

# Congestion Control (cont'd)

# Packets vs. Bytes

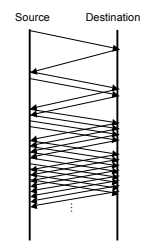
- TCP window sizes are in bytes
- Congestion control works on packets
  - Increase by 1 packet every RTT
  - $P_{cwnd} = cwnd / MSS$
  - $P_{cwnd} = P_{cwnd} + 1/P_{cwnd}$  on each ACK
- Multiply by MSS to get byte-sized formula
  - $cwnd = cwnd + MSS * (MSS / cwnd)$  on each ACK
  - Increase by 1 MSS every RTT

# TCP Start Up

- How do we set initial window size?
- Additive increase too slow
- Example: DSL line
  - RTT=100ms, MSS=1500b, BW=200KB/s
  - After 1 RTT, rate is 15 KB/s
  - After 1s, rate is 150 KB/s
  - Takes 8.3s to transfer 500 KB file

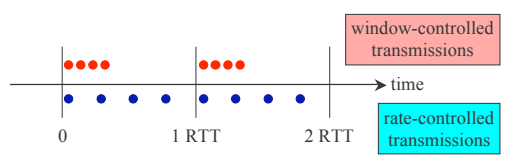
# Slow Start

- Objective
  - Determine initial available capacity
- Idea
  - Begin with `CongestionWindow` = 1 packet
  - Double `CongestionWindow` each RTT
    - Increment by 1 packet for each ACK
  - Continue increasing until loss, then switch to AIMD
- Result
  - Exponential growth
  - Slower than all at once



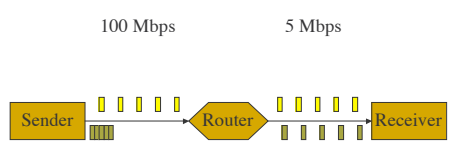
# Window rate control

- Congestion window ensures *average* rate is  $cwnd / RTT$ 
  - Instantaneous rate may be larger



# ACK clocking

- ACK clocking spreads out bursts
  - Packets sent in a burst arrive spread out
  - ACKs follow the timing of received rate



## ACK clocking

- ACK clocking spreads out bursts
  - Packets sent in a burst arrive spread out
  - ACKs follow the timing of received rate
  - New sending rate follows ACK rate

100 Mbps                      5 Mbps

## Slow start

- ACK clocking, with 2 packets per ACK

## Slow start

- ACK clocking, with 2 packets per ACK

## Slow start

- ACK clocking, with 2 packets per ACK

NB: There's a proposed alternative to slow-start that uses ACK clocking

## TCP Timeout

## Timeout Handling

- Cumulative ACK opens up entire window
  - Do we send entire window all at once?
  - No ACKs to clock transmission
- Use slow-start to recover ACK clock
  - Reset cwnd to 1 (packet)
  - Use exponential increase (add 1 packet to cwnd for every ACK)

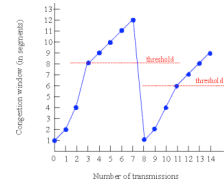
## Congestion Threshold

- New variable: Congestion Threshold
  - Target window size
  - Estimate network capacity
- If  $cwnd < cthresh$ , increase exponentially
  - slow start
- If  $cwnd > cthresh$ , increase linearly
  - additive increase
- Initially,  $ctresh = \text{max window}$
- At loss,  $ctresh = 1/2 \text{ cwnd}$



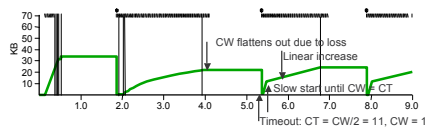
## Slow Start

- Initial values
  - $ctresh = 8$
  - $cwnd = 1$
- Loss after transmission 7
  - $cwnd$  currently 12
  - Set  $ctresh = cwnd/2$
  - Set  $cwnd = 1$



## Slow Start

- Example trace of CongestionWindow

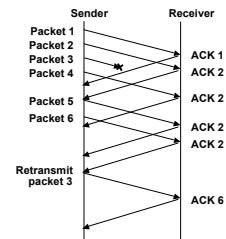


- Problem
  - Have to wait for timeout
  - Can lose half CongestionWindow of data



## Fast Retransmit and Fast Recovery

- Problem
  - Coarse-grain TCP timeouts lead to idle periods
- Solution
  - Fast retransmit: use duplicate ACKs to trigger retransmission

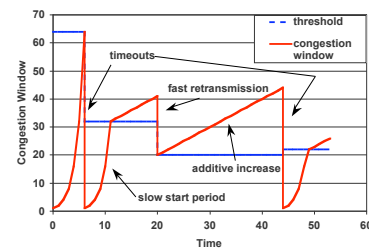


## Fast Retransmit and Fast Recovery

- Send ACK for each segment received
- When duplicate ACK's received
  - Resend lost segment immediately
  - Do not wait for timeout
  - In practice, retransmit on 3rd duplicate
- Fast recovery
  - When fast retransmission occurs, skip slow start
  - Congestion window becomes 1/2 previous
  - Start additive increase immediately



## TCP Congestion Window Trace



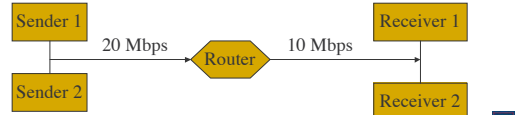
## TCP Congestion Control Summary

- Congestion control mechanisms
  - Timeouts
  - RTT estimation
  - Congestion window
  - Slow start
  - Fast retransmit

I

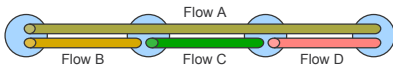
## Fairness

- TCP congestion control is fair
  - Both senders will settle at around 5 Mbps
  - Intuition: flows using more bandwidth more likely to experience loss



I

## What's Fair?



Which is more fair:

Globally Fair:  $F_a = \text{Capacity}/4$ ,  $F_b = F_c = F_d = 3\text{Capacity}/4$

or

Locally Fair:  $F_a = F_b = F_c = F_d = \text{Capacity}/2$

This is the so-called "max-min fair" rate allocation. The minimum rate is maximized.

I

## TCP fairness

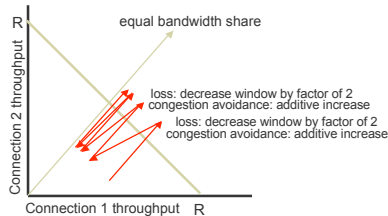
- TCP is "RTT-fair"
  - On each congested link, host gets shared of bandwidth proportional to RTT
- Intuition: during additive increase, each host adds one new packet every RTT
  - If RTT twice as large, additive increase is half as fast
- Is this closer to globally fair or locally fair?

I

## Why is TCP fair?

Two competing sessions:

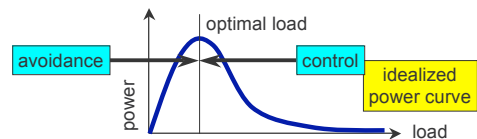
- Additive increase improves fairness
- Multiplicative decrease preserves fairness



I

## Congestion Avoidance

- Control vs. avoidance
  - Control: minimize impact of congestion when it occurs
  - Avoidance: avoid producing congestion
- In terms of operating point limits



I

## Congestion Avoidance

- TCP's strategy
  - Repeatedly *cause* congestion
  - Control it once it happens
- Alternative Strategy
  - Predict when congestion is about to happen and reduce the rate at which hosts send data just before packets start being discarded
  - Congestion avoidance, as compared to congestion control
- Approaches
  - Routers *implement* CA (ATM, RSVP)
  - Routers *help* end-hosts implement CA (DECbit, RED)
  - End-hosts do it themselves (TCP Vegas)



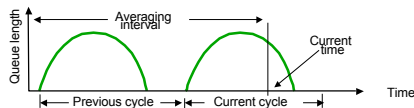
## DECbit (Destination Experiencing Congestion Bit)

- Developed for the Digital Network Architecture
- Basic idea
  - One bit allocated in packet header
  - Any router experiencing congestion sets bit
  - Source adjusts rate based on bits
- Note that responsibility is shared
  - Routers identify congestion
  - Hosts act to avoid congestion



## DECbit

- Router
  - Monitors length over last busy + idle cycle
  - Sets congestion bit if average queue length is greater than 1 when packet arrives
  - Attempts to balance throughput against delay
    - smaller values result in more idle time
    - larger values result in more queuing delay



## DECbit

- End Hosts
  - Destination echoes congestion bit back to source
  - Source records how many packets resulted in set bit
  - If less than 50% of last window had bit set
    - Increase **CongestionWindow** by 1 packet
  - If 50% or more of last window had bit set
    - Decrease **CongestionWindow** by 1/8th
- Note:
  - Techniques used in DECbit known as explicit congestion notification (ECN)
  - Proposal to add ECN bit to TCP in progress



## Router-Based Congestion Avoidance

- Random Early Detection (RED) gateways
  - Developed for use with TCP
  - Basic idea
    - Implicit rather than explicit notification
    - When a router is "almost" congested
    - Drop packets randomly
  - Responsibility is again shared
    - Router identifies, host acts
    - Relies on TCP's response to dropped packets



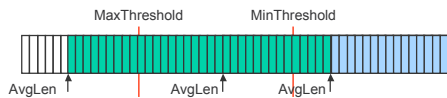
## Random Early Detection (RED)

- Hosts
  - Implement TCP congestion control
  - Back off when a packet is dropped
- Routers
  - Compute average queue length (exponential moving average)
    - $AvgLen = (1 - Weight) * AvgLen + Weight * SampleLen$
    - $0 < Weight < 1$  (usually 0.002)
    - **SampleLen** is queue length at packet arrival time



## RED – Dropping Policy

- If  $AvgLen \leq MinThreshold$ 
  - Enqueue packet
- If  $MinThreshold < AvgLen < MaxThreshold$ 
  - Calculate P and drop arriving packet with probability P
- If  $MaxThreshold \leq AvgLen$ 
  - Drop arriving packet

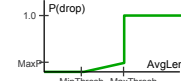


## RED – Dropping Probability

- Computing P
  - P is a function of AvgLen and Count
  - Count is the number of packets that have arrived since last reset
  - Reset happens when either a packet is dropped or AvgLen is above MaxThreshold

$$TempP = \frac{MaxP * (AvgLen - MinThreshold)}{MaxThreshold - MinThreshold}$$

$$P = \frac{TempP}{(1 - count * TempP)}$$



## RED Parameters

- MaxP is typically set to 0.02
  - When the average queue size is halfway between the two thresholds, the gateway drops roughly one out of 50 packets.
- MinThreshold is typically max/2
- Choosing parameters
  - Carefully tuned to maximize power function
  - Confirmed through simulation
  - But answer depends on accuracy of traffic model



## Tuning RED

- Probability of dropping a particular flow's packet(s)
  - Roughly proportional to the that flow's current share of the bandwidth
- If traffic is bursty
  - MinThreshold should be sufficiently large to allow link utilization to be maintained at an acceptably high level
  - If no buffer space is available, RED uses Tail Drop
- Difference between two thresholds
  - Should be larger than the typical increase in the calculated average queue length in one RTT
  - Setting MaxThreshold to twice MinThreshold is reasonable for traffic on today's Internet



## Source-Based Congestion Avoidance

- Idea
  - Source watches for some sign that some router's queue is building up and congestion will happen soon
- Examples
  - RTT is growing
  - Sending rate flattens

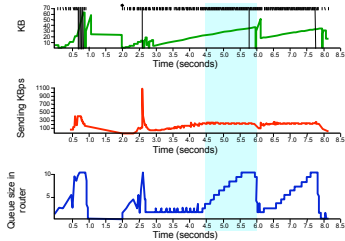


## Source-Based Congestion Avoidance

- Observe RTT
  - If current RTT is greater than average of minRTT and maxRTT, decrease congestion window by one-eighth
- Observe RTT and Window Size
  - Adjust window once every two RTT
    - If  $(CurrWindow - OldWindow) * (CurrRTT - OldRTT) > 0$ , decrease window by one-eighth, otherwise increase window by one MSS
- Observe sending rate
  - Increase window and compare throughput to previous value
- Observe throughput
  - Compare expected throughput with observed throughput
  - TCP Vegas



## TCP Vegas



## TCP Vegas

- Basic idea
  - Watch for signs of queue growth
  - In particular, difference between
    - increasing congestion window
    - stable throughput (presumably at capacity)
  - Keep just enough “extra data” in the network
    - Time to react if bandwidth decreases
    - Data available if bandwidth increases

## TCP Vegas

- Implementation
  - Estimate uncongested RTT
    - baseRTT = minimum measured RTT
    - Expected throughput = congestion window / baseRTT
  - Measure throughput each RTT
    - Mark time of sending distinguished packet
    - Calculate data sent between send time and receipt of ACK

## TCP Vegas

- Act to keep the difference between estimated and actual throughput in a specified range
  - Below minimum threshold
    - Increase congestion window
  - Above maximum threshold
    - Decrease congestion window
- Additive decrease used only to avoid congestion
- Want between 1 and 3 packets of extra data (used to pick min/max thresholds)

## TCP Vegas Algorithm

- Let BaseRTT be minimum of all measured RTTs
  - Commonly the RTT of the first packet
- If not overflowing the connection, then
  - ExpectRate = CongestionWindow/BaseRTT
- Source calculates sending rate (ActualRate) once per RTT
- Source compares ActualRate with ExpectRate
  - Diff = ExpectRate - ActualRate
  - if Diff < a
    - Increase CongestionWindow linearly
  - else if Diff > b
    - Decrease CongestionWindow linearly
  - else
    - Leave CongestionWindow unchanged

## TCP Vegas Algorithm

- Parameters
  - $\alpha = 1$  packet
  - $\beta = 3$  packets
- Even faster retransmit
  - Keep fine-grained timestamps for each packet
  - Check for timeout on first duplicate ACK

