

TCP Congestion Control

- Idea
 - assumes best-effort network (FIFO or FQ routers) each source determines network capacity for itself
 - uses implicit feedback
 - ACKs pace transmission (*self-clocking*)
- Challenge
 - determining the available capacity in the first place
 - adjusting to changes in the available capacity

Housekeeping

- Pick up folders for exam study
- Exam next Friday, Nov. 16
 - Chapt. 5, 6 (not 6.5) and 7.1
- Should have Hwk 4 ready to hand back next Wed.
- Hwk 4 solutions will be posted prior to that

Additive Increase/Multiplicative Decrease

- Objective: adjust to changes in the available capacity
- New state variable per connection: **CongestionWindow**
 - limits how much data source has in transit

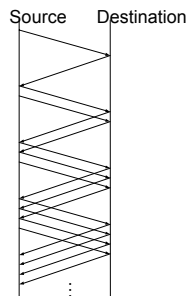
$$\text{MaxWin} = \text{MIN}(\text{CongestionWindow}, \text{AdvertisedWindow})$$
$$\text{EffWin} = \text{MaxWin} - (\text{LastByteSent} - \text{LastByteAcked})$$

- Idea:
 - increase **CongestionWindow** when congestion goes down
 - decrease **CongestionWindow** when congestion goes up

AIMD (cont)

- Question: how does the source determine whether or not the network is congested?
- Answer: a timeout occurs
 - timeout signals that a packet was lost
 - packets are seldom lost due to transmission error
 - lost packet implies congestion

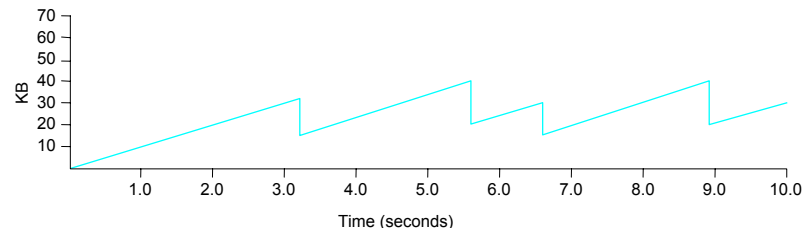
AIMD (cont)



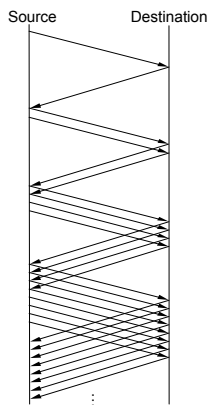
- Algorithm
 - increment **CongestionWindow** by one packet per RTT (*linear increase*)
 - divide **CongestionWindow** by two whenever a timeout occurs (*multiplicative decrease*)
- In practice: increment a little for each ACK
 - Increment = MSS * (MSS/CongestionWindow)**
 - CongestionWindow += Increment**

AIMD (cont)

- Trace: sawtooth behavior



Slow Start

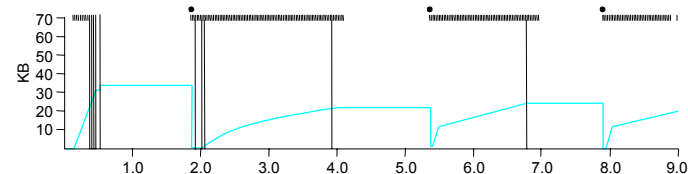


- Objective: determine the available capacity in the first
- Idea:
 - begin with **CongestionWindow** = 1 packet
 - double **CongestionWindow** each RTT (increment by 1 packet for each ACK)

Slow Start (cont)

- Exponential growth, but slower than all at once
- Used...
 - when first starting connection
 - when connection goes dead waiting for timeout

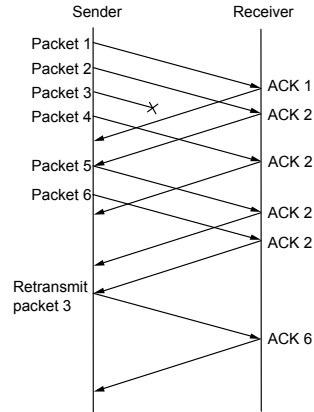
- Trace



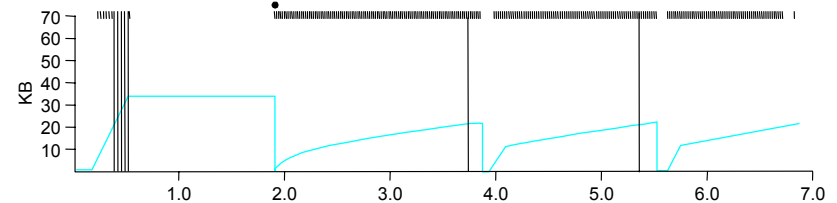
- Problem: lose up to half a **CongestionWindow**'s worth of data

Fast Retransmit and Fast Recovery

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



Results



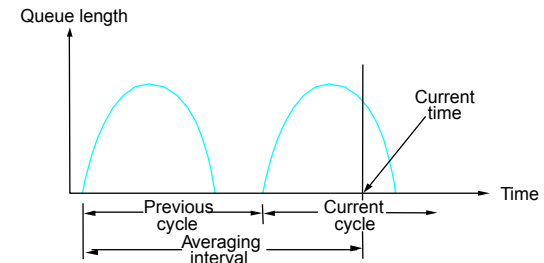
- Fast recovery
 - skip the slow start phase
 - go directly to half the last successful **CongestionWindow (ssthresh)**

Congestion Avoidance

- TCP's strategy
 - control congestion once it happens
 - repeatedly increase load in an effort to find the point at which congestion occurs, and then back off
- Alternative strategy
 - predict when congestion is about to happen
 - reduce rate before packets start being discarded
 - call this congestion *avoidance*, instead of congestion *control*
- Two possibilities
 - router-centric: DECbit and RED Gateways
 - host-centric: TCP Vegas

DECbit

- Add binary congestion bit to each packet header
- Router
 - monitors average queue length over last busy+idle cycle



- set congestion bit if average queue length > 1
- attempts to balance throughput against delay

End Hosts

- Destination echoes bit back to source
- Source records how many packets resulted in set bit
- If less than 50% of last window's worth had bit set
 - increase `CongestionWindow` by 1 packet
- If 50% or more of last window's worth had bit set
 - decrease `CongestionWindow` by 0.875 times

Packet Pair

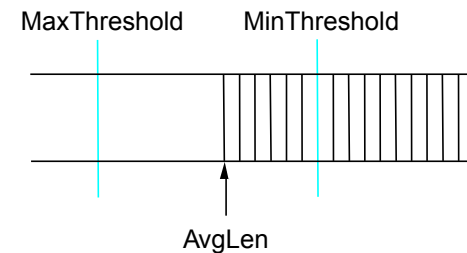
- Send two packets back-to-back
 - Measure how far apart the acks are
 - Gives estimate of the available bandwidth
 - Example:
 - Ping www.yahoo.com with 3000 byte packet
 - First two fragments arrive 1.2 ms apart
 - Estimated bandwidth 1500 bytes/1.2ms =

Random Early Detection (RED)

- Notification is implicit
 - just drop the packet (TCP will timeout)
 - could make explicit by marking the packet
- Early random drop
 - rather than wait for queue to become full, drop each arriving packet with some *drop probability* whenever the queue length exceeds some *drop level*

RED Details

- Compute average queue length
$$\text{AvgLen} = (1 - \text{Weight}) * \text{AvgLen} + \text{Weight} * \text{SampleLen}$$
$$0 < \text{Weight} < 1 \text{ (usually } 0.002)$$
$$\text{SampleLen} \text{ is queue length each time a packet arrives}$$



RED Details (cont)

- Two queue length thresholds

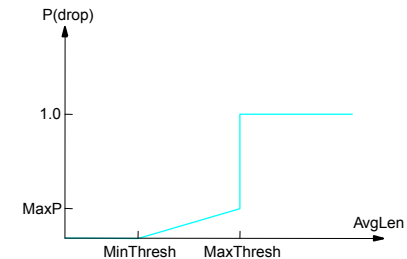
```
if AvgLen <= MinThreshold then
  enqueue the packet
if MinThreshold < AvgLen < MaxThreshold then
  calculate probability P
  drop arriving packet with probability P
if MaxThreshold <= AvgLen then
  drop arriving packet
```

RED Details (cont)

- Computing probability P

$$\text{TempP} = \text{MaxP} * (\text{AvgLen} - \text{MinThreshold}) / (\text{MaxThreshold} - \text{MinThreshold})$$
$$P = \text{TempP} / (1 - \text{count} * \text{TempP})$$

- Drop Probability Curve

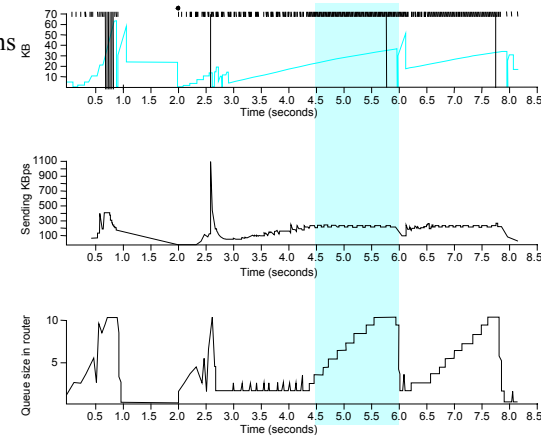


Tuning RED

- Probability of dropping a particular flow's packet(s) is roughly proportional to the share of the bandwidth that flow is currently getting
- MaxP** is typically set to 0.02, meaning that when the average queue size is halfway between the two thresholds, the gateway drops roughly one out of 50 packets.
- If traffic is bursty, then **MinThreshold** should be sufficiently large to allow link utilization to be maintained at an acceptably high level
- 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
- Penalty Box for Offenders

TCP Vegas

- Idea: source watches for some sign that router's queue is building up and congestion will happen too; e.g.,
 - RTT grows
 - sending rate flattens



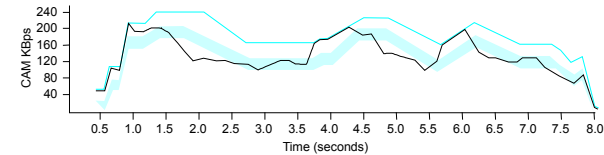
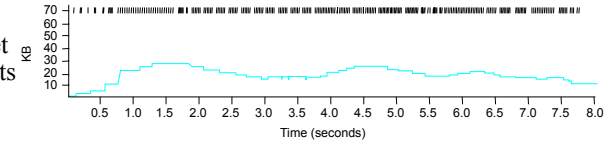
Algorithm

- Let **BaseRTT** be the minimum of all measured RTTs (commonly the RTT of the first packet)
- If not overflowing the connection, then
$$\text{ExpectRate} = \text{CongestionWindow} / \text{BaseRTT}$$
- Source calculates sending rate (**ActualRate**) once per RTT
- Source compares **ActualRate** with **ExpectRate**

```
Diff = ExpectedRate - ActualRate
if Diff <  $\alpha$ 
    increase CongestionWindow linearly
else if Diff >  $\beta$ 
    decrease CongestionWindow linearly
else
    leave CongestionWindow unchanged
```

Algorithm (cont)

- Parameters
 - $\alpha = 1$ packet
 - $\beta = 3$ packets



- Even faster retransmit
 - keep fine-grained timestamps for each packet
 - check for timeout on first duplicate ACK