COMP 790-088
Networked and Distributed Systems

Congestion Avoidance

Jasleen Kaur

October 14, 2009

Avoiding Congestion
Strategies

- TCP's strategy: congestion control
  - » Control congestion once it occurs
    - ❖ Repeatedly increase load imposed on network
    - ❖ Back-off when congestion occurs
  - » TCP *needs* to create losses to find available bandwidth for transfer
- Alternative strategy: congestion avoidance
  - » Predict when congestion is about to happen
  - » Reduce host sending rate *before* packets start getting dropped
- We'll study three different congestion avoidance strategies:
  - » Router-assisted strategies:
    - ❖ ECN
    - ❖ RED
  - » End-point strategy:
    - ❖ TCP Vegas

ECN
Explicit Congestion Notification

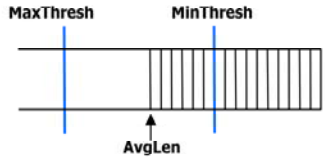
- Basic Approach:
  - » Equally split responsibility of congestion control between routers and hosts
- Router:
  - » Monitors the load it is experiencing
    - ❖ Average queue length, average utilization, etc
  - » Explicitly notifies end hosts when congestion is about to occur
    - ❖ By setting a binary congestion bit in packets that it forwards
    - ❖ Destination hosts echo the bit in ACKs sent to the source
- Source:
  - » Adjusts sending rate on receiving congestion notification

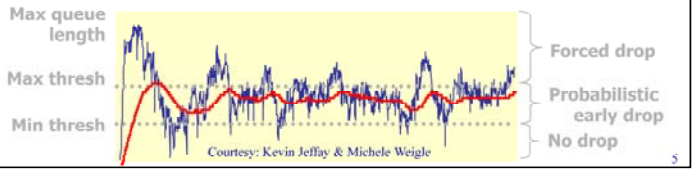
RED
Random Early Detection

- Two main characteristics:
  - » Implicit notification
    - ❖ Just drop packet (end-host detects loss and infers congestion)
  - » Early random drop
    - ❖ Don't wait for queues to be full
    - ❖ Drop packets with some drop probability whenever queue exceeds some drop level
- Is an example of an Active Queue Management (AQM) scheme
  - » Queues are monitored and managed before heavy congestion sets in
  - » Other examples: PI, REM, Blue, …

RED
Queue Monitoring

- Computes an average queue length:
  $AvgLen = (1-a)*AvgLen + a*SampleLen;$ where $a \approx 0.002$
  - » Captures the notion of "long-lived" congestion
    - ❖ Bursty traffic could cause queue to fill up and empty again quickly
- Uses two queue length thresholds:
  if $AvgLen \leq MinThresh$
  enqueue the packet
  else if $AvgLen < MaxThresh$
  drop arriving packet with probability p
  else drop arriving packet

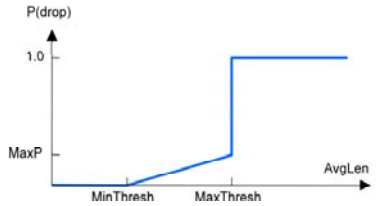




5

RED
Computing the Drop Probability

- How to compute drop probability p, when $AvgLen$ is within the thresholds ?
  - » Depends on $AvgLen$
    - ❖ Higher $AvgLen \Rightarrow$ higher p
  - » Also depends on how long it has been since the last packet was dropped:



tempP = $MaxP*(AvgLen - MinThresh)/(MaxThresh - MinThresh)$
p = tempP / (1 – count*tempP)
- ◆ Where, count is the number of packets received since the last drop
- ❖ Makes it less likely to drop closely spaced packets
  - ◆ So that all drops don't occur in a single transfer
    - – Since packets from same transfer tend to arrive in bursts
  - ◆ Single drop sufficient to cause window reduction
    - – Multiple drops might send transfer back to timeouts and slow start !

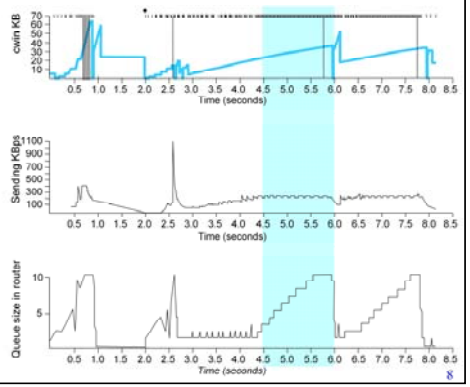
COMP 790-088
6

RED
Discussion

- Fairness of resource allocation:
  - » The probability that RED drops a packet from a given flow is proportional to the flow's current share of bandwidth
    - Flows that are sending more traffic are more likely to be penalized if queues grow
- How to set the *MinThresh* and *MaxThresh*?
  - » If traffic is bursty?
    - *MinThresh* should be set high to allow good link utilization
  - » Given that sources take RTT delay to respond to first indication of congestion?
    - *(MaxThresh–MinThresh)* should be larger than typical increase in AvgLen in RTT
      - *MaxThresh = 2*MinThresh*
    - Value of *a* should help filter out changes in queue length over timescales much smaller than 100 ms

End-point Congestion Avoidance
Congestion Indicators

- How can you detect incipient stages of congestion at end-hosts?
  - » See if there's a measurable increase in RTTs
  - » See if it is correlated with increase in cwin
    if (currWin – oldWin)/(currRTT – oldRTT) > 0, decrease cwin; else increase cwin
  - » Sending rate flattens as network approaches congestion



TCP Vegas
Approach

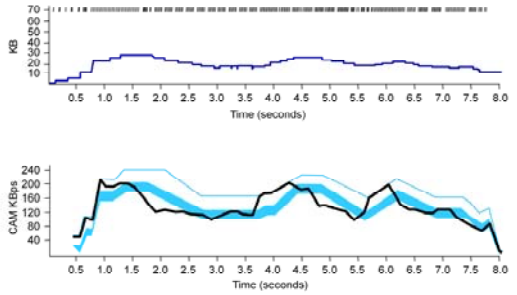
- Vegas uses a mixture of above ideas
  - » Controls the amount of extra data that a transfer has in transit
    - Extra ⇒ data that would not have been transmitted if sending at exactly the network's available bandwidth
  - » Computes and maintains for the transfer:
    - *BaseRTT*: min RTT seen for this transfer
    - *ExpectedRate* = *cwin*/*BaseRTT*
    - *ActualRate*: # of packets sent before an ACK for the first one arrives
  - » Window update (where: *diff* = *ExpectedRate* – *ActualRate*):
    - If *diff* < 0, update *BaseRTT*
    - Else if *diff* < *a*, increase *cwin* linearly
    - Else if *diff* > *b*, decrease *cwin* linearly
    - Else leave *cwin* unchanged
    - If packet loss, decrease *cwin* multiplicatively

Goal: Keep between a*RTT – b*RTT bytes in bottleneck queue

COMP 790-088 9

TCP Vegas
Approach

- Objective: Keep between *a*RTT – b*RTT* bytes in bottleneck queue



COMP 790-088 10

Delay-based Congestion Control

Concerns

- Is it efficient?
  - » Will it react to transient queues (that loss-based TCP will simply let the buffers absorb)?
  - » Can the RTT signal be tainted by OS issues such as interrupt-coalescence, burst-switching, etc?
  - » Will Vegas react to queuing on the reverse path?
- Is it fair?
  - » How will Vegas survive in a TCP-dominated world?
    - ❖ Would it get a fair share of bandwidth against competing TCP transfers?
- How will it survive in wireless environments?
  - » Where several sources of random delays exist
    - ❖ Medium access times
    - ❖ Collision-induced exponential retransmissions
    - ❖ Environment-based rate-adaptation