COMP 790-088
Networked and Distributed Systems

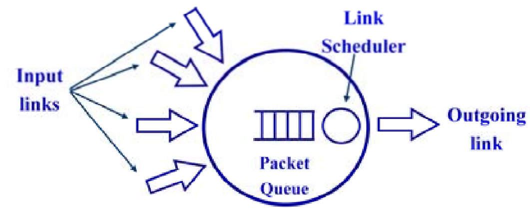
Congestion Control

Jasleen Kaur

October 7, 2009

Network Congestion
Causes

- When and where does congestion occur? (And what *is* congestion?)
  - » When outgoing link capacity is a bottleneck (e.g., access links)
  - » When sum of incoming traffic exceeds outgoing capacity at large timescales
    - ❖ Small timescale bursts are absorbed by queues
- How often does congestion occur in the Internet?
  - » Don't really know (have only anecdotal evidence)
  - » "Congestion collapse" in the 80s led to design of TCP congestion-control



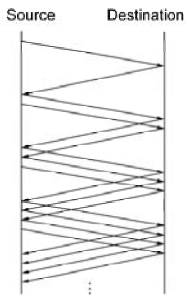
Congestion Control
Conceptual Idea

- Why do we need congestion control?
  - » To enable sharing of common network resource by multiple data sources
- Goal: apply back-pressure to slow down senders if network is congested
  - » Each host determines how much capacity is available in the network
    - This tells it how many packets it can safely have in transit
  - » Once it has these many packet in transit, it uses "self-clocking" to send more
    - The arrival of an ACK is a signal that one of its packets has left the network
    - Hence, it is safe to insert a new packet
  - » If available bandwidth changes, adjust number of packets in transit

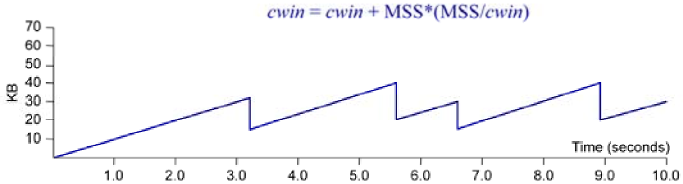
Congestion Window (*cwin*)
New State Variable

- TCP sender maintains a new state variable: Congestion Window
  - » Used by sender to limit how much data it is allowed to have in transit
    - Counterpart to flow control's "*AdvWin*"
  - » Denotes the maximum number of unacknowledged bytes
    - MaxWin = min (*cwin*, *AdvWin*)
    - EffectiveWin = MaxWin – (LBsent – LBacked)
  - » Sender not allowed to send faster than can be accommodated by slowest component (network or destination host)
- Challenge: how to learn the right value for *cwin*?
  - » Unlike destination, network does not explicitly inform sender
- Approach: set *cwin* based on the level of congestion perceived
  - » Decrease *cwin* when congestion increases
  - » Increase *cwin* when congestion decreases

Setting Congestion Window
Saw-tooth Behavior



- How does source detect network congestion?
  - » Retransmission timeouts (which indicate packet losses, likely due to congestion)
- Multiplicative Decrease:
  - » Decrease *cwin* by half every time a timeout occurs
- Additive Increase:
  - » Increment *cwin* by 1 MSS per RTT
  - » In practice, for each ACK:
    cwin = *cwin* + MSS*(MSS/*cwin*)



Additive Increase/Multiplicative Decrease
Rationale

- Why is increase "additive" and decrease "multiplicative"?
  - » Willingness to reduce congestion window greater than willingness to increase it
  - » Necessary condition for stability
  - » Consequences of having too large a window are worse than having too small a window

Slow Start

Hastening Up Initial Bandwidth Discovery

- Two problems with cwin behavior:
  - » Initial additive ramp-up to appropriate *cwin* may take too long
    - ❖ Can we figure out the level of available bandwidth quickly?
  - » After recovery from timeout, dumping cwin/2 may be too aggressive
    - ❖ Such a "burst" of packets may lead to further losses, even if bandwidth is high
- Slow-start mechanism:
  - » Increase exponentially (rather than linearly), when *cwin* is below "*SSThresh*"
    - ❖ Double the number of packets-in-transit every RTT
  - » For every new ACK received:

    If (*cwin* > *SSThresh*)

    increment = increment*increment/*cwin*

    cwin = **min** (*cwin* + increment, *AdvWin*)
- After recovery from timeout:
  - » set: *SSThresh* = *cwin/2*; *cwin* = 1

COMP 790-088 7

Slow Start Behavior

Hastening Up Initial Bandwidth Discovery

- For every new ACK received:

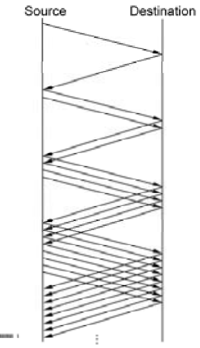
  If (*cwin* > *SSThresh*)

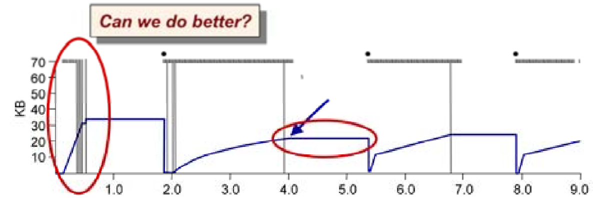
  increment = increment*increment/*cwin*

  cwin = **min** (*cwin* + increment, *AdvWin*)
- After recovery from timeout:

  SSThresh = *cwin/2*

  cwin = 1



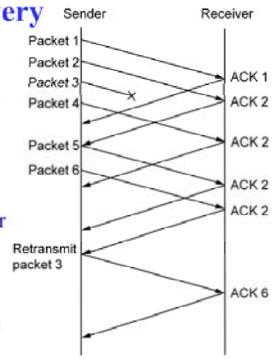


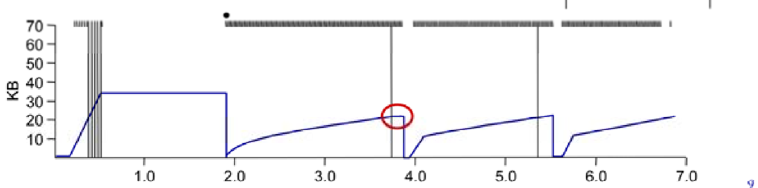
COMP 790-088 8

Fast Retransmit/Fast Recovery (FR/R)

Reducing the Latency For Loss Recovery

- Fast Retransmit:
  - » Heuristic for triggering retransmissions sooner than timeouts
  - » Exploits the fact that receivers send ACKs (in response to data received) even if they are duplicates of earlier ACKs
  - » Use the receipt of 3 duplicate ACKs as indicator that next segment was lost
- Fast Recovery:
  - » Decrease *cwin* to *SSThresh* after fast retransmit





9

RTOs vs. FR/R

Frequency of Occurrence In Practice

- Study of out-of-order segments in 3 million TCP transfers

Trace	All Connections			OOS Connections		
	# Conn	# Bytes	# Packets	% Conn	% Bytes	% Packets
abi	388.9 K	180.1 G	148.5 M	17.60 %	68.11 %	68.85 %
lei	75.4 K	10.5 G	12.6 M	18.82 %	74.88 %	77.24 %
jap	18.5 K	3.3 G	3.1 M	48.65 %	96.08 %	93.44 %
unc	774.8 K	121.3 G	129.6 M	21.82 %	78.45 %	77.83 %
ibi	287.5 K	161.8 G	157.2 M	27.31 %	83.30 %	82.37 %

Fig. 6. Connections That Transmit More Than 10 Segments

Trace	Total OOS	% Network Reorder	% No Inference	Retransmissions						
				# Total	% RTO	% Dupack	% PA	% SACK	% Implicit	% Unexp
abi	409.9 K	18	9.6	296.2 K	32.5 (45)	11.5 (15.9)	2.0 (2.78)	4.5 (6.21)	18.3 (25.3)	3.5 (4.8)
lei	51.1 K	0.47	1.7	49.9 K	46.3 (47.4)	5.9 (6.0)	1.9 (1.9)	4.9 (5.0)	27.7 (28.3)	11.1 (11.4)
jap	51.6 K	2.9	0.4	49.9 K	47.5 (49.1)	11.9 (12.4)	2.6 (2.7)	1.7 (1.8)	31.4 (32.4)	1.6 (1.6)
unc	697.7 K	29	6.69	445.6 K	34.2 (53.5)	6.1 (9.6)	2.8 (4.4)	1.5 (2.3)	13.2 (20.6)	6.0 (9.5)
ibi	504.2K	0.2	0.7	499.3 K	33.1 (33.4)	14.0 (14.0)	4.8 (4.9)	0.0 (0.0)	44.1 (44.5)	3.0 (3.1)

Table 2. Classification of OOS segments (numbers in parenthesis are normalized w.r. to total retransmissions)

10

Unneeded Retransmissions

Premature Timeouts & Reordering-triggered FR/R

- Study of out-of-order segments in 3 million TCP transfers

Trace	Total OOS	% Network Reorder	% No Inference	Retransmissions						
				# Total	% RTO	% Dupack	% PA	% SACK	% Implicit	% Unexp
abi	409.9 K	18	9.6	296.2 K	32.5 (45)	11.5 (15.9)	2.0 (2.78)	4.5 (6.21)	18.3 (25.3)	3.5 (4.8)

Trace	Total # Retran	Our Approach									Allman [6]	
		Needed	Unneeded							No Inference	Needed	Unneeded
			%Total	Implicit	RTO	TDA	PartialAck	Sack	Unexp			
abi	296.2 K	70.0%	20.1%	4.4%	5.8%	1.1%	0.2%	1.0%	7.6%	9.9%	87.4%	12.6%
lei	49.9 K	46.1%	26.8%	7.1%	7.3%	0.7%	0.3%	0.4%	10.9%	27.1%	88.8%	11.2%
jap	49.9 K	70.7%	18.2%	13.3%	2.9%	0.3%	0.2%	0.0%	1.7%	11.1%	84.3%	15.7%
unc	445.6 K	38.5%	38.5%	3.3%	13.3%	2.3%	1.5%	0.1%	17.2%	23.0%	64.3%	35.7%
ibi	499.3 K	67.5%	21.9%	15.3%	2.0%	1.0%	1.2%	0.0%	2.5%	10.6%	77.8%	22.2%

Table 2. Classification of OOS segments (numbers in parenthesis are normalized w.r. to total retransmissions)

Congestion Control in High Speed Networks

The Sluggishness of TCP

- 10 Gbps network with 100 ms round-trip time
  - » Desired *cwin* ≈ 83,000 packets
- Initial bandwidth discovery:
  - » *SSThresh* usually set to no more than 32-64 segments
  - » Would take hours to achieve a sending rate of 10 Gbps
- Bandwidth rediscovery after timeout:
  - » Cwin reset to 1
  - » Additive increase would still take hours to recover 10 Gbps throughput

Congestion Control in High Speed Networks
The Sluggishness of TCP

- Steady-state Congestion Avoidance behavior:
  - » If congestion events occur frequently, average throughput will be less than C
- To achieve 10 Gbps with TCP, only 1 in ($2*10^{10}$) packets should be dropped
  - » This is past the limits of achievable fiber error rates
  - » Packet loss rate of 10^{-7} is reasonable to expect

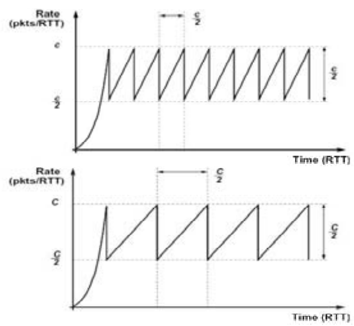


Figure 1: Traditional TCP scaling properties.

Scalable TCP
Basic Idea

- Multiplicative increase:
  - » Increase window more aggressively
    - Standard TCP: cwin = cwin + 1
    - Scalable TCP: cwin = (1+ a)*cwin
- Multiplicative decrease:
  - » Decrease window less aggressively
    - Cwin = b*cwin, where b > 0.5

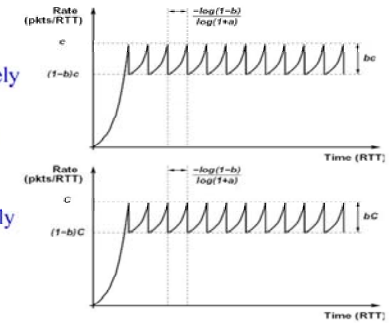


Figure 2: Scalable TCP scaling properties.

Average link utilization achieved is independent of link capacity

Achieving TCP Friendliness

How to Ensure Co-existence with Regular TCP Traffic?

- Define *LowThresh*
  - » Adopt TCP *cwin* behavior below *LowThresh*
  - » Adopt high-speed growth behavior when *cwin* is above *LowThresh*