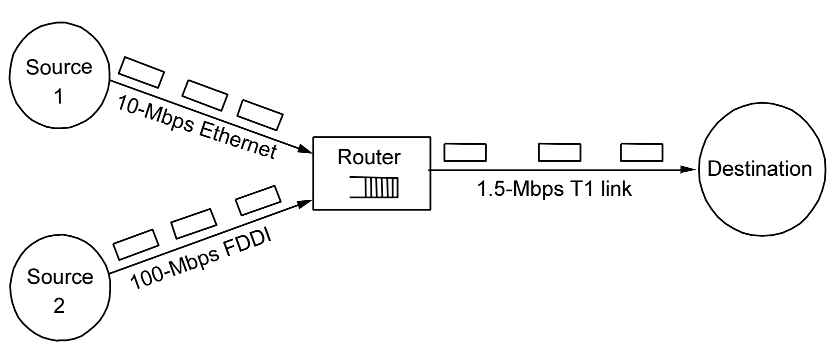
Congestion Control

Resource allocation and congestion control problem

- Problem 1: Resource allocation
  - How to effectively and fairly allocate resources among competing users?
  - **resources = bandwidth of links + buffers on the routers**
- Problem 2: Congestion
  - How to react when queues overflow and packets have to be dropped?
- Allocation vs. congestion control: two sides of the same coin
  - can pre-allocate resources to avoid congestion
  - can control congestion if (and when) it occurs



Where to implement?

- Network initiated resource allocation
  - proactive approach
  - may be difficult (resources distributed throughout the network, need to schedule multiple links connecting a series of routers)
- Easier approach
  - let packet sources send as much data as they want, and recover from congestion when it happens
  - reactive approach
- Solution in the middle: two points of implementation
  - hosts at the edges of the network (transport protocol)
  - routers inside the network (queuing discipline)

Outline

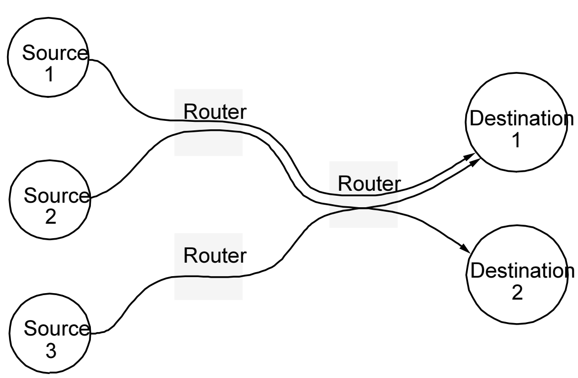
- Congestion control and resource allocation
- Queuing disciplines
- TCP congestion control algorithm
- Congestion avoidance at routers and hosts

Resource allocation

- Resource allocation and congestion control are active areas of research
  - not isolated to one level of protocol hierarchy
  - implemented partially in routers inside the network (queuing mechanisms), partially in transport protocols (TCP, etc.)
- Terminology:
  - resource allocation = network elements try to meet the competing demands for link bandwidth and buffer space (main network resources)
  - congestion control = efforts made by network nodes to prevent or respond to overload conditions, keeping senders from sending too much data into a network
  - fairness = try to share the pain among all users, rather than causing great pain to a few
  - flow control = keeping a fast sender from overrunning a slow receiver

Framework

- Network model
  - packet switched network
  - bottle neck link(s) exist and traffic needs control
- Underlying service model
  - best-effort (assume for now)
  - multiple qualities of service, Differentiated Services (later)
- Connectionless flows
  - sequence of packets sent between source/destination pair
  - maintain soft state at the routers
  - flow either implicitly or explicitly established



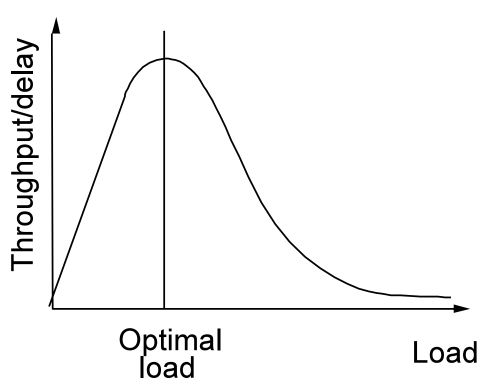
Framework (cont.)

- Taxonomy:
  - router-centric versus host-centric, addressing the problem
    - inside the network (routers)
      - router decides when packets are forwarded and selects dropped packets (drop policy)
    - on the edges of the network (hosts)
      - hosts observe network conditions and behave accordingly
  - reservation-based versus feedback-based
    - hosts ask reservations, routers allocate enough resources
    - no reservations, end hosts adjust sending rates based on feedback
  - window-based versus rate-based

- Above not mutually exclusive characterizations, for example:
  - current Internet offers best-effort service ⇒ feedback based ⇒ primarily host based, window based
  - NextGen Internet offers QoS ⇒ combination of reservation and feedback based ⇒ combination of host and router centric

Evaluation

- Common criteria
  - fairness, effectiveness
- Common definition for effectiveness
  - Power: ratio of throughput to mean delay
  - balances throughput, T, and mean delay, E[D]
  - In an M/M/1 queue, $E[D] = 1 / (\mu - \lambda)$ and $T = \lambda / \mu \Rightarrow \text{Power} = \lambda - \lambda^2 / \mu$
  - an optimum load can be determined for Power-curve



Fairness

- Fairness is another important issue
  - no universal (mathematical) definition for fairness
  - depends on how many relevant dependencies are included in the model
- All being equal aspect (in best effort networks)
  - everybody gets equal service
  - all resources available to everybody
  - each is expected to respect others and behave accordingly
  - when a new connection is added, everybody gets a little bit worse service
- Economical aspect (in QoS enabled networks)
  - you should get what you pay for
  - old flows should not experience harm if a new flow is accepted

Outline

- Congestion control and resource allocation
- Queuing disciplines
- TCP congestion control algorithm
- Congestion avoidance at routers and hosts

Queuing Discipline

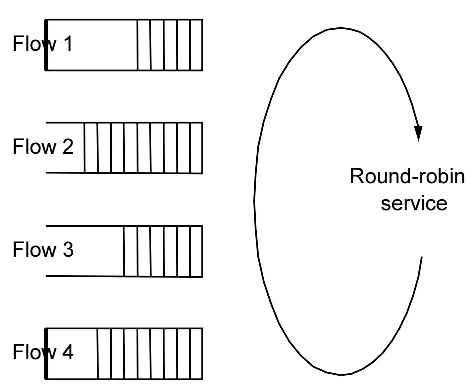
- Choice of queuing discipline affects:
  - allocation of bandwidth (which packets get transmitted) and allocation of buffer space (which packets get discarded)
- Two mechanisms:
  - scheduling (order in which packets are transmitted)
  - drop policy (which packets are dropped)

- First-In-First-Out (FIFO)
  - does not discriminate between traffic sources
  - FIFO with tail-drop ⇒ congestion control and resource allocation pushed out to the edges of the network (current Internet)
  - problems: no protection btw. traffic flows and ill-behaved source can take all capacity
- Priority queuing
  - problem: high priority queue can starve all other queues
  - high priority traffic must be regulated (e.g., by pricing)
  - used to protect most important packets (e.g., routing updates after topology change)

Fair Queuing (FQ) overview

- Problem with FIFO: traffic flows interfere with each other

- FQ: separate queue for each active flow, served in round-robin manner
  - segregates traffic
  - no flow captures more than its fair share of capacity
  - operates together with end-to-end congestion control (i.e., per flow)
  - complication: packets of different length ⇒ need bit-by-bit round-robin
  - work conserving: server never idle as long as there are packets



- FQ extensions
  - FQ for “traffic classes” (Diff Serv)
  - non-equal sharing: weighted fair queuing (WFQ)

FQ algorithm

- Suppose clock ticks each time a bit is transmitted
- Definitions
  - let P(i) denote the length of packet i
  - let S(i) denote the time when start to transmit packet i
  - let F(i) denote the time when finish transmitting packet i

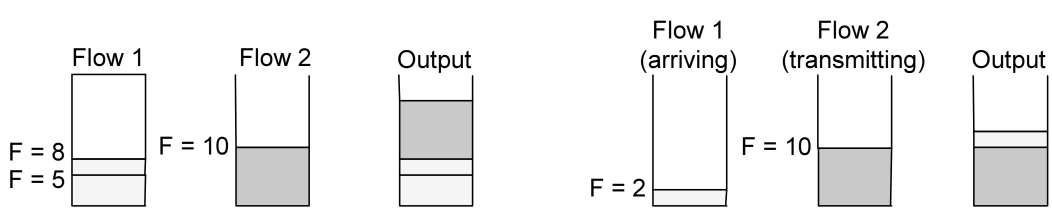
$$\Rightarrow F(i) = S(i) + P(i)$$

- When does router start transmitting packet i?
  - if before router finished packet i - 1 from this flow, then immediately after last bit of i - 1
  - if no current packets for this flow, then start transmitting when arrives (call this A(i))

$$\Rightarrow F(i) = MAX(F(i - 1), A(i)) + P(i)$$

FQ algorithm (cont.)

- For multiple flows
  - calculate F(i) for each packet that arrives on each flow
  - treat all F(i)'s as timestamps
  - next packet to transmit is one with lowest timestamp
- Not perfect: can't preempt current packet
- Example



(a) (b)

Outline

- Congestion control and resource allocation
- Queuing disciplines
- TCP congestion control algorithm
- Congestion avoidance at routers and hosts

TCP Congestion Control

- Introduced in late 1980s after series of congestion collapses:
  - sources sending packets as fast as advertised window allows ⇒ packet drops ⇒ retransmissions ⇒ even worse congestion
  - packets = TCP segments
- Idea
  - assumes best-effort network (FIFO or FQ routers) where each source determines network capacity for itself
  - send packets without reservation and react to observable events
  - uses implicit feedback (oberserves lost packets)
  - **self clocking**
    - TCP does **not** calculate time to send next packet (not rate based)
    - instead, arrival stream of ACKs pace transmission (for each received ACK, new packet can be sent)
- Challenge
  - determining the available capacity in the first place
  - adjusting to changes in the currently available capacity
  - TCP uses only info about packet drops for feedback

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
  - TCP source sending no faster than the slowest component (network or destination host) can tolerate

MaxWin = MIN(CongestionWindow, AdvertisedWindow)
EffWin = MaxWin - (LastByteSent - 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
  - recall how timeout was determined adaptively (measuring RTT)
- AIMD algorithm principle
  - increment CongestionWindow by **one packet per RTT** (*linear increase*)
  - divide CongestionWindow by 2 whenever a timeout occurs (*multiplicative decrease*)
- AIMD properties
  - stability: too large a window much worse than too small
  - for stability, important to approach congestion conservatively and back off aggressively

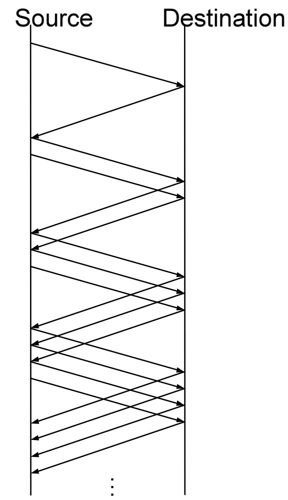
AIMD (cont)

- AIMD in practice:
  - increment window "a little" for each ACK
  - per packet interpretation:
    - w denotes window size in packets
    - increment by 1/w ⇒ increment by 1 for w packets
    - transmitting w packets takes (roughly) one RTT
  - however, TCP counts window in bytes (not packets)

Increment = (MSS * MSS)/CongestionWindow
CongestionWindow += Increment

  - for each loss

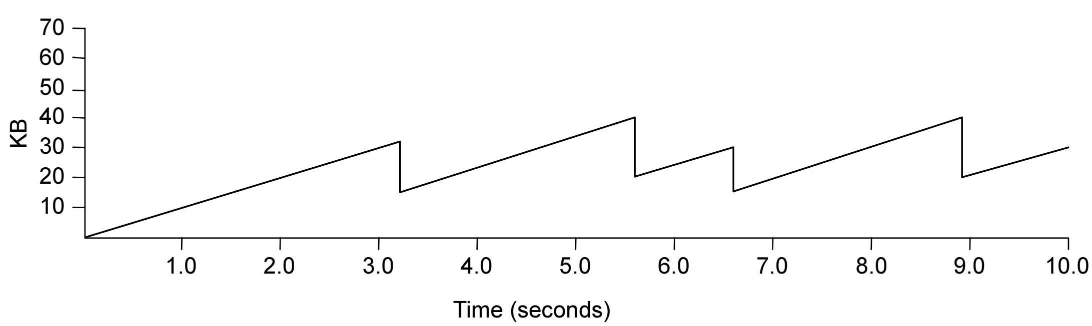
CongestionWindow = CongestionWindow/2



S-38.188 - Computer Networks - Spring 2005

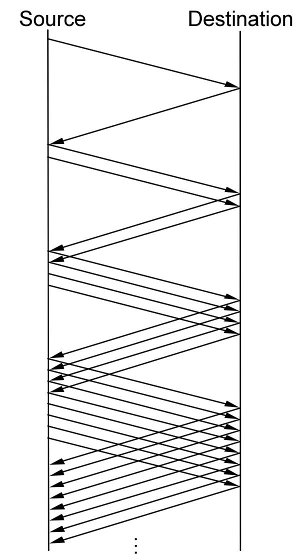
AIMD (cont)

- Trace:
  - window size vs. time
  - sawtooth behavior



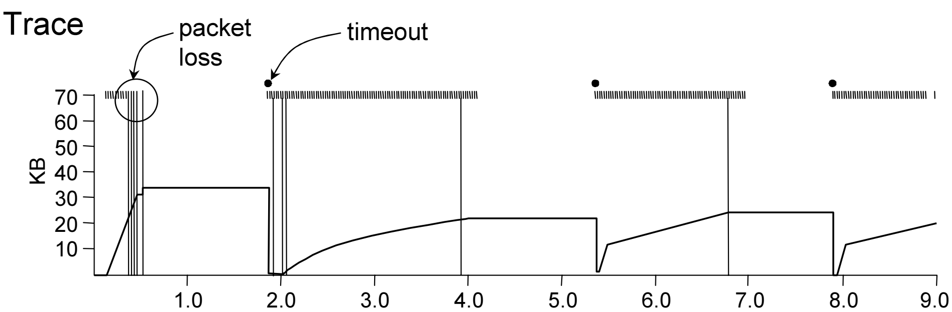
Slow Start

- Objective
  - determine the available capacity at the beginning
- Idea
  - begin with CongestionWindow = 1 packet
  - double CongestionWindow each RTT (increment by 1 packet for each ACK)
  - trying to space packets out to avoid bursts
  - congestion window increases exponentially (still nicer than sending all at once as a burst)
- Used in 2 situations
  - at the beginning of connection
  - when connection goes dead while waiting for a timeout
    - if no packets in transit, no ACKs to “clock” transmission of new packets



Slow Start and AIMD

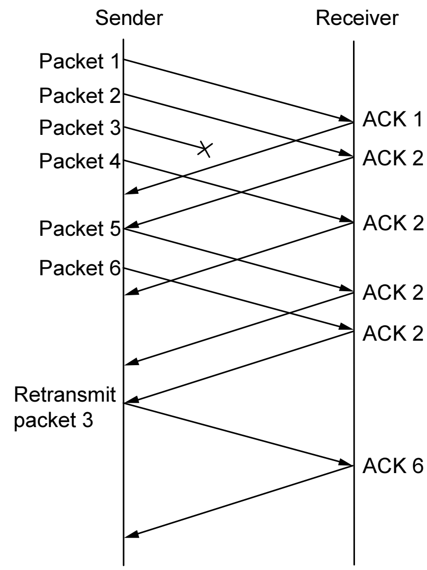
- Switching from slow start to AIMD
  - when transmission goes dead, TCP knows current value of CongestionWindow (= value prior to loss / 2)
  - use that as a “target” window size (= CongestionThreshold)
  - use slow start up to this value, then use additive increase (AIMD)
- Trace



- Problem:
  - during initial slow start may lose up to half a CongestionWindow’s worth of data

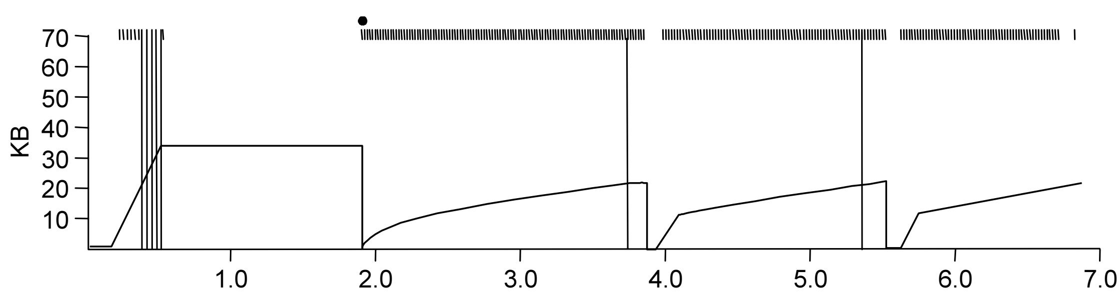
Fast Retransmit and Fast Recovery

- Problem:
  - coarse-grain TCP timeouts lead to idle periods
  - solutions: fast retransmit and fast recovery
- Fast retransmit: use duplicate ACKs to trigger retransmission
  - usually 3 duplicate ACKS
  - about 20% improvement in throughput
- Fast recovery: possible to use ACKs that are still in pipe to clock sending
  - removes some slow start phases
  - halves congestion window and resumes additive increase



Improved TCP behavior and TCP variants

- Trace of TCP with fast retransmit



- TCP with fast recovery
  - under ideal conditions, AIMD type saw tooth without slow starts (except initial slow start)
- TCP variants
  - TCP Tahoe
    - original TCP by Van Jacobson
    - had basic TCP algorithms, AIMD, Slow Start, Fast Retransmit
  - TCP Reno
    - addition of Fast Recovery

About TCP performance

- Window size and sending rate
  - window size = w (in packets, upper bound on number of unacked packets)
  - during one RTT at most w packets can be sent
  - thus, sending rate ~ w/RTT

- TCP throughput influenced by packet loss and RTT, but how?

- Floyd's simple deterministic model
  - window grows linearly from w/2 to w and after reaching w, packet is lost

$$\Rightarrow \quad \frac{w}{2} + (\frac{w}{2} + 1) + \cdots + w \approx \frac{3}{8} w^2 \text{ packets sent / lost packet}$$

$$\Rightarrow \quad p = \frac{8}{3w^2} \quad \Rightarrow \quad rate = \frac{w}{RTT} = \sqrt{\frac{8}{3}} \cdot \frac{1}{RTT \cdot \sqrt{p}}$$

TCP friendly congestion control

- TCP is the most important transport protocol

- TCP friendly: a protocol that behaves like TCP
  - backs off if congestion and uses a fair share of resources
  - protocol that obeys TCP long term throughput relation, T ~ k / (RTT * √p)

- Internet requirement: new transport protocols must be TCP friendly
  - applies also to application layer protocols transmitting over UDP, e.g., real time telephony or streaming applications
  - rate control implemented on top of UDP as part of application

- Non-TCP friendly: a protocol that
  - takes more than its fair share of bandwidth (greedy)
  - may cause fluctuations in network load and result in congestion collapse

- How to protect your protocol against non-TCP friendly greedy protocols?

Outline

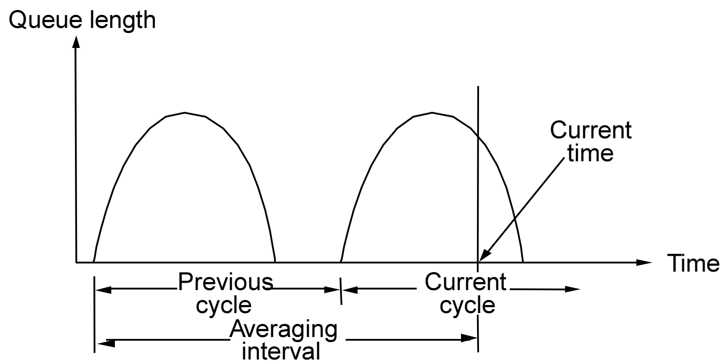
- Congestion control and resource allocation
- Queuing disciplines
- TCP congestion control algorithm
- Congestion avoidance at routers and hosts

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
  - needs to create losses to find out network resources
- 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 + current cycle



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

End host actions

- Operates with TCP sources
- Destination echoes bit back to source
- Source records how many packets resulted in setting the 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

Random Early Detection (RED)

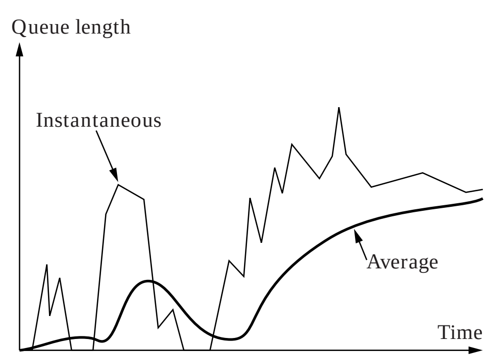
- Notification is implicit
  - just drop the packet (TCP will timeout or see duplicate ACKs)
  - could be made explicit by marking the packet
- Early random drop
  - rather than wait for queue to become full, drop each arriving packet with some drop probability whenever the queue length (load) is “too large”
  - let dropping probability depend on queue length (load)
- Designed to work with TCP sources
  - if congestion detected, drop packets from some (**not all**) TCP sources $\Rightarrow$ some (**not all**) TCPs will back off
  - note: with tail-drop TCP sources can become synchronized easily (all sources increase and decrease windows at the same time)

RED details

- Congestion indicator: averaged queue length
  - low-pass filter, allows transient bursts in the buffer
  - permanent congestion leads to increased averaged queue length
- Computation of average queue length

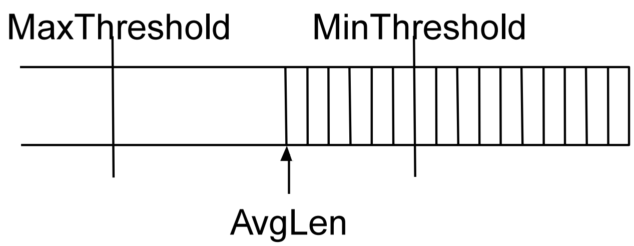
AvgLen = (1 - Weight) * AvgLen + Weight * SampleLen

- 0 < Weight < 1 (usually 0.002)
- SampleLen is queue length each time a packet arrives



RED Details (cont)

- Two queue length thresholds
  - if AvgLen ≤ MinThreshold then enqueue (accept) the packet
  - if MinThreshold < AvgLen < MaxThreshold then
    - calculate probability P
    - drop arriving packet with probability P
  - if AvgLen ≥ MaxThreshold then drop arriving packet



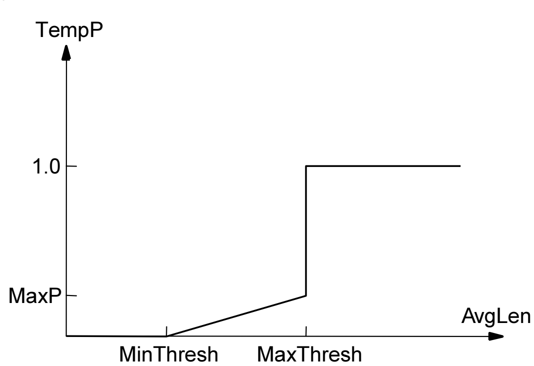
RED Details (cont)

- Computing probability P

TempP = MaxP * (AvgLen - MinThreshold) / (MaxThreshold - MinThreshold)
P = TempP / (1 - count * TempP)

  - count: time in packets since previous drop, used to space drops more evenly

- Drop probability curve:



Tuning RED and problems with RED

- Tuning RED appears to be difficult, topic of current research
  - 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 = 2 x MinThreshold is reasonable for traffic on today's Internet

- Problems with RED
  - tuning is problematic (may even cause oscillations)
  - more importantly, RED does not isolate ill-behaving flows (e.g., UDP flows)
  - has many variants (SRED, RED+, gentle RED, FRED, etc.)

TCP Vegas

- Detecting incipient congestion at end hosts
  - DECbit and RED router based mechanisms
  - could rising congestion be detected at end hosts (at transport layer)?

- Legacy TCP variants
  - TCP Tahoe
  - TCP Reno
  - only react when congestion has already occurred

- TCP Vegas
  - latest TCP variant
  - additional features in congestion control
  - idea: source watches for some sign that router's queue is building up and congestion will soon happen
    - RTT grows
    - sending rate flattens
  - calculates the difference between the expected and the actual sending rates

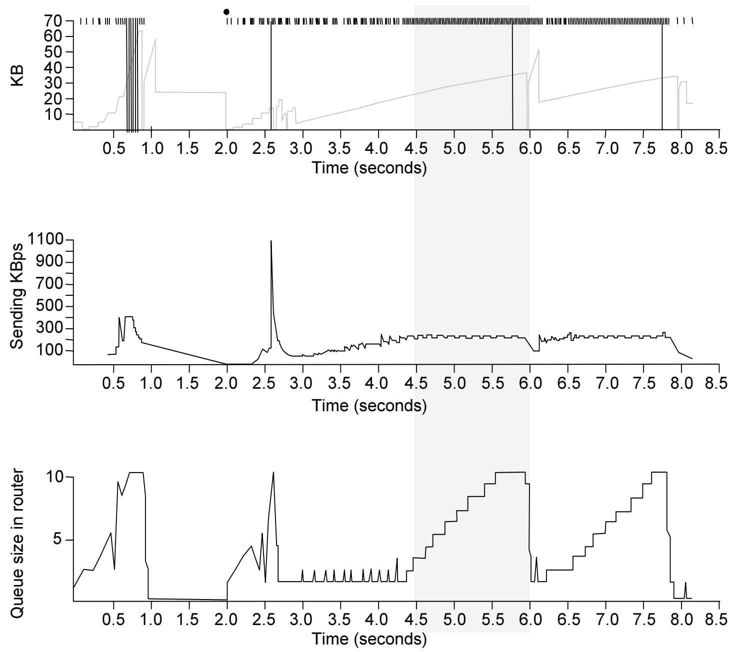
Key observation for TCP Vegas

Observation:

Between 4.5 and 6 s congestion window increases but throughput stays flat

⇒ Throughput can not increase beyond available bandwidth

⇒ Any increase in window size would just increase queues in the bottleneck router



TCP Vegas algorithm

- BaseRTT = minimum of all measured RTTs (usually RTT of first packet)
- If not overflowing the connection, then

ExpectRate = CongestionWindow/BaseRTT

- Source calculates sending rate (ActualRate) once per RTT

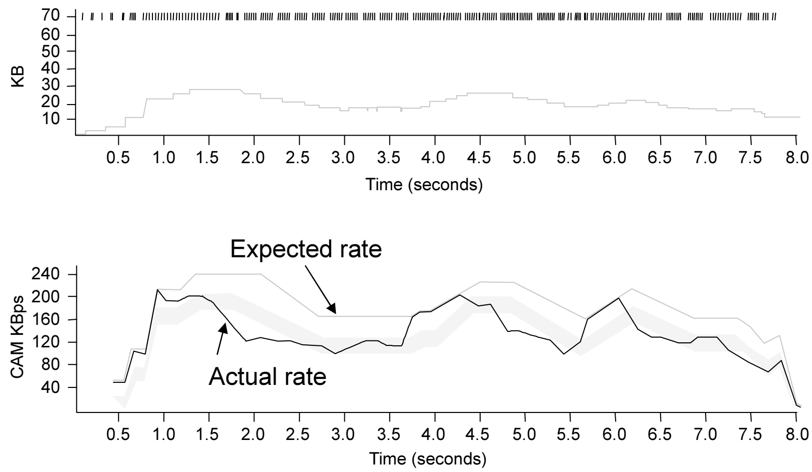
Diff = ExpectedRate – ActualRate

- Source compares ActualRate with ExpectRate

if Diff < a,
  increase CongestionWindow linearly
else if Diff > b,
  decrease CongestionWindow linearly
else,
  leave CongestionWindow unchanged
end

Algorithm (cont)

- Parameters
  – a = 1 packet
  – b = 3 packets



- Even faster retransmit
  – keep fine-grained timestamps for each packet
  – check for timeout on first duplicate ACK
  – multiplicative decrease when timeout occurs, otherwise linear decrease

S-38.188 - Computer Networks - Spring 2005

Evaluating new congestion control mechanisms

- Research has produced a large number of alternative congestion control methods

- Did the algorithm get a great throughput only because it was greedy and all other sources were nice and backed off?
  – What about fairness?
  – Concept of TCP friendliness should help, but still leaves a lot of design freedom…

- How to test the algorithm?
  – Can’t do experiments in the Internet
  – Testing on simulated networks or private testbed networks
  – Challenge: come up with a topology and traffic loads that represent the real Internet
  – What real Internet??? There is no such thing - Internet is changing all the time (keep that in mind when making new algorithms, you need robustness)