Data Communication Networks

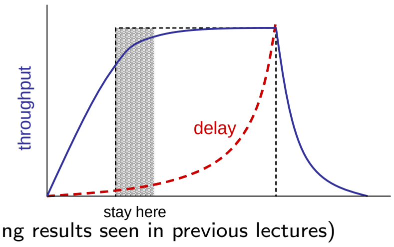
Lecture 8

Saad Mneimneh
Computer Science
Hunter College of CUNY
New York

Congestion control

We would like a congestion control algorithm that is:

- Efficient

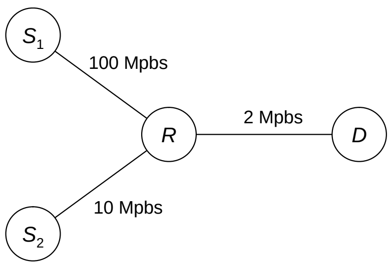


(combining results seen in previous lectures)

  - lost packets $\Rightarrow$ retransmissions
  - long delays $\Rightarrow$ timeouts and retransmissions
  - packets consume resources and then dropped
  - throughput drops tremendously

- Fair
  - make $\frac{(\sum_i r_i)^2}{n \sum_i r_i^2} \approx 1$ for sources sharing a common bottleneck
- Distributed
  - sources are not aware of each other
  - sources are not aware what resrouces (e.g. links) they are using

Example



- S_1 and S_2 are not aware of each other
- S_1 and S_2 don't know that link (R, D) is slow
- Efficiency: $r_1 + r_2 \approx 2$ Mbps
- Fairness: $r_1 = r_1$

- R could actively participate in the congestion control algorithm
  - make S_1 and S_2 aware of each other
  - convey state of link (R, D)
- But we will assume that
  - router is dummy
  - router is FIFO with tail drop strategy (as we have seen before)
  - sources are entirely responsible for congestion control

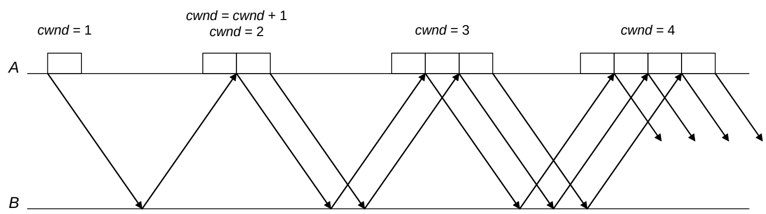
Source based congestion control - Tail drop FIFO router, the TCP way

- TCP controls congestion using the following periodic behavior
  1. create congestion
  2. detect the point at which congestion occurs
  3. back off
- This can be achieved by sending a number of packets, and then
  - if some packets are dropped, decrease the rate
  - if no drops, increase the rate
  - (how does TCP detect a drop?)
- This can be integrated in the sliding window algorithm, i.e. a source can change its sending rate by controlling its window size $cwnd$
  - decrease rate $\Rightarrow$ make $cwnd$ smaller
  - increase rate $\Rightarrow$ make $cwnd$ larger
- Why not control rate directly?
  - already have window algorithm in place
  - if rate increases but window remains small $\Rightarrow$ not much achieved
  - recall, $cwnd = throughput.RTT$

How much increase/decrease?

Additive increase multiplicative decrease AIMD

- Given the window size (every 1 RTT)
  - if no loss (entire window is delivered), $cwnd = cwnd + 1$
  - if there is a loss, $cwnd = cwnd/2$

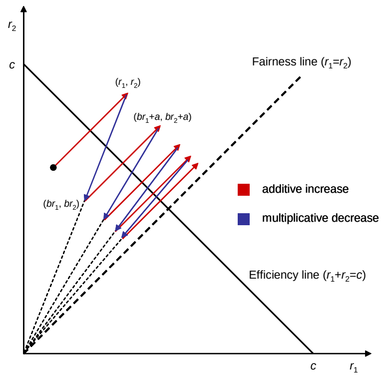


- Actually, TCP does the following:
  - on ACK: $cwnd = cwnd + 1/cwnd$ * (in packets)
  - on timeout: $cwnd = cwnd/2$

(*) if bytes: $cwnd = cwnd + MSS^2/cwnd$

Why AIMD ?

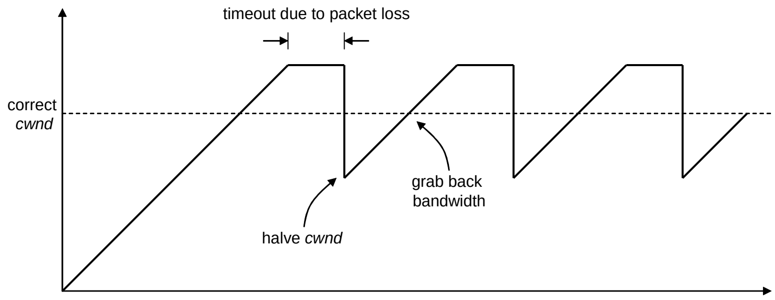
- Theoretically, it can be shown that AIMD converges to a point where the system is both efficient and fair.
- Intuitive illustration



- Both sources share a common bottlenck
  - they are likely to experience similar drops
- Both sources have same RTT
  - $r_1 = r_2 = cwnd/RTT$
- Dashed lines keep $\frac{(\sum_i r_i)^2}{\sum_i r_i^2}$ constant

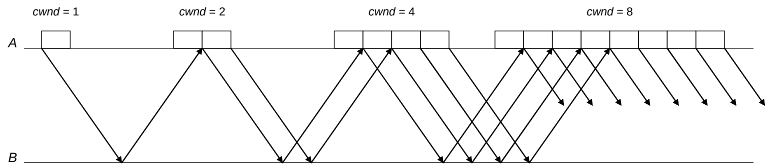
- Bottleneck and RTT assumptions are important

AIMD illustrated



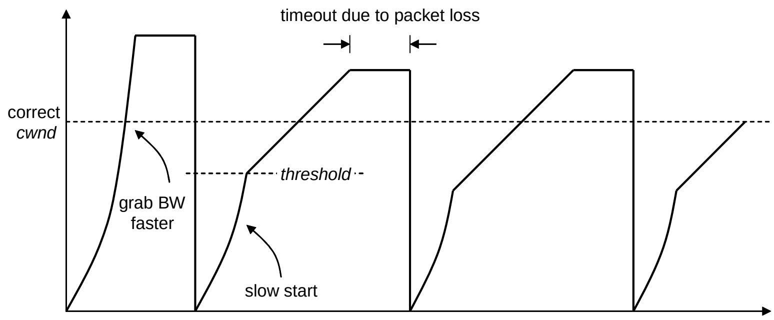
- TCP with AIMD takes a long time to reach the desired bandwidth
- At start up, increase exponentially (multiplicative increase)

Slow start



- on ACK:
  - if $cwnd = threshold$, $cwnd = cwnd + 1/cwnd$ (additive)
  - else $cwnd = \min(cwnd + 1, threshold)$
- on timeout:
  - $threshold = cwnd/2$
  - $cwnd = 0$
- The term slow "slow" start will be put in context shortly

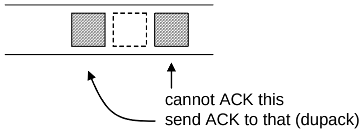
Effect of slow start



- Slow start allows to reach the desired bandwidth faster
- But causes many losses to occur at the beginning
- Why the term slow start ?
  - drops $cwnd$ to zero, requires some time to reach back the desired threshold (instead of immediately starting at the threshold)
  - precaution after timeout (available bandwidth is now possibly less)

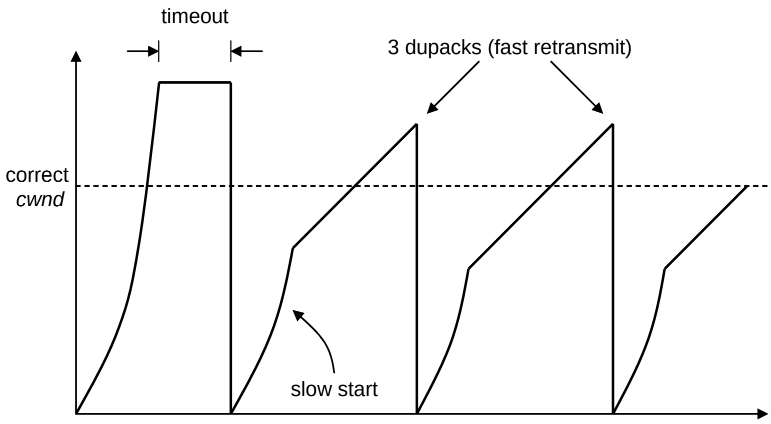
Fast retransmit

- Timeouts are too slow
- Receiver sends ACK upon receipt of each packet (even if out of order)
  - send ACK for the last in-order packet (previously ACKed)
  - sender will obtain a duplicate ACK (dupack)



- Sender interprets a duplicate ACK as a sign of drop, but since packets may be reordered in the network, the sender waits for 3 dupacks
- This technique of waiting for 3 dupacks instead of a timeout is called fast retransmit

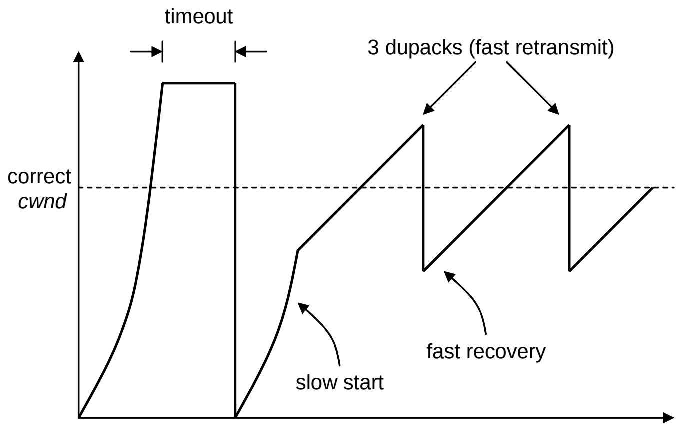
Effect of fast retransmit



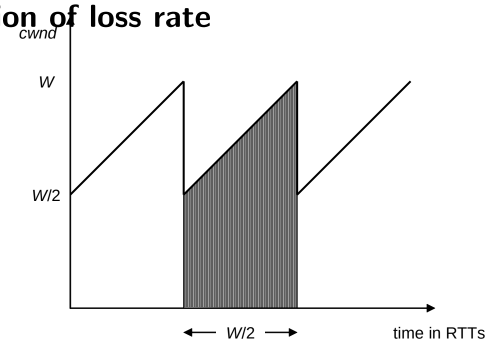
Fast recovery

- If there are still ACKs coming in, then do not slow start
- Let $cwnd = cwnd/2$ after fast retransmit
- On ACK or dupack, $cwnd = cwnd + 1/cwnd$ (packets)
- This technique of not slow starting on dupacks is called fast recovery
- Slow start is used only at the beginning of the connection and after a timeout

Effect of fast retransmit/fast recovery



TCP throughput as function of loss rate



- Every $W/2$ RTTs, we deliver $(W/2)^2 + 1/2(W/2)^2 = 3/8W^2$ packets
- If drop rate is p, then $3/8W^2 \approx 1/p$

$$W = \frac{4}{\sqrt{3p}}$$

- The throughput is

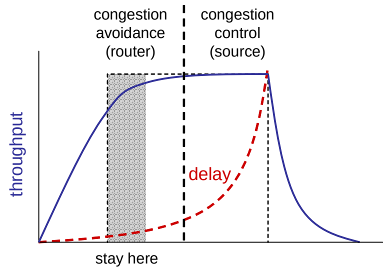
$$\frac{3/8W^2}{RTT.W/2} = \frac{\sqrt{3/2}}{RTT\sqrt{p}} \propto \frac{1}{\sqrt{p}}$$

  packets/sec

What's not so good?

- TCP is based on causing congestion then detecting it
- Assumes flows are long enough for window to stablize
- Assumes all sources cooperate, no pretection against misbehaving sources
- Large average queue size (wait until queue is full to drop)
- Too bursty
  - Window based, packets are dropped in bursts
  - can't use 3 dupacks
  - wait for timeout (less efficient)
- Synchronization and oscillation
  - all losses occur together
  - sources decrease rates together (underutilization)
  - source increase rates together (overflow)
- Vulnerable to non-congestion related drops (e.g. errors)
- Not really fair to flows with large RTT

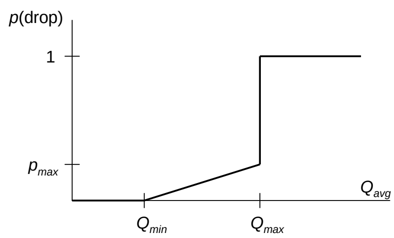
Router participation and congestion avoidance



- Let router participate to avoid congestion before it happens
- Pros
  - a router could better control the resource it owns
  - can do many useful things e.g. fairness (later)
- Cons
  - makes router more complicated (we would like it dummy)
  - deplyment, hard to change existing routers

Random Early Detection (RED)
(drop packets before you really have to)

- RED keeps two thresholds: Q_{min} and Q_{max}
  - try to keep the queue length Q betweeh these two thresholds
- How? When a packet arrives:
  - compute $Q_{avg} = (1-w)Q_{avg} + wQ$, where $0 < w < 1$ (a weight factor)
  - drop the packet with probability $p = f(Q_{avg})$

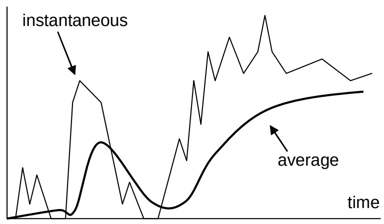


- Originally, $p = \frac{Q_{avg}-Q_{min}}{Q_{max}-Q_{min}} p_{max}$
- Actually, $p_{count} = \frac{p}{1-p\times count}$, where $count$ is the number of packets that have been queued while $Q_{\min} \leq Q_{avg} \leq Q_{max}$ (better avoids clustered drops)

Why average?

Why does RED compute a running average instead of using the instantaneous queue length

- it captures better the notion of congestion
- bursty nature of traffic $\Rightarrow$ queue becomes full quickly and become empty again
- if queue spends most time empty, it is not appropriate to conclude that the network is congested

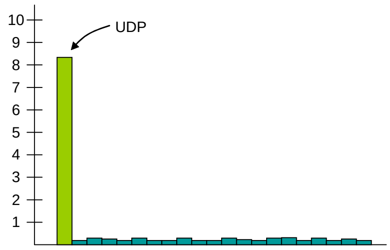


What does RED achieve?

- Less bias against bursty traffic
  - a drop does not necessarily imply a successive drop
- Smaller average queue length
  - starts dropping early when $Q_{min} \leq Q_{avg} \leq Q_{max}$
- Reduces likelihood of bursty drops
  - every packet is dropped independently with a certain probability (original definition of p)
  - closely spaced drops less likely than widely spaced drops (p_{count})
- Reduces likelihood of synchronization
  - drops do not occur simultaneously for all flows (e.g. when queue is full)

Misbehaving flows

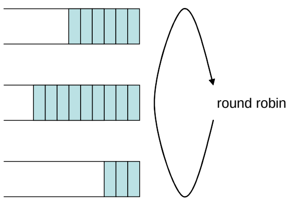
- RED does not provide a protection against misbehaving flows
- Consider a UDP flow sending at 10 Mpbs and sharing a 10 Mbps link with many TCP flows



- UDP does not interprest drops as a sign to decrease its rate ⇒ TCP flows will starve
- Need router intervention to enforce fairness

Fair queueing

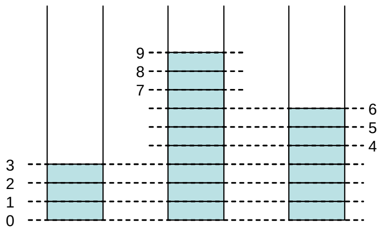
- Router keeps a queue for each flow
- Queues are served in round robin order



- The round robin service at the router does not replace congestion control
  - it does not limit how fast a source is sending packets
  - it is used to enforce fairness
- But is it really fair?
  - if S_1 has 1000 byte packets on average, and S_2 has 500 byte packets on average, then $r_1 = 2/3$ and $r_2 = 1/3$ (unfair)
  - to be really fair, we need a bit-by-bit round robin
  - but we must deal with packets (and packet do not come in one size)

Track the ideal bit-by-bit system

Ideal system



Fair Queueing

- We think of the traffic as being infinitely divisible
- Queues with packets are served simultanesouly
- a_i^k (arrival time): time k^{th} packet of flow i arrives
- S_i^k (start time): time k^{th} packet of flow i starts service
- F_i^k (finish time): time k^{th} packet of flow i finishes service, i.e. departs

- Run the ideal system in the backgroud
- Upon a packet arrival, compute its finish time
- Serve packets in order of their finish times

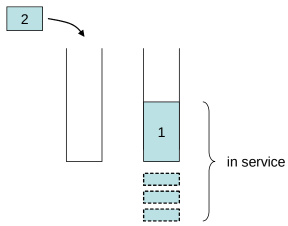
But...

Although the Fair Queueing statement seems innocent, there are few subtleties we need to consider

- Ordering
  - does Fair Queueing mean that packets leave in the same order in both systems?
- Guarantees
  - What does Fair Queueing really achieve?
- Implementation issues
  - how do we run the ideal system in the background

Ordering

- Intuitively, smaller packets are served first
  - smaller finish times
- But longer packets are not pre-empted
  - once a packet start service, it must finish
- In the ideal system, however, a packet can arrive and depart before an existing packet that is being served



Ideal system: packet 2 will depart before packet 1
FQ system: packet 1 will depart before packet 2

- Packets do not leave in the same order in both systems

Guarantees
(we will not show the math)

- Let F_p be the finish time of packet p (in the ideal system)
- Let $\hat{F}_p$ be the time packet p departs the FQ system

$$\hat{F}_p \leq F_p + \frac{L_{max}}{r}$$

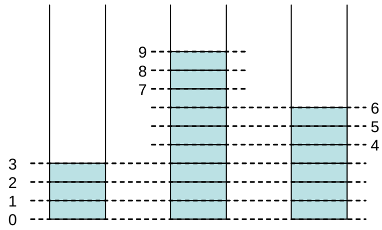
where L_{max} is the length of the largest packet, and r is the service rate

- Let Q_i be the length of the queue for flow i in the ideal system
- Let $\hat{Q}_i$ be the length of the queue in the FQ system

$$\hat{Q}_i \leq Q_i + L_{max}$$

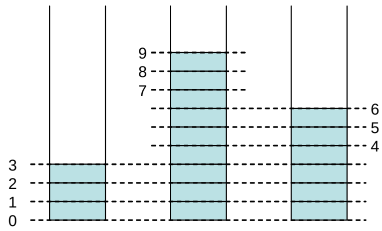
- Therefore, the FQ system tracks the ideal system closely (up to one packet of largest length), provided that we can run the ideal system in the background to compute finish times

Implementation



- Flows are served simultanesouly at equal rate
  - rate of serving a packet varies with number of flows
  - more flows $\Rightarrow$ slower
  - less flows $\Rightarrow$ faster
  - rate not constant during lifetime of a packet
- How to compute finish time of packet?
  - cannot predict future!
  - compute finish times in terms of rounds
  - simulate time in an intelligent way (variable speed clock)

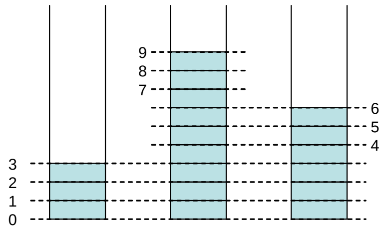
Time in rounds



- p_1 will finish in the 3^{rd} round
- p_3 will finish in the 6^{th} round
- p_2 will finish in the 9^{th} round
- Rounds have different speeds, if server has a rate of c bps then:
  - rounds 1-3 take $3/c$ sec each
  - rounds 4-6 take $2/c$ sec each
  - rounds 7-9 take $1/c$ sec each
- In general, given a Δt and n flows, the number of rounds is

$$\frac{c\Delta t}{n}$$

Computing F_i^k



- The finish time of a packet (in rounds) can be computed as

$$F_i^k = S_i^k + L_i^k$$

  where L_i^k is the length of the packet
- The start time of a packet (in rounds) can be computed as

$$S_i^k = \begin{cases} a_i^k & \text{queue is empty} \\ F_i^{k-1} & \text{otherwise} \end{cases}$$

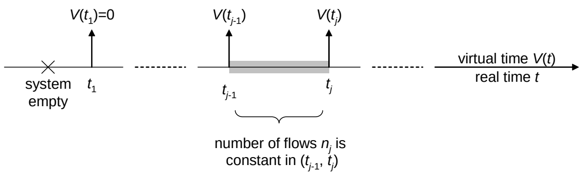
  therefore, $S_i^k = \max(F_i^{k-1}, a_i^k)$
- But a_i^k must be in rounds!

Computing a_i^k

- When a packet arrives at time t, we need to compute a_i^k in terms of rounds
- Let the virtual time $V(t)$ be the time in rounds corresponding to the real time t, i.e. $a_i^k = V(t)$
- if $V(t')$ for some $t' \le t$ is known, and the number of flows n remains the same during (t', t), then

$$V(t) = V(t') + \frac{c\Delta t}{n} = V(t') + \frac{c(t - t')}{n}$$

- Denote by event either an arrival or departure (in the ideal system), then n is constant during an interval of time with no events



$$V(t_j) = V(t_{j-1}) + \frac{c(t_j - t_{j-1})}{n_j}$$

  where n_j is the number of flows in (t_{j-1}, t_j)

Finally...

We only need to compute $V(t)$ for times t where an event occurs

- Arrivals
  - they are the same in both systems
  - if event is arrival at time t, then $a_i^k = V(t)$
- Departures
  - they are not the same in both systems
  - how to detect a departure in the ideal system?
  - the next packet p to leave the ideal system (say at time t_{next}) is the one that has the smallest finish time $F_p > V(t)$ (this does not mean that p is the one to leave next in the FQ system)
  - Let t be the time of the last event, then

$$F_p = V(t) + \frac{c(t_{next} - t)}{n}$$

$$t_{next} = [F_p - V(t)]n + t$$

- Upon arrival at time t, compute $a_i^k = V(t)$ and the next time t_{next} when $V(t_{next})$ must be recomputed

Weighted fair queueing

Each flow i has a weight ϕ_i and flows get service proportional to their weights

$$F_i^k = \max(F_i^{k-1}, a_i^k) + \frac{L_i^k}{\phi_i}$$

$$V(t_j) = V(t_{j-1}) + \frac{c(t_j - t_{j-1})}{\sum_{i \in B_j} \phi_i}$$

where B_j is the set of flows during (t_{j-1}, t_j)