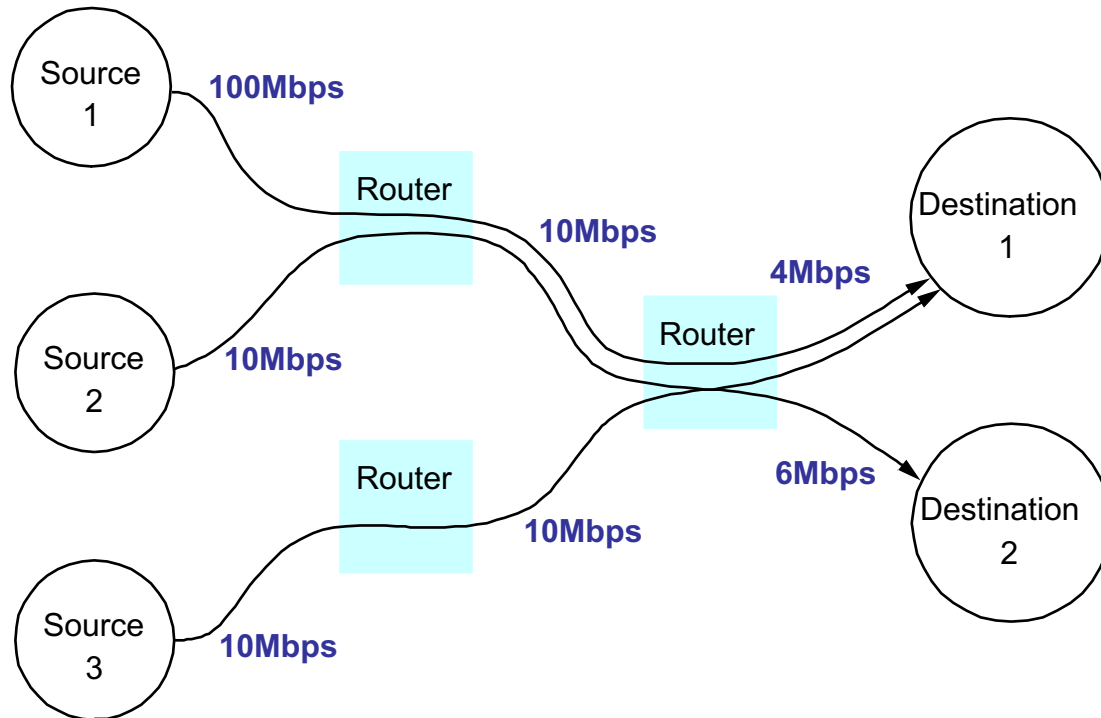
Host-based Congestion Control



- How fast should a sending host transmit data?
 - ◆ Not too fast, not too slow, just right...
- Should not be faster than the sender's share of bw
 - ◆ Bandwidth allocation
- Should not be faster than the network can process
 - ◆ Congestion control
- Congestion control & bandwidth allocation are separate ideas, but frequently combined

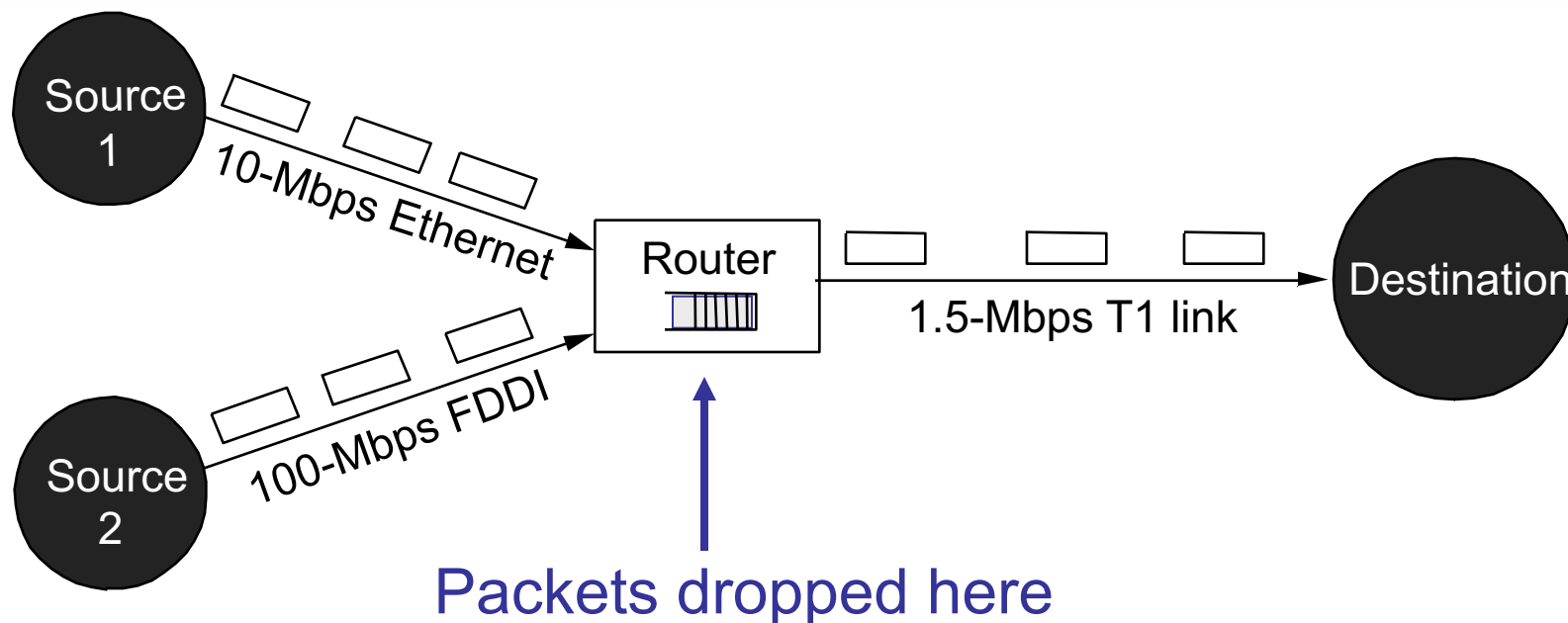


Bandwidth Allocation



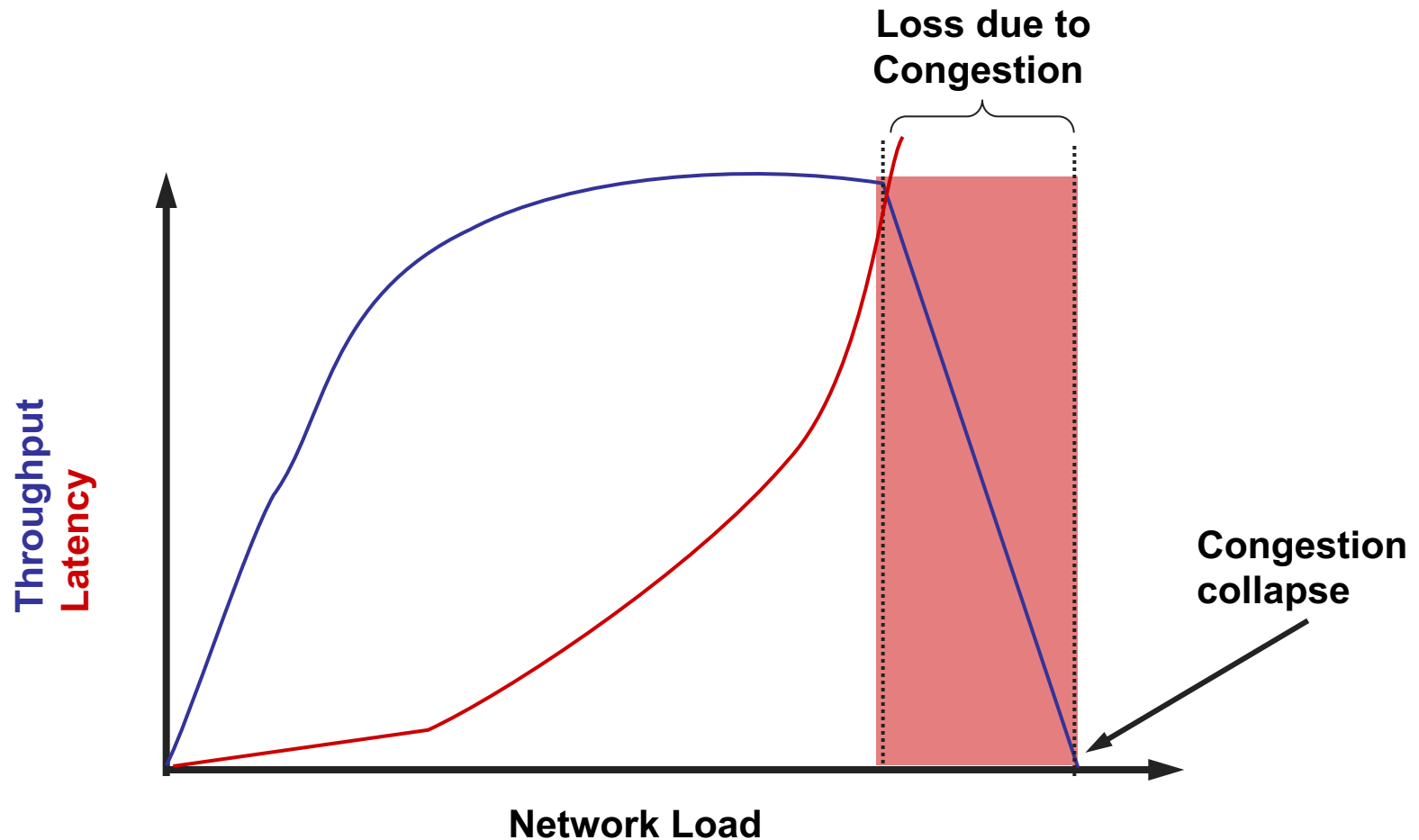
- How much bandwidth should each **flow** from a source to a destination receive when they compete for resources?
 - ◆ What is a “fair” allocation?

Congestion occurs at routers



- Buffer intended to absorb bursts when input rate $>$ output
- But if sending rate is persistently $>$ drain rate, queue builds
- Dropped packets represent wasted work; goodput $<$ throughput

Drop-Tail Queuing Problem...





Congestion Collapse

- Rough definition: “When an increase in network load produces a decrease in useful work”
- Why does it happen?
 - ◆ Sender sends faster than **bottleneck link** speed
 - ◆ Packets queue until dropped
 - ◆ In response to packets being dropped, sender retransmits
 - ◆ All hosts repeat in steady state...



Mitigation Options

- Increase network resources
 - ◆ More buffers for queuing
 - ◆ Increase link speed
 - ◆ Pros/Cons of these approaches?
- **Reduce network load**
 - ◆ Hosts send data more slowly
 - ◆ How much more slowly?
 - ◆ When to slow down?

Designing a Control Scheme



How?

- Open loop
 - ◆ Explicitly reserve bandwidth in the network in advance
- *Closed loop*
 - ◆ Respond to feedback and adjust bandwidth allocation

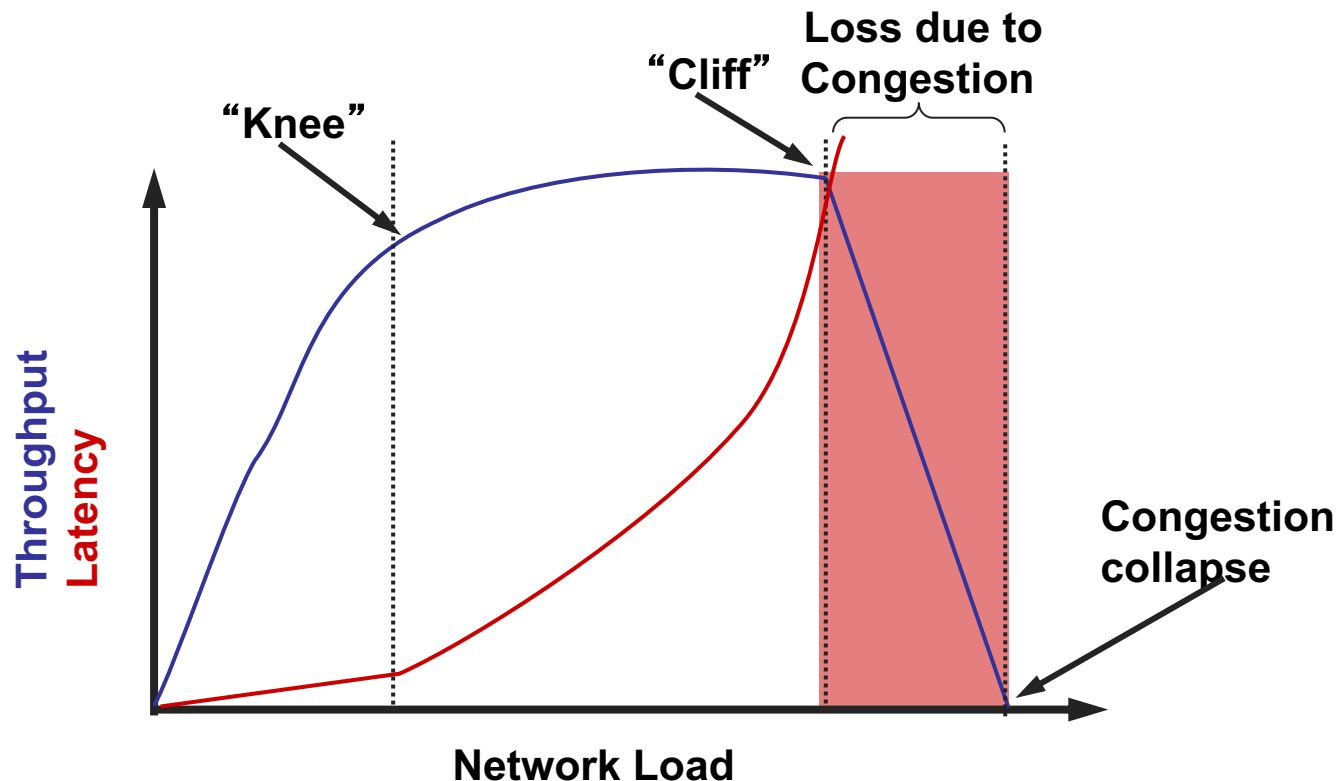
Where?

- Network-based
 - ◆ Network implements and enforces bandwidth allocation
- *Host-based*
 - ◆ Hosts are responsible for controlling their sending rate



Proactive vs. Reactive

- Congestion avoidance: try to stay to the left of the knee
- Congestion control: try to stay to the left of the cliff





Challenges to Address

- How to detect congestion?
- How to limit sending data rate?
- How fast to send?



Detecting Congestion

- Explicit congestion signaling
 - ◆ Source Quench: ICMP message from router to sender
 - ◆ DECEBit / Explicit Congestion Notification (ECN):
 - » Router *marks* packet based on queue occupancy (i.e. indication that packet encountered congestion along the way)
 - » Receiver tells sender if queues are getting too full
- *Implicit congestion signaling*
 - ◆ Packet loss (remember router drop policy?)
 - » Assume congestion is primary source of packet loss
 - » Lost packets indicate congestion
 - ◆ Packet delay
 - » Round-trip time increases as packets queue
 - » Packet inter-arrival time is a function of bottleneck link



Throttling Options

- Sender Window-based (TCP)
 - ◆ Constrain number of outstanding packets allowed in network
 - ◆ Increase window to send faster; decrease to send slower
 - ◆ Pro: Cheap to implement (already have windowing), good failure properties
 - ◆ Con: Creates traffic **bursts** (requires bigger buffers)
- Rate-based (many streaming media protocols)
 - ◆ Two parameters (period, packets)
 - ◆ Allow sending of x packets in period y
 - ◆ Pro: smooth traffic
 - ◆ Con: fine-grained per-connection timers, what if receiver fails?

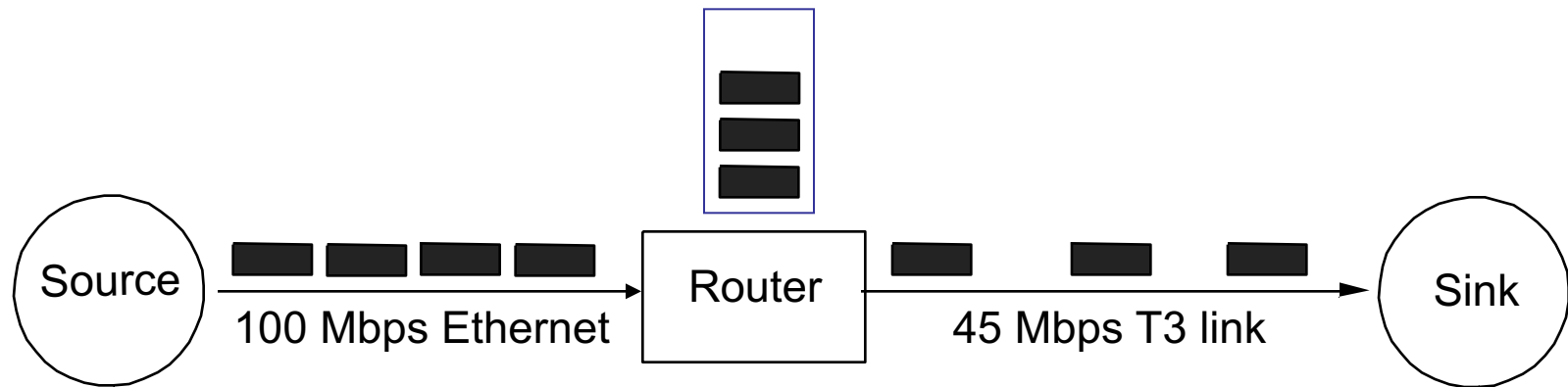


Choosing a Send Rate

- Ideally: Keep equilibrium at “knee” of power curve
 - ◆ Find “knee” somehow
 - ◆ Keep number of packets “in flight” the same
 - ◆ Don’t send a new packet into the network until you know one has left (I.e. by receiving an ACK)
 - ◆ What if you guess wrong, or if bandwidth availability changes?
- Compromise: adaptive approximation
 - ◆ If congestion signaled, reduce sending rate by x
 - ◆ If data delivered successfully, increase sending rate by y
 - ◆ How to relate x and y ? Most choices don’t converge...



TCP's Probing Approach



- Each source independently probes the network to determine how much bandwidth is available
 - ◆ Changes over time, since everyone does this
- Assume that packet loss implies congestion
 - ◆ Since errors are rare; also, requires no support from routers



Basic TCP Algorithm

- Window-based congestion control
 - ◆ Unified congestion control and flow control mechanism
 - ◆ *rwin*: advertised flow control window from receiver
 - ◆ *cwnd*: congestion control window
 - » Estimate of how much outstanding data network can deliver in a round-trip time
 - ◆ Sender can only send $\text{MIN}(rwin, cwnd)$ at any time
- Idea: decrease *cwnd* when congestion is encountered; increase *cwnd* otherwise
 - ◆ Question: how much to adjust?



Congestion Avoidance

- Goal: Adapt to changes in available bandwidth
- Additive Increase, Multiplicative Decrease (AIMD)
 - ◆ Increase sending rate by a constant (e.g. MSS)
 - ◆ Decrease sending rate by a linear factor (e.g. divide by 2)
- Rough intuition for why this works
 - ◆ Let L_i be queue length at time i
 - ◆ In steady state: $L_i = N$, where N is a constant
 - ◆ During congestion, $L_i = N + yL_{i-1}$, where $y > 0$
 - ◆ Consequence: queue size increases multiplicatively
 - » Must reduce sending rate multiplicatively as well



TCP Bandwidth Probing

- TCP uses AIMD to adjust congestion window
 - ◆ Converges to fair share of bottleneck link
 - ◆ Increases modestly in good times
 - ◆ Cuts drastically in bad times
- But what rate should a TCP flow use initially?
 - ◆ Need some initial congestion window
 - ◆ We'd like to TCP to work on all manner of links
 - ◆ Need to span 6+ orders of magnitude, e.g., 10 K to 10 Gbps.
 - ◆ Starting too fast is catastrophic!

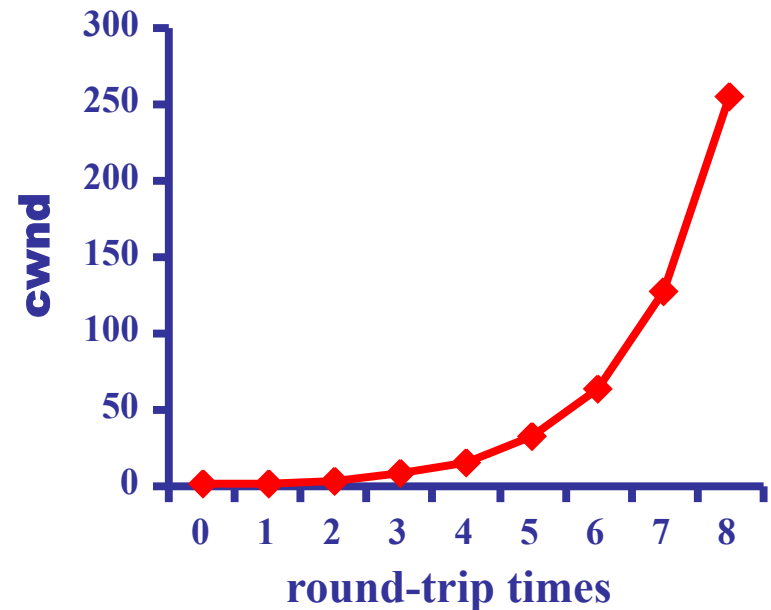
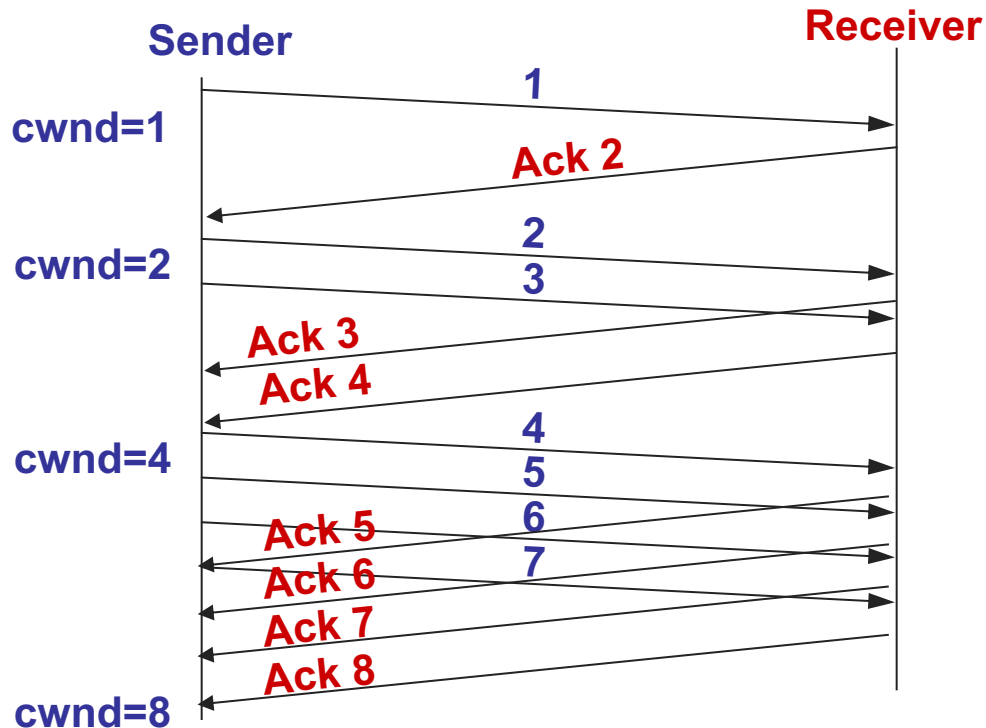


Slow Start

- Goal: quickly find the equilibrium sending rate
- Quickly increase sending rate until congestion detected
 - ◆ Remember last rate that worked and don't overshoot it
- TCP Reno Algorithm:
 - ◆ On new connection, or after timeout, set $cwnd=1$ MSS
 - ◆ For each segment acknowledged, increment $cwnd$ by 1 MSS
 - ◆ If timeout then divide $cwnd$ by 2, and set $ssthresh = cwnd$
 - ◆ If $cwnd \geq ssthresh$ then exit slow start
- Why called slow? Its exponential after all...



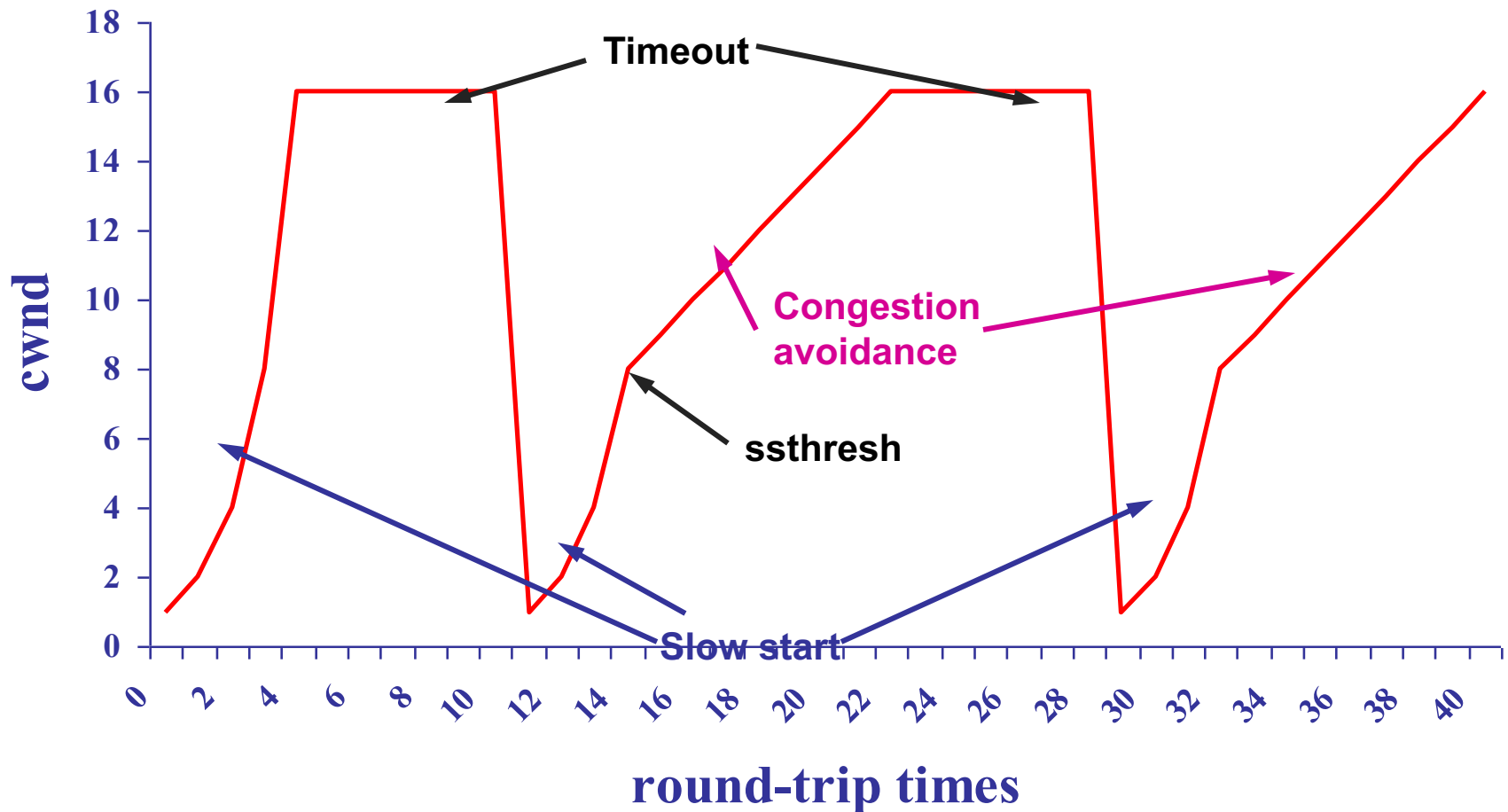
Slow Start Example





Basic Mechanisms

Slow Start + Congestion Avoidance



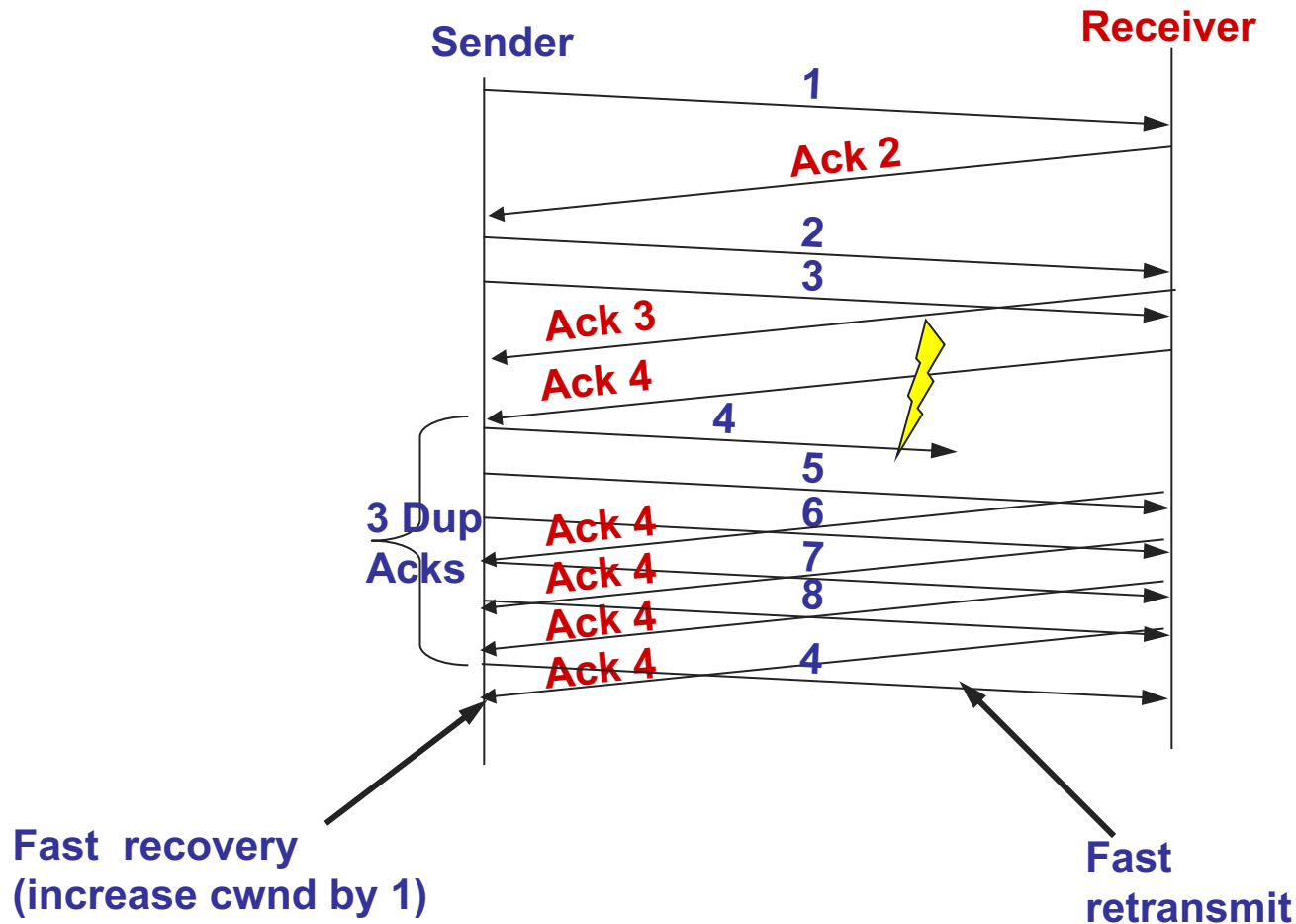
Fast Retransmit & Recovery



- Fast retransmit
 - ◆ Timeouts are slow (default often 200 ms or 1 second)
 - ◆ When packet is lost, receiver still ACKs last in-order packet
 - ◆ Use 3 duplicate ACKs to indicate a loss; detect losses quickly
 - » Why 3? When wouldn't this work?
- Fast recovery
 - ◆ Goal: avoid stalling after loss
 - ◆ If there are still ACKs coming in, then no need for slow start
 - ◆ If a packet has made it through -> we can send another one
 - ◆ Divide *cwnd* by 2 after fast retransmit
 - ◆ Increment *cwnd* by 1 MSS for each additional duplicate ACK

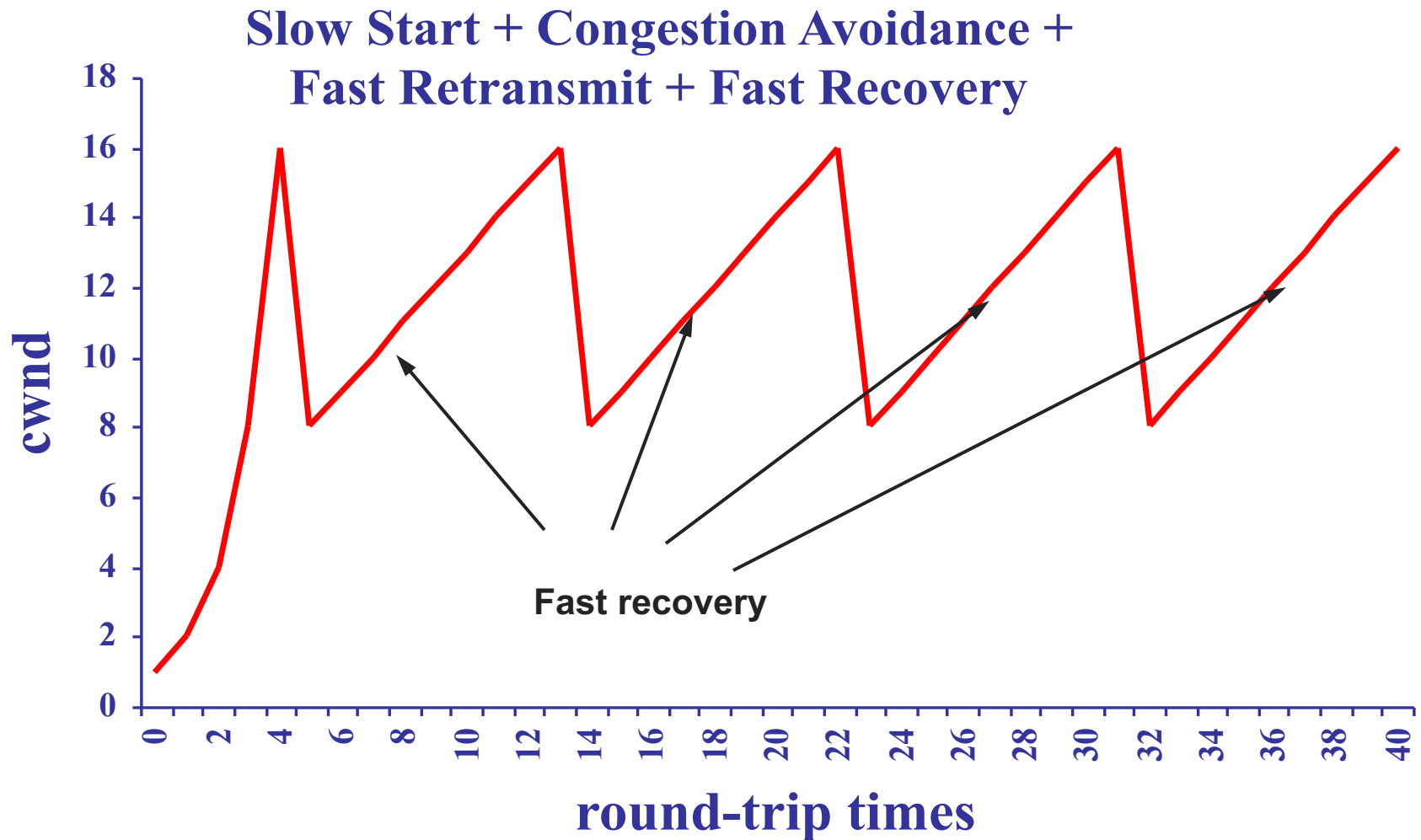


Fast Retransmit Example





More Sophistication





Short Connections

- Short connection: only contains a few pkts
- How do short connections and Slow-Start interact?
 - ◆ What happens when a packet is lost during Slow-Start?
 - ◆ What happens when the SYN is dropped?
- Bottom line: Which packet gets dropped matters a lot
 - ◆ SYN
 - ◆ Slow-Start
 - ◆ Congestion avoidance
- Do you think most flows are short or long?
- Do you think most traffic is in short flows or long flows?

Open Issues



- TCP is designed around the premise of cooperation
 - ◆ What happens to TCP if it competes with a UDP flow?
 - ◆ What if we divide *cwnd* by 3 instead of 2 after a loss?
- There are a bunch of magic numbers
 - ◆ Decrease by 2x, increase by $1/cwnd$, 3 duplicate acks, initial timeout = 3 seconds, etc.
- But overall it works really well!
 - ◆ Still being constantly tweaked...

TCP CC Summary



- TCP Probes the network for bandwidth, assuming that loss signals congestion
- The congestion window is managed with an additive increase/multiplicative decrease policy
 - ◆ It took fast retransmit and fast recovery to get there
 - ◆ Fast recovery keeps pipe “full” while recovering from a loss
- Slow start is used to avoid lengthy initial delays
 - ◆ Ramp up to near target rate, then switch to AIMD



For next time...

- Read P&D 2.1
- Keep going on Project 3 due Thursday!