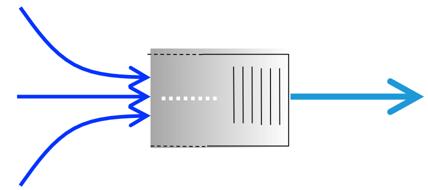
eecs489 COMPUTER NETWORKS

Lectures 31:
TCP Congestion Control

What is Congestion?

What gives rise to congestion?

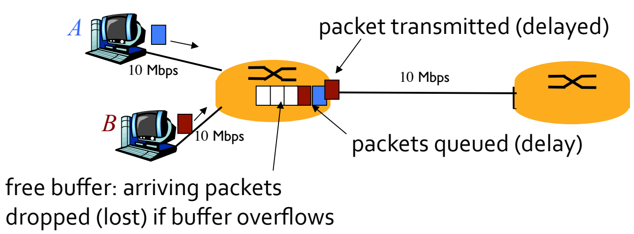
Resource contention: offered load is greater than system capacity
- too much data for the network to handle
- how is it different from flow control?



Why is Congestion Bad?

Causes of congestion:
- packets arrive faster than a router can forward them
- routers queue packets that they cannot serve immediately

Why is congestion bad?
- if queue overflows, packets are dropped
- queued packets experience delay



Consequences of Congestion

If queueing delay > RTO, sender retransmits packets, adding to congestion

Dropped packets also lead to more retransmissions

If unchecked, could result in congestion collapse
- increase in load results in a decrease in useful work done

When a packet is dropped, "upstream" capacity already spent on the packet was wasted

Approaches to Congestion

Free for all
- many dropped (and retransmitted) packets
- can cause congestion collapse
- the long suffering wins

Paid-for service
- pre-arrange bandwidth allocations
- requires negotiation before sending packets
- requires a pricing and payment model
- don't drop packets of the high-bidders
- only those who can pay get good service

Dealing with Congestion

Dynamic adjustment (TCP)
- every sender infers the level of congestion
- each adapts its sending rate "for the greater good"

What is "the greater good" (performance objective)?
- maximizing goodput, even if some users suffer more?
- fairness? (what's fair?)

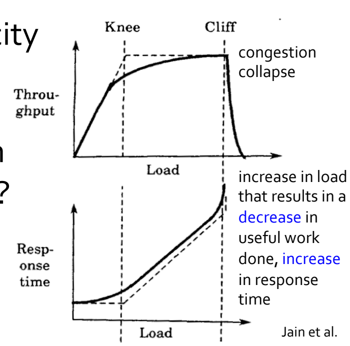
Constraints:
- decentralized control
- unlike routing, no local reaction at routers (beyond buffering and dropping)
- long feedback time
- dynamic network condition: connections come and go

What is the Performance Objective?

System capacity: load vs. throughput:
- congestion avoidance: operate system at "knee" capacity
- congestion control: drive system to near "cliff" capacity

To avoid or prevent congestion, sender must know system capacity and operate below it

How do senders discover system capacity and control congestion?
- detect congestion
- slow down transmission

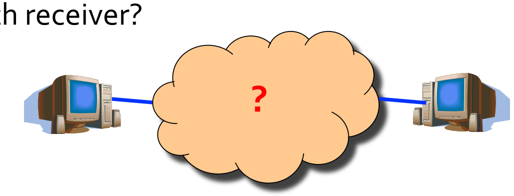


Sender Behavior

How does sender detect congestion?
- explicit feedback from the network?
- implicit feedback: inferred from network performance?

How should the sender adapt?
- explicit sending rate computed by the network?
- sender coordinates with receiver?
- sender reacts locally?

How fast should new TCP senders send?

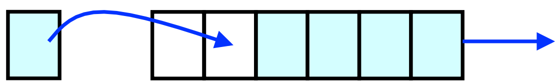


What does the sender see?
What can the sender change?

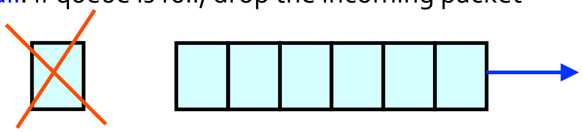
How Routers Handle Packets

Congestion happens at router links
Simple resource scheduling: FIFO queue and drop-tail

Queue scheduling: manages access to bandwidth
- first in first out: packets transmitted in the order they arrive



Drop policy: manages access to buffer space
- drop tail: if queue is full, drop the incoming packet

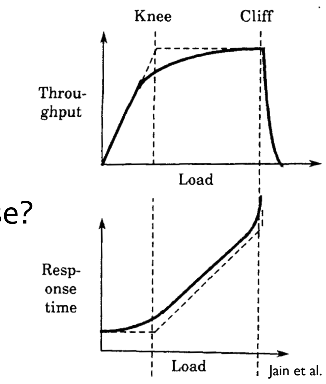


[Rexford]

How it Looks to the Sender

Packet delay
- packet experiences high delay

Packet loss
- packet gets dropped along the way



How does TCP sender learn of these?
- delay:
  - round-trip time estimate (RTT)
- loss
  - retransmission timeout (RTO)
  - duplicate acknowledgments

How do RTT and RTO translate to system capacity?
- how to detect "knee" capacity?
- how to know if system has "gone off the cliff"?

[Rexford]

What can Sender Do?

Upon detecting congestion (packet loss)
- decrease sending rate

But, what if congestion abated?
- suppose some connections ended transmission and
- there is more bandwidth available
- would be a shame to stay at a low sending rate

Upon not detecting congestion
- increase sending rate, a little at a time
- and see if packets are successfully delivered

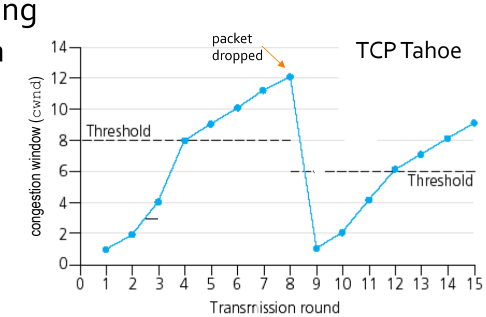
Both good and bad
- pro: obviate the need for explicit feedback from network
- con: under-shooting and over-shooting cliff capacity

[Rexford]

Discovering System Capacity

What TCP sender does:
- probe for point right before cliff ("pipe size")
- slow down transmission on detecting cliff (congestion)
- fast probing initially, up to a threshold ("slow start")
- slower probing after threshold is reached ("linear increase")

Why not start by sending a large amount of data and slow down only upon congestion?

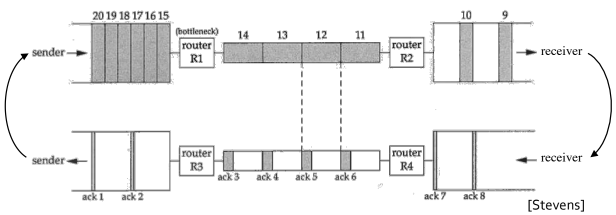


Self-Clocking TCP

TCP uses cumulative ACK for flow control and retransmission and congestion control

TCP follows a so-called "Law of Packet Conservation": Do not inject a new packet into the network until a resident departs (ACK received)

Since packet transmission is timed by receipt of ACK, TCP is said to be self-clocking



[Stevens]

TCP Congestion Control

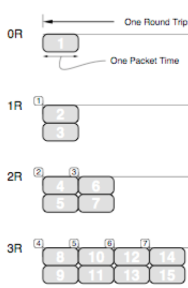
Sender maintains a congestion window (`cwnd`)
- to account for the maximum number of bytes in transit
- i.e., number of bytes still awaiting acknowledgments

Sender's send window (`wnd`) is
`wnd = MIN(rwnd, floor(cwnd))`
- `rwnd`: receiver's advertised window
- initially set `cwnd` to 1 MSS, never drop below 1 MSS
- increase `cwnd` if there's no congestion (by how much?)
  - exponential increase up to `ssthresh` (initially 64 KB)
  - linear increase afterwards
- on congestion, decrease `cwnd` (by how much?)
- always struggling to find the right transmission rate, just to the left of cliff

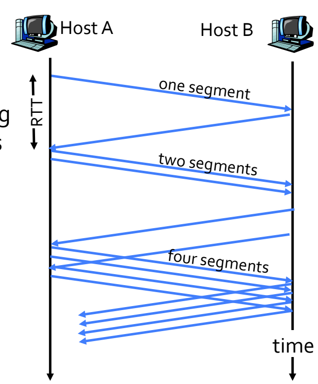
TCP Slow-Start

When connection begins, increase rate exponentially until first loss event:
- double `cwnd` every RTT (or: increased by 1 for every returned ACK)

⇒ really, fast start, but from a low base, vs. starting with a whole receiver window's worth of data as TCP originally did, without congestion control



Jacobson and Karels



Increasing `cwnd`

Probing the "pipe-size" (system capacity) in two phases:

1. slow-start: exponential increase

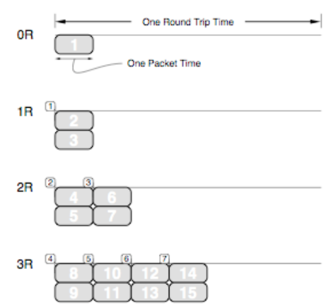
```
while (cwnd <= ssthresh) {
   cwnd += 1
} for every returned ACK
```

OR: `cwnd` *= 2 for every `cwnd`-full of ACKs

2. congestion avoidance: linear increase

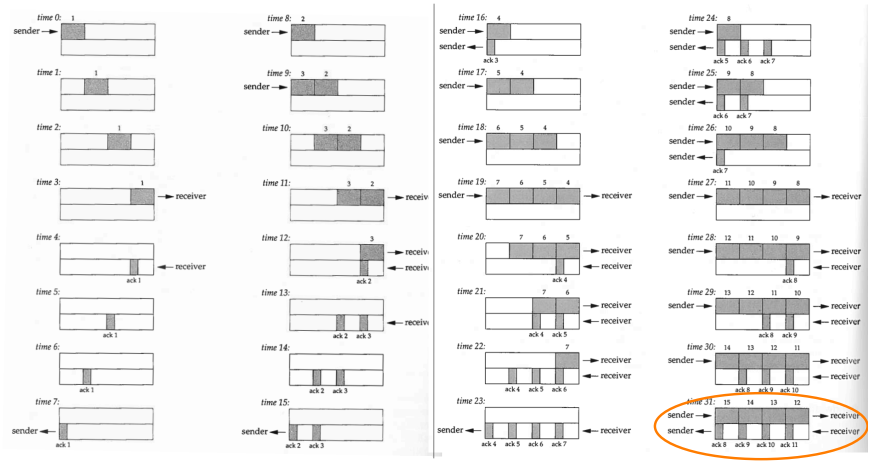
```
while (cwnd > ssthresh) {
   cwnd += 1/floor(cwnd)
} for every returned ACK
```

OR: `cwnd` += 1 for every `cwnd`-full of ACKs



Jacobson & Karels

TCP Slow Start Example



pipe full

Stevens

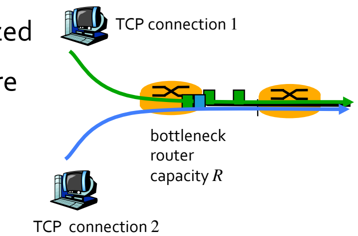
Dealing with Congestion

Once congestion is detected,

- how should the sender reduce its transmission rate?
- how does the sender recover from congestion?

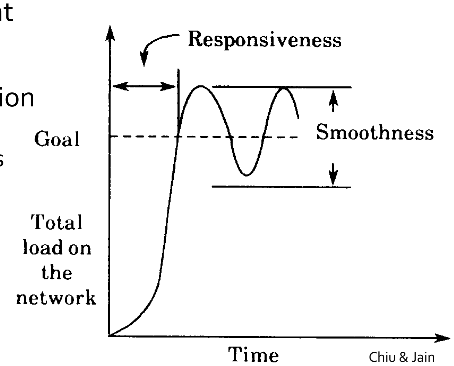
Goals of congestion control:

1. Efficiency: resources are fully utilized
2. Fairness: if k TCP connections share the same bottleneck link of bandwidth R, each connection should get an average rate of R/k



Goals of Congestion Control

3. Responsiveness: fast convergence, quick adaptation to current capacity
4. Smoothness: little oscillation
   - larger change-step increases responsiveness but decreases smoothness
5. Distributed control: no (explicit) coordination between nodes



Chiu & Jain

Guideline for congestion control (as in routing): be skeptical of good news, react fast to bad news

Adapting to Congestion

By how much should `cwnd` (w) be changed?
Limiting ourselves to only linear adjustments:

- increase when there's no congestion: $w' = b_i w + a_i$
- decrease upon congestion: $w' = b_d w + a_d$

Alternatives for the coefficients:

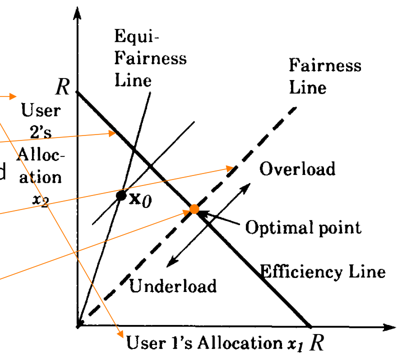
1. Additive increase, additive decrease:
   $a_i > 0,\ a_d < 0,\ b_i = b_d = 1$
2. Additive increase, multiplicative decrease:
   $a_i > 0,\ b_i = 1,\ a_d = 0,\ 0 < b_d < 1$
3. Multiplicative increase, additive decrease:
   $a_i = 0,\ b_i > 1,\ a_d < 0,\ b_d = 1$
4. Multiplicative increase, multiplicative decrease:
   $b_i > 1,\ 0 < b_d < 1,\ a_i = a_d = 0$

Resource Allocation

View resource allocation as a trajectory through an n-dimensional vector space, one dimension per user

A 2-user allocation trajectory:

- x_1, x_2: the two users' allocations
- Efficiency Line: $x_1 + x_2 = \mathbf{x}_i = R$
  - below this line, system is under-loaded
  - above, overloaded
- Fairness Line: $x_1 = x_2$
- Optimal Point: efficient and fair
- Goal of congestion control: to operate at optimal point



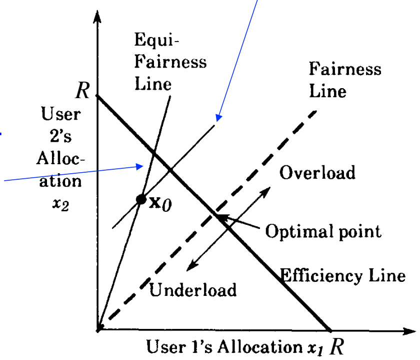
Chiu & Jain

Additive/Multiplicative Factors

Additive factor: adding the same amount to both users' allocation moves an allocation along a 45° line

Multiplicative factor: multiplying both users' allocation by the same factor moves an allocation on a line through the origin (the "equi-fairness," or rather, "equi-unfairness" line)

- the slope of this line, not any position on it, determines fairness

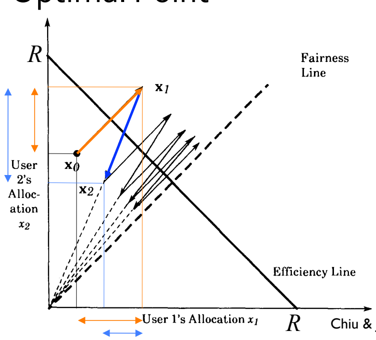


Chiu & Jain

AIMD

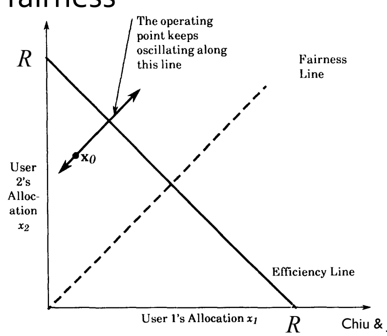
It can be shown that only AIMD takes system near optimal point

Additive Increase, Multiplicative Decrease: system converges to an equilibrium near the Optimal Point



Chiu & Jain

Additive Increase, Additive Decrease: system converges to efficiency, but not to fairness



Chiu & Jain

TCP Congestion Recovery

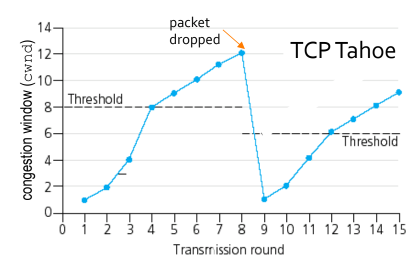
Once congestion is detected,

- by how much should sender decrease `cwnd`?
- how does sender recover from congestion?
  - which packet(s) to retransmit?
  - how to increase `cwnd` again?

First, reduce the exponential increase threshold `ssthresh` = `cwnd/2`

TCP Tahoe:

- retransmit using Go-Back-N
- reset `cwnd`=1
- restart slow-start



Fast Retransmit

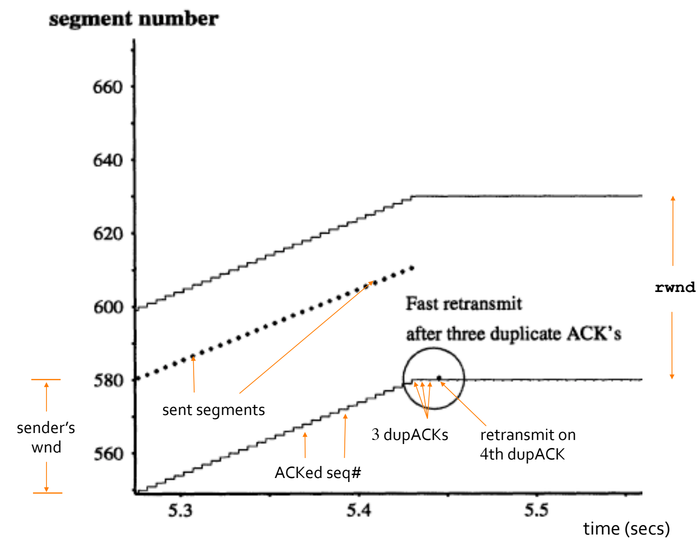
Motivation: waiting for RTO is too slow

TCP Tahoe also does fast retransmit:

- with cumulative ACK, receipt of packets following a lost packet causes duplicate ACKs to be returned
- interpret 3 duplicate ACKs as an implicit NAK
- retransmit upon receiving 3 dupACKs, i.e., on receipt of the 4th ACK with the same seq#, retransmit segment
- why 3 dupACKs? why not 2 or 4?

With fast retransmit, TCP can retransmit after 1 RTT instead of waiting for RTO

Fast Retransmit Example

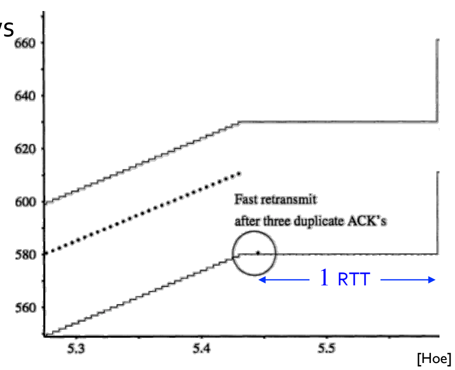


[Hoe]

TCP Tahoe Recovers Slowly

`cwnd` re-opening and retransmission of lost packets regulated by returning ACKs

- duplicate ACK doesn't grow `cwnd`, so TCP Tahoe must wait at least 1 RTT for fast retransmitted packet to cause a non duplicated ACK to be returned
- if RTT is large, Tahoe re-grows `cwnd` very slowly



[Hoe]

TCP Reno and Fast Recovery

TCP Reno does fast recovery:

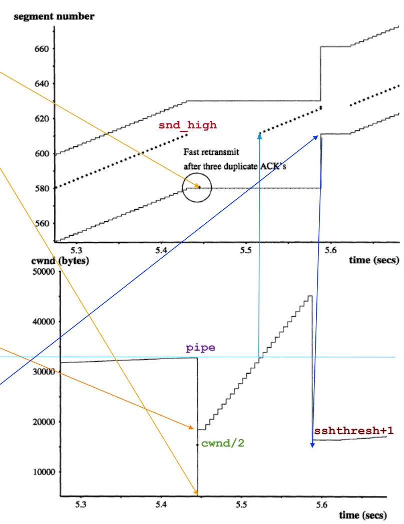
- current value of `cwnd` is the estimated system (`pipe`) capacity
- after congestion is detected, want to continue transmitting at half the estimated capacity

  How?

  - each returning ACK signals that an outstanding packet has left the network
  - don't send any new packet until half of the expected number of ACKs have returned

Fast Recovery

1. on congestion, retransmit lost segment, set `ssthresh = cwnd/2`
2. remember highest seq# sent, `snd_high`; and remember current `cwnd`, let's call it `pipe`
3. decrease `cwnd` by half
4. increment `cwnd` for every returning dupACK, incl. the 3 used for fast retransmit
5. send new packets (above `snd_high`) only when `cwnd > pipe`
6. exit fast-recovery when a non-dup ACK is received
7. set `cwnd = ssthresh + 1` and resume linear increase



`cwnd`: number of bytes unACKed

[Hoe]

Summary: TCP Congestion Control

- When `cwnd` is below `ssthresh`, sender in slow-start phase, window grows exponentially
- When `cwnd` is above `ssthresh`, sender is in congestion-avoidance phase, window grows linearly
- When a 3 dupACKs received, `ssthresh` set to `cwnd/2` and `cwnd` set to new `ssthresh`
- If more dupACKs return, do fast recovery
- Else when RTO occurs, set `ssthresh` to `cwnd/2` and set `cwnd` to 1 MSS

TCP Congestion Control Examples

TCP keeps track of outstanding bytes by two variables:

1. `snd_una`: lowest unACKed seq#,
   i.e., `snd_una` records the seq# associated with the last ACK
2. `snd_next`: seq# to be sent next

Amount of outstanding bytes:

`pipe = snd_next - snd_una`

Scenario:

- 1 byte/pkt
- receiver R takes 1 transmit time to return an ACK
- sender S sends out the next packet immediately upon receiving an ACK
- `rwnd` $= \infty$
- `cwnd` $= 21$, in linear increase mode
- `pipe` $= 21$

Factors in TCP Performance

- RTT estimate
- RTO computation
- sender's sliding window (`wnd`)
- receiver's window (`rwnd`)
- congestion window (`cwnd`)
- slow-start threshold (`ssthresh`)
- fast retransmit
- fast recovery

TCP Variants

Original TCP:

- loss recovery depends on RTO

TCP Tahoe:

- slow-start and linear increase
- interprets 3 dupACKs as loss signal,
  but restart sslow-start after fast retransmit

TCP Reno:

- fast recovery, i.e., consumes half returning dupACKs before transmitting one new packet for each additional returning dupACKs
- on receiving a non-dupACK, resumes linear-increase from half of old `cwnd` value

Summary of TCP Variants

TCP New Reno:

- implements fast retransmit phase whereby a partial ACK, a non-dupACK that is $<$ `snd_high` (seq# sent before detection of loss), doesn't take TCP out of fast recovery, instead retransmits the next lost segment
- only non-dupACK that is $\geq$ `snd_high` takes TCP out of fast recovery: resets `cwnd` to `ssthresh+1` and resumes linear increase