CS644
Advanced Networks

Lecture 4

Congestion Control

Andreas Terzis

With help from I. Stoica

Problem

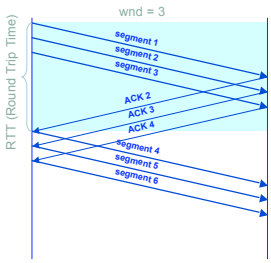
- Where we stopped: Best-effort, datagram based network
- How much traffic do network endpoints send?
- Two components
  - Flow control – make sure that the receiver can receive as fast as you send
  - Congestion control – make sure that the network delivers the packets to the receiver

Why is Congestion Control Important?

- Otherwise you get to congestion collapse
- How might this happen?
  - Assume network is congested (a router drops packets)
  - You learn the receiver didn't get the packet
    - either by ACK, NACK, or Timeout
  - What do you do? retransmit packet
  - Still receiver didn't get the packet
  - Retransmit again
  - …. and so on …
  - And now assume that everyone is doing the same!
- Network will become more and more congested
  - And this with duplicate packets rather than new packets!

Window Size and Throughput

- Sliding-window based flow control:
  - Higher window → higher throughput
    - Throughput = wnd/RTT
- Remember: window size control throughput

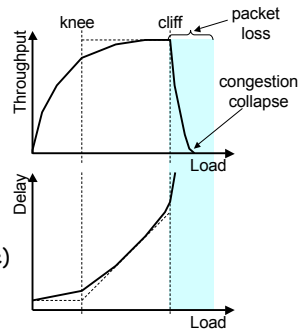


Solutions?

- Increase router buffer size. Why not?
- Slow down
  - If you know that your packets are not delivered because network congestion, slow down
- Questions:
  - How do you detect network congestion?
  - By how much do you slow down?

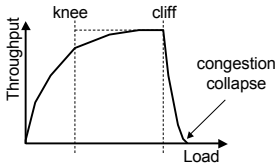
What's Really Happening?

- Knee – point after which
  - Throughput increases very slow
  - Delay increases fast
- Cliff – point after which
  - Throughput starts to decrease very fast to zero (congestion collapse)
  - Delay approaches infinity
- Note (in an M/M/1 queue)
  - Delay = 1/(1 – utilization)



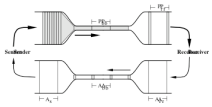
Congestion Control vs. Congestion Avoidance

- Congestion control goal
  - Stay left of cliff
- Congestion avoidance goal
  - Stay left of knee



Congestion Control Goals

- Packet Conservation Principle
  - Don't put a packet into network until another packet leaves. How do you do it?
  - Use ACK: send a new packet only after you receive and ACK. Why?
  - Maintain number of packets in network "constant"



Why is network not in equilibrium?

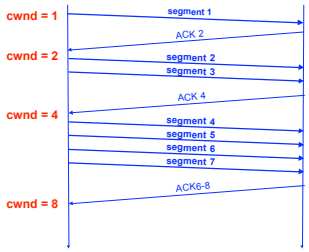
- Connections don't get to steady state
- A sender sends a new packet before an old packet left the network
- Equilibrium value is too high
  - Not enough network resources on the network to reach steady state

TCP: Slow Start

- Goal: discover congestion quickly
- How?
  - Quickly increase *cwnd* until network congested → get a rough estimate of the optimal of *cwnd*
  - Whenever starting traffic on a new connection, or whenever increasing traffic after congestion was experienced:
    - Set *cwnd* =1
    - Each time a segment is acknowledged increment *cwnd* by one (*cwnd*++).
- Slow Start is not actually slow
  - *cwnd* increases exponentially

Slow Start Example

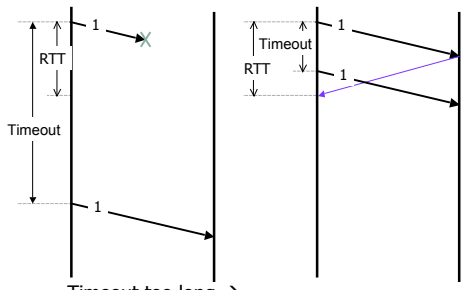
- The congestion window size grows very rapidly
- TCP slows down the increase of cwnd steady state is reached
  - How do know we reached steady state?



Setting Timers

- The sender needs to set retransmission timers in order to know when to retransmit a packet the may have been lost
- How long to set the timer for?
  - Too short: may retransmit before data or ACK has arrived, creating duplicates
  - Too long: if a packet is lost, will take a long time to recover (inefficient)

Timing Illustration



Timeout too long → inefficiency

Timeout too short → duplicate packets

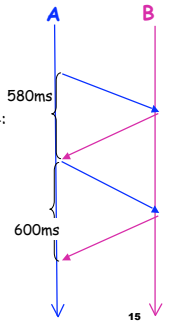
Adaptive Timers

- The amount of time the sender should wait is about the round-trip time (RTT) between the sender and receiver
  - For link-layer networks (LANs), this value is essentially known
  - For multi-hop WANS, rarely known
- Must work in both environments, so protocol should adapt to the path behavior
- Measure successive ack delays T(n)
  Set timeout = average + 4 deviations
- Use exponential back-off to schedule multiple retransmissions

RTT measurement and RTO

SRTT = (1-α) x SRTT + α x SampleRTT
rttvar = rttvar + β x (|diff| - rttvar)
Where diff = SampleRTT - SRTT

Assume SRTT = 500msec, rttvar = 120, α = 1/8, β = 1/4:
diff = SampleRTT - SRTT = 80ms
SRTT = SRTT + α x diff = 510ms
rttvar = rttvar + β (|diff| – rttvar) = 110
RTO = SRTT + 4 x rttvar = 510 + 440 = 950



Adapting to network path changes

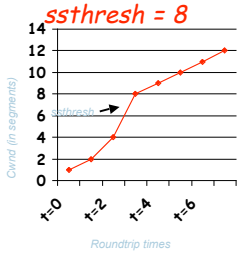
- Equilibrium value changes with time
  - Network path changes
  - Variable number of competing flows
- Congestion Avoidance strategy
  - Network signal
  - Endpoints adapt to congestion signal
- Types of network signals
  - Implicit network signal
    - Loss (e.g. TCP Tahoe, Reno, New Reno, SACK)
    - +relatively robust, -no avoidance
  - Explicit network signal
    - Send packet back to source (e.g. ICMP Source Quench)
    - Set bit in header (e.g ECN)
    - Unless on every router, still need end-to-end signal

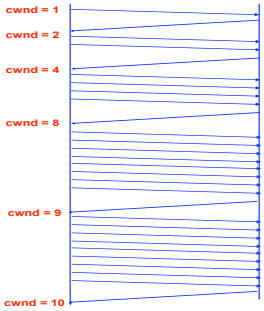
Congestion Avoidance

- Slow down "Slow Start"
- If cwnd > ssthresh then
  each time a segment is acknowledged increment cwnd by 1/cwnd (cwnd += 1/cwnd).
- So cwnd is increased by one only if all segments have been acknowlegded.
- (more about ssthresh latter)

Slow Start/Congestion Avoidance Example

- Assume that *ssthresh = 8*

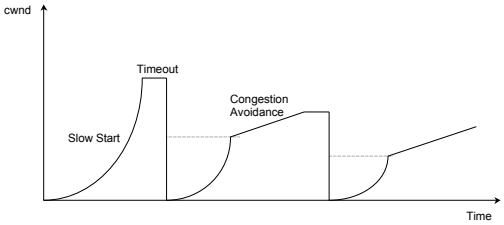




Putting Everything Together: TCP Pseudocode

```
Initially:
    cwnd = 1;
    ssthresh = infinite;
New ack received:
    if (cwnd < ssthresh)
        /* Slow Start*/
        cwnd = cwnd + 1;
    else
        /* Congestion Avoidance */
        cwnd = cwnd + 1/cwnd;
Timeout:
    /* Multiplicative decrease */
    ssthresh = cwnd/2;
    cwnd = 1;
```


The big picture

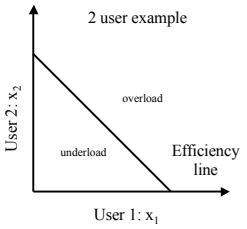


Theoretical foundations

- Why did TCP choose additive increase, multiplicative decrease?
- How can we evaluate different congestion control algorithms?
  - Efficiency
  - Fairness

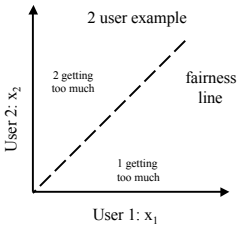
Efficient Allocation

- Too slow
  - Fail to take advantage of available bandwidth → underload
- Too fast
  - Overshoot knee → overload, high delay, loss
- Everyone's doing it
  - May all under/over shoot → large oscillations
- Optimal:
  - $\Sigma x_i = X_{goal}$
- Efficiency = 1 - distance from efficiency line



Fair Allocation

- Maxmin fairness
  - Flows which share the same bottleneck get the same amount of bandwidth
- Assumes no knowledge of priorities
- Fairness = 1 - distance from fairness line



Possible Choices

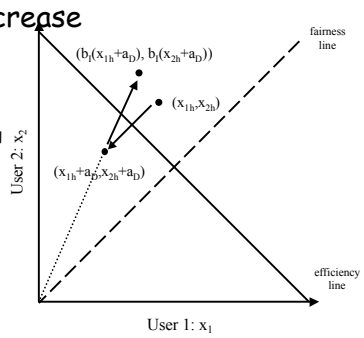
$$x_i(t+1) = \begin{cases} a_I + b_I x_i(t) & increase \\ a_D + b_D x_i(t) & decrease \end{cases}$$

- Multiplicative increase, additive decrease
  - $a_I=0$, $b_I>1$, $a_D<0$, $b_D=1$
- Additive increase, additive decrease
  - $a_I>0$, $b_I=1$, $a_D<0$, $b_D=1$
- Multiplicative increase, multiplicative decrease
  - $a_I=0$, $b_I>1$, $a_D=0$, $0<b_D<1$
- Additive increase, multiplicative decrease
  - $a_I>0$, $b_I=1$, $a_D=0$, $0<b_D<1$
- Which one?

Multiplicative Increase, Additive Decrease

- Does not converge to fairness
  - Not stable at all
- Does not converges to efficiency
  - Stable iff

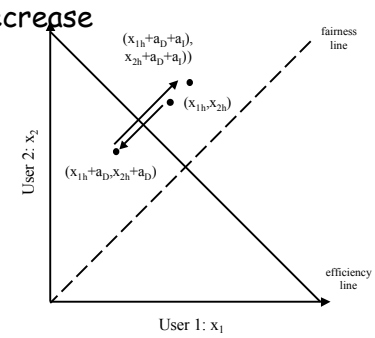
$$x_{1h} = x_{2h} = \frac{b_I a_D}{1 - b_I}$$



Additive Increase, Additive Decrease

- Does not converge to fairness
  - Stable
- Does not converge to efficiency
  - Stable iff

$$a_D = a_I$$

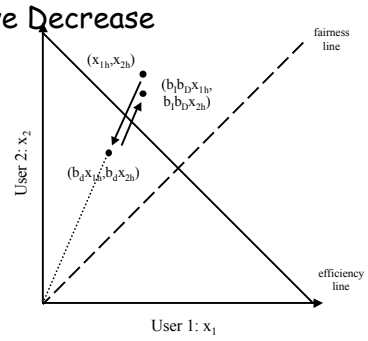


Multiplicative Increase, Multiplicative Decrease

- Does not converge to fairness
  - Stable
- Converges to efficiency iff

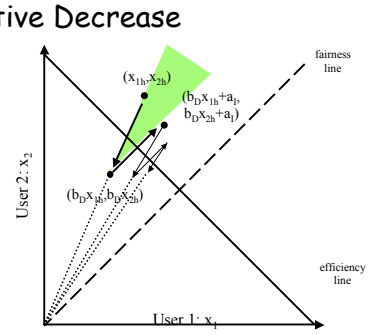
$$b_I \geq 1$$

$$0 \leq b_D < 1$$



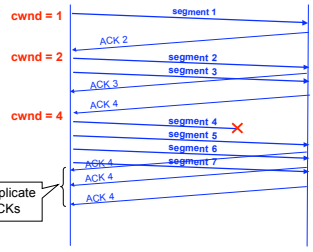
Additive Increase, Multiplicative Decrease

- Converges to fairness
- Converges to efficiency
- Increments smaller as fairness increases
  - effect on metrics?



Fast Retransmit

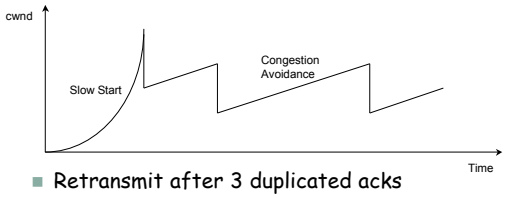
- Don't wait for window to drain
- Resend a segment after 3 duplicate ACKs
  - remember a duplicate ACK means that an out-of sequence segment was received
- Notes:
  - duplicate ACKs due to packet reordering
    - why reordering?
  - window may be too small to get duplicate ACKs



Fast Recovery

- After a fast-retransmit set *cwnd* to *ssthresh/2*
  - i.e., don't reset *cwnd* to 1
- But when RTO expires still do *cwnd* = 1
- Fast Retransmit and Fast Recovery → implemented by TCP Reno; most widely used version of TCP today

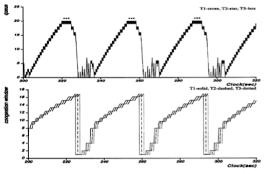
Fast Retransmit and Fast Recovery



- Retransmit after 3 duplicated acks
  - prevent expensive timeouts
- No need to slow start again
- At steady state, *cwnd* oscillates around the optimal window size.

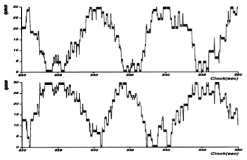
[ZSC91] Aspects of the TCP congestion control algorithm

- One way traffic
  - Synchronization among all TCP flows
    - Through packet loss
  - Leads to reduced link utilization
  - Packets from a single flow are clustered together
    - Due to the way flow opens its window



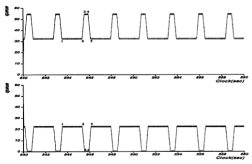
Two way traffic

- Two way traffic very different
  - Rapid queue size fluctuations
  - Reduced utilization
    - Does not improve with increased buffer size
  - Buffers and windows out of phase synchronization



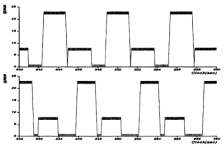
ACK Compression

- TCP relies on ACK clocking to pace packets
  - Assumes that ACKs do not encounter congestion
- Competing flow sends packets in *clusters*
  - ACKs get queued
  - Forward flow responds to cluster of ACKs with cluster of packets



Utilization in two-way traffic case

- In one-way traffic if Sum(W) > 2P link is fully utilized
  - P is the pipe size
- Not true for 2-way traffic
  - ACK queueing increases *effective* pipe size
  - Competing flows means that both links cannot be active at the same time



Congestion Control: Are we there yet?

- TCP shortcomings
  - Assumes that all sources cooperate
  - Assumes that congestion occurs on time scales greater than 1 RTT
  - Only useful for reliable, in order delivery, non-real time applications
  - Vulnerable to non-congestion related loss (e.g. wireless)
  - Can be unfair to long RTT flows