TCP: Congestion Control (part II)

CS 168, Fall 2014

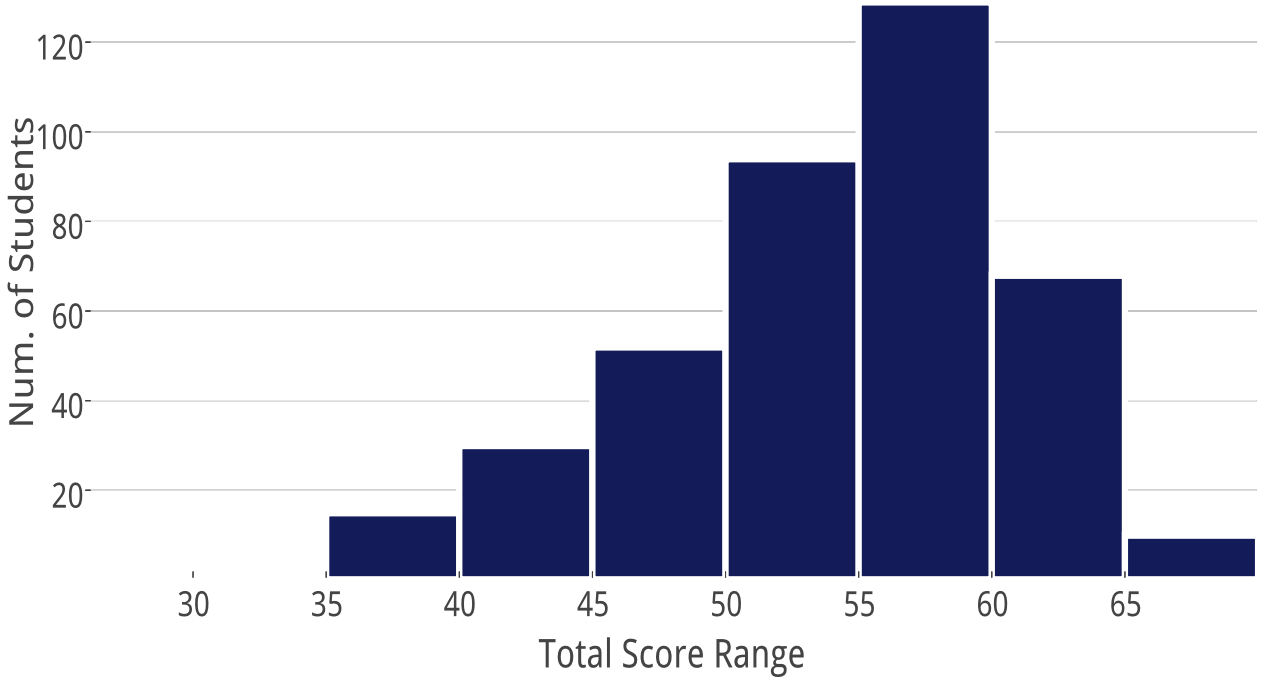
Sylvia Ratnasamy

http://inst.eecs.berkeley.edu/~cs168

Material thanks to Ion Stoica, Scott Shenker, Jennifer Rexford, Nick McKeown, and many other colleagues

Announcements

- Midterm grades and solutions to be released soon after lecture today
- Distribution



Announcements

- We will accept re-grade requests up to noon, Oct 30
- Regrade process if we made a clerical mistake:
  - e.g., you selected answer iii, we graded as ii, *etc*.
  - we'll correct your score immediately
- Regrade process if you disagree with our assessment
  - we will re-grade your entire exam

Last Lecture

- TCP Congestion control: the gory details

Today

- Wrap-up CC details (fast retransmit)
- Critically examining TCP
- Advanced techniques

Recap: TCP congestion control

- Congestion Window: CWND
  - How many bytes can be sent without overflowing routers
  - Computed by the sender using congestion control algorithm
- Flow control window: AdvertisedWindow (RWND)
  - How many bytes can be sent without overflowing receiver's buffers
  - Determined by the receiver and reported to the sender
- Sender-side window = minimum{CWND, RWND}

Recall: Three Issues

- Discovering the available (bottleneck) bandwidth
  - Slow Start
- Adjusting to variations in bandwidth
  - AIMD (Additive Increase Multiplicative Decrease)
- Sharing bandwidth between flows
  - AIMD *vs.* MIMD/AIAD…

Recap: Implementation

- State at sender
  - CWND (initialized to a small constant)
  - ssthresh (initialized to a large constant)

- Events
  - ACK (new data)
  - dupACK (duplicate ACK for old data)
  - Timeout

Event: ACK (new data)

- If CWND < ssthresh
  - CWND += 1

- *CWND packets per RTT*
- *Hence after one RTT with no drops:*
  CWND = 2xCWND

Event: ACK (new data)

- If CWND < ssthresh
  - CWND += 1

Slow start phase

- Else
  - CWND = CWND + 1/CWND

"Congestion Avoidance" phase (additive increase)

- *CWND packets per RTT*
- *Hence after one RTT with no drops:*

  CWND = CWND + 1

Event: TimeOut

- On Timeout
  - ssthresh ← CWND/2
  - CWND ← 1

Event: dupACK

- dupACKcount ++
- If dupACKcount = 3 /* fast retransmit */
  - ssthresh = CWND/2
  - CWND = CWND/2

One Final Phase: Fast Recovery

- The problem: congestion avoidance too slow in recovering from an isolated loss

Example

- Consider a TCP connection with:
  - CWND=10 packets
  - Last ACK had seq# 101
    - i.e., receiver expecting next packet to have seq# 101
- 10 packets [101, 102, 103,…, 110] are in flight
  - Packet 101 is dropped

Timeline (at sender)

In flight: 101, 102, 103, 104, 105, 106, 107, 108, 109, 110 101

- ACK 101 (due to 102) cwnd=10 dupACK#1 (no xmit)
- ACK 101 (due to 103) cwnd=10 dupACK#2 (no xmit)
- ACK 101 (due to 104) cwnd=10 dupACK#3 (no xmit)
- RETRANSMIT 101 ssthresh=5 cwnd= 5
- ACK 101 (due to 105) cwnd=5 + 1/5 (no xmit)
- ACK 101 (due to 106) cwnd=5 + 2/5 (no xmit)
- ACK 101 (due to 107) cwnd=5 + 3/5 (no xmit)
- ACK 101 (due to 108) cwnd=5 + 4/5 (no xmit)
- ACK 101 (due to 109) cwnd=5 + 5/5 (no xmit)
- ACK 101 (due to 110) cwnd=6 + 1/5 (no xmit)
- ACK 111 (due to 101) ← only now can we transmit new packets
- Plus no packets in flight so ACK "clocking" (to increase CWND) stalls for another RTT

Solution: Fast Recovery

Idea: Grant the sender temporary “credit” for each dupACK so as to keep packets in flight

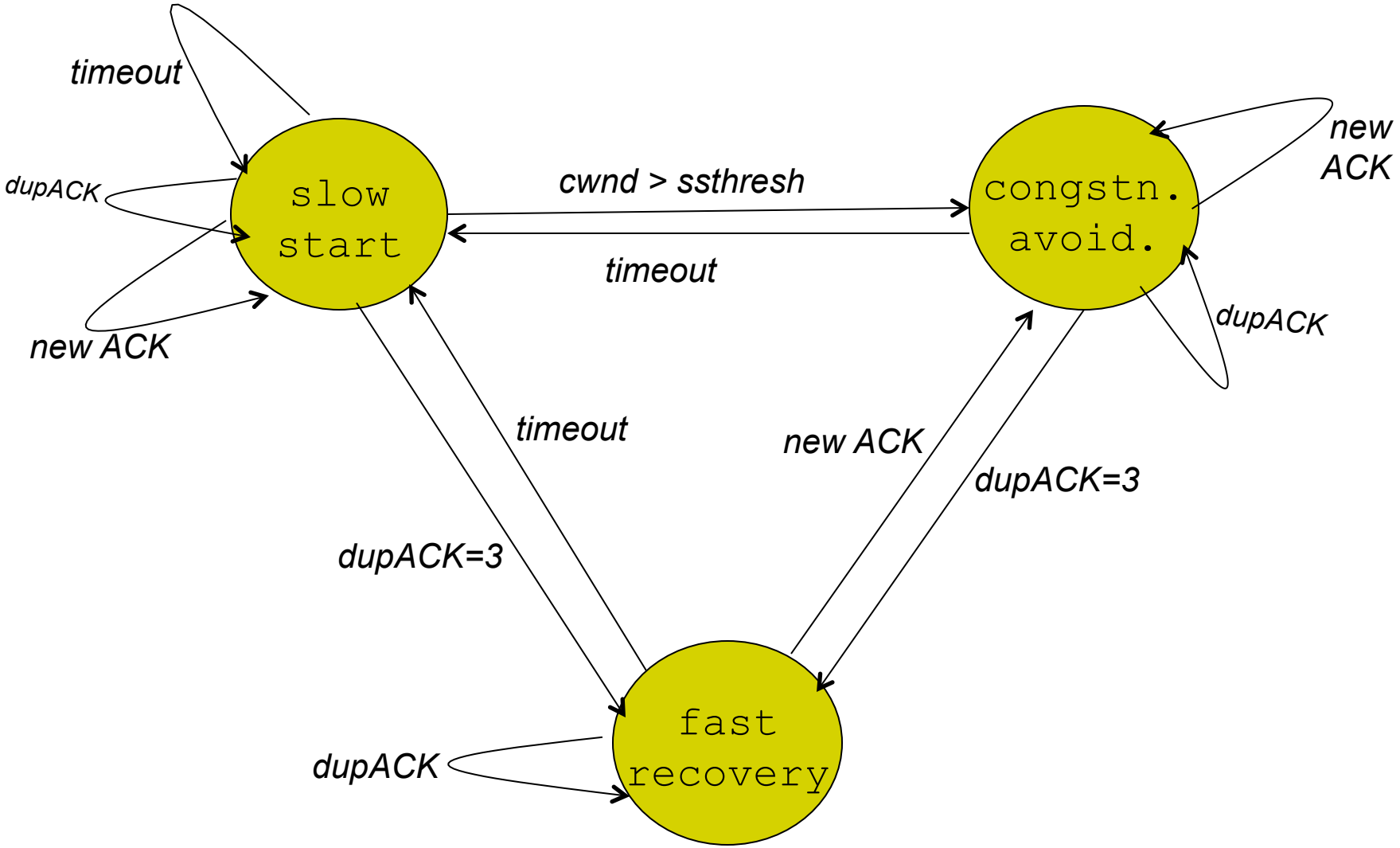
- If dupACKcount = 3
  - ssthresh = cwnd/2
  - cwnd = ssthresh + 3
- While in fast recovery
  - cwnd = cwnd + 1 for each additional duplicate ACK
- Exit fast recovery after receiving new ACK
  - set cwnd = ssthresh

Timeline (at sender)

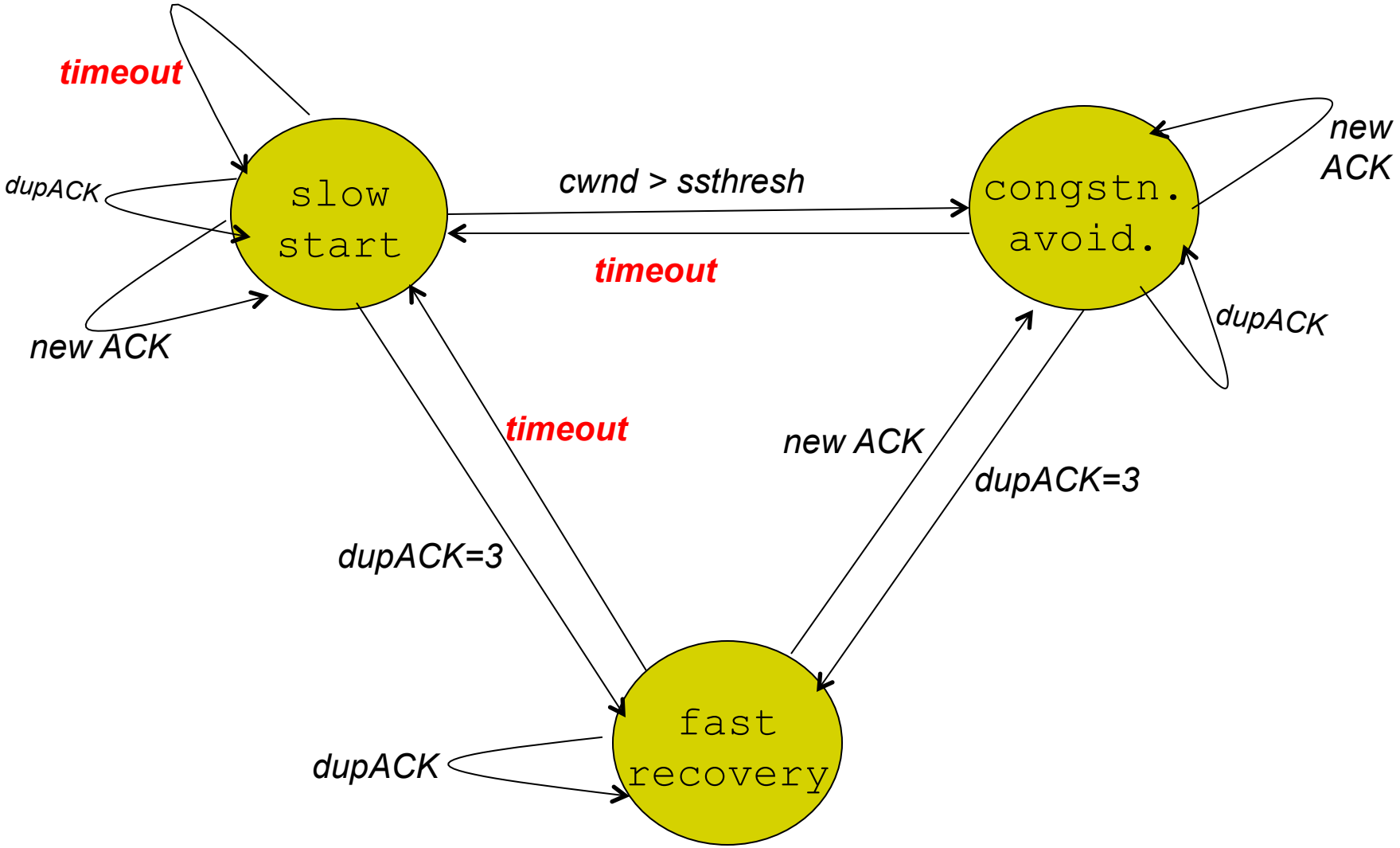
In flight: 101, 102, 103, 104, 105, 106, 107, 108, 109, 110

- ACK 101 (due to 102) cwnd=10 dup#1
- ACK 101 (due to 103) cwnd=10 dup#2
- ACK 101 (due to 104) cwnd=10 dup#3
- REXMIT 101 ssthresh=5 cwnd= **8 (5+3)**
- ACK 101 (due to 105) **cwnd= 9** (no xmit)
- ACK 101 (due to 106) cwnd=10 (no xmit)
- ACK 101 (due to 107) cwnd=11 (**xmit 111**)
- ACK 101 (due to 108) cwnd=12 (**xmit 112**)
- ACK 101 (due to 109) cwnd=13 (**xmit 113**)
- ACK 101 (due to 110) cwnd=14 (**xmit 114**)
- ACK 111 (due to 101) cwnd = 5 (xmit 115) ← exiting fast recovery
- Packets 111-114 already in flight
- ACK 112 (due to 111) cwnd = 5 + 1/5 ← back in congestion avoidance

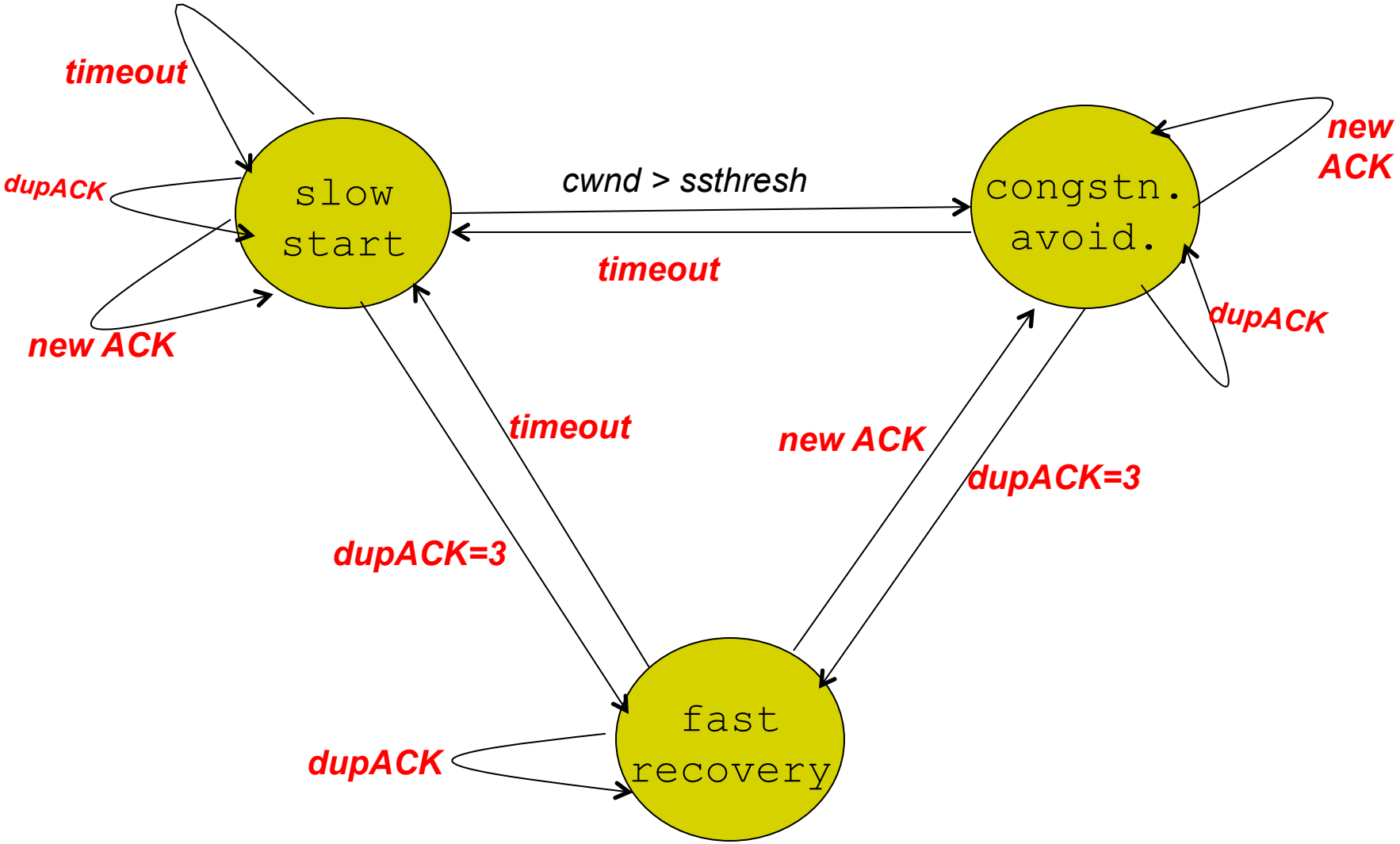
TCP State Machine



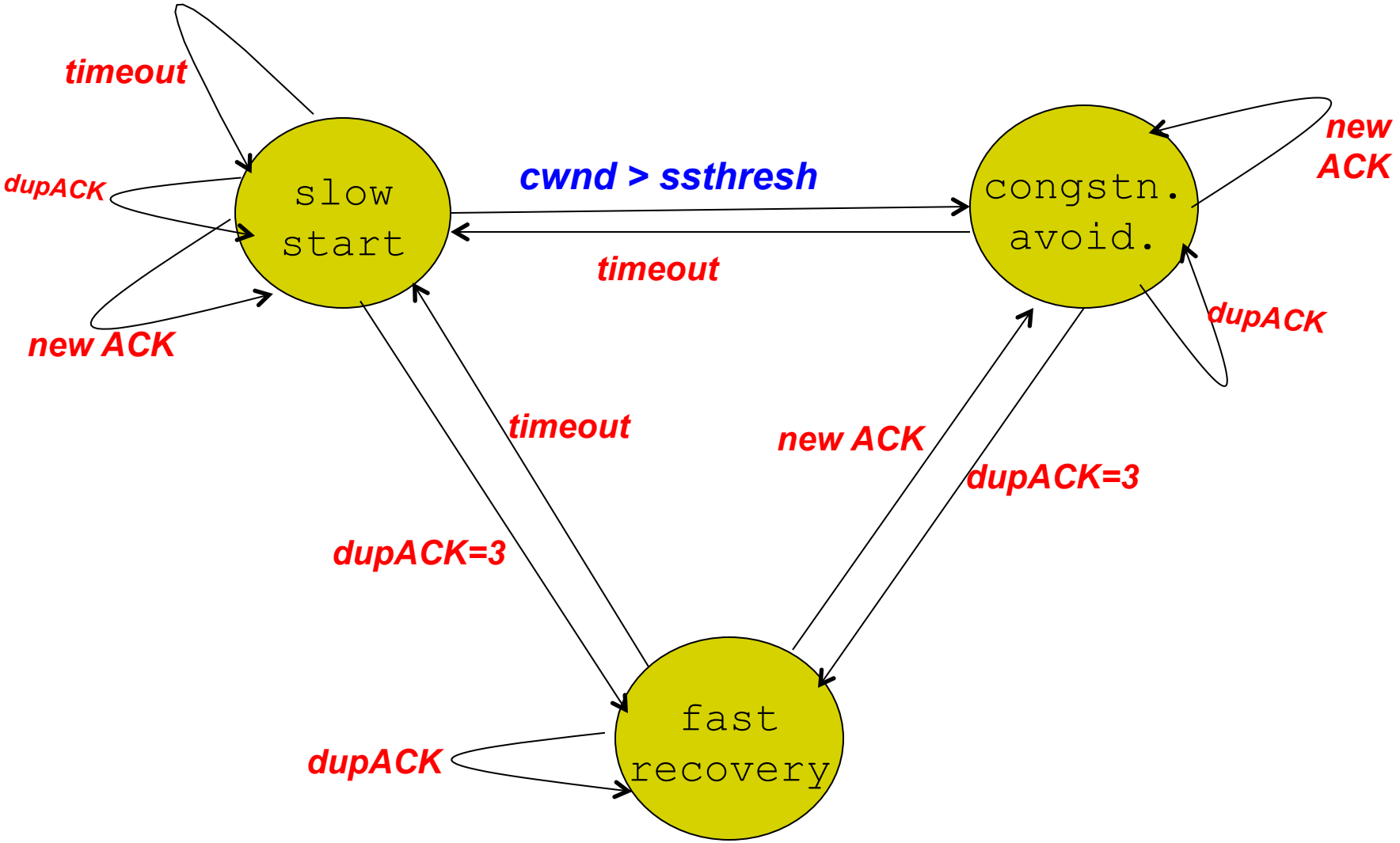
TCP State Machine



TCP State Machine



TCP State Machine



TCP Flavors

- TCP-Tahoe
  - CWND =1 on triple dupACK
- TCP-Reno
  - CWND =1 on timeout
  - CWND = CWND/2 on triple dupack
- TCP-newReno
  - TCP-Reno + improved fast recovery
- TCP-SACK
  - incorporates selective acknowledgements

Our default assumption

Interoperability

- How can all these algorithms coexist? Don’t we need a single, uniform standard?

- What happens if I’m using Reno and you are using Tahoe, and we try to communicate?

Last Lecture

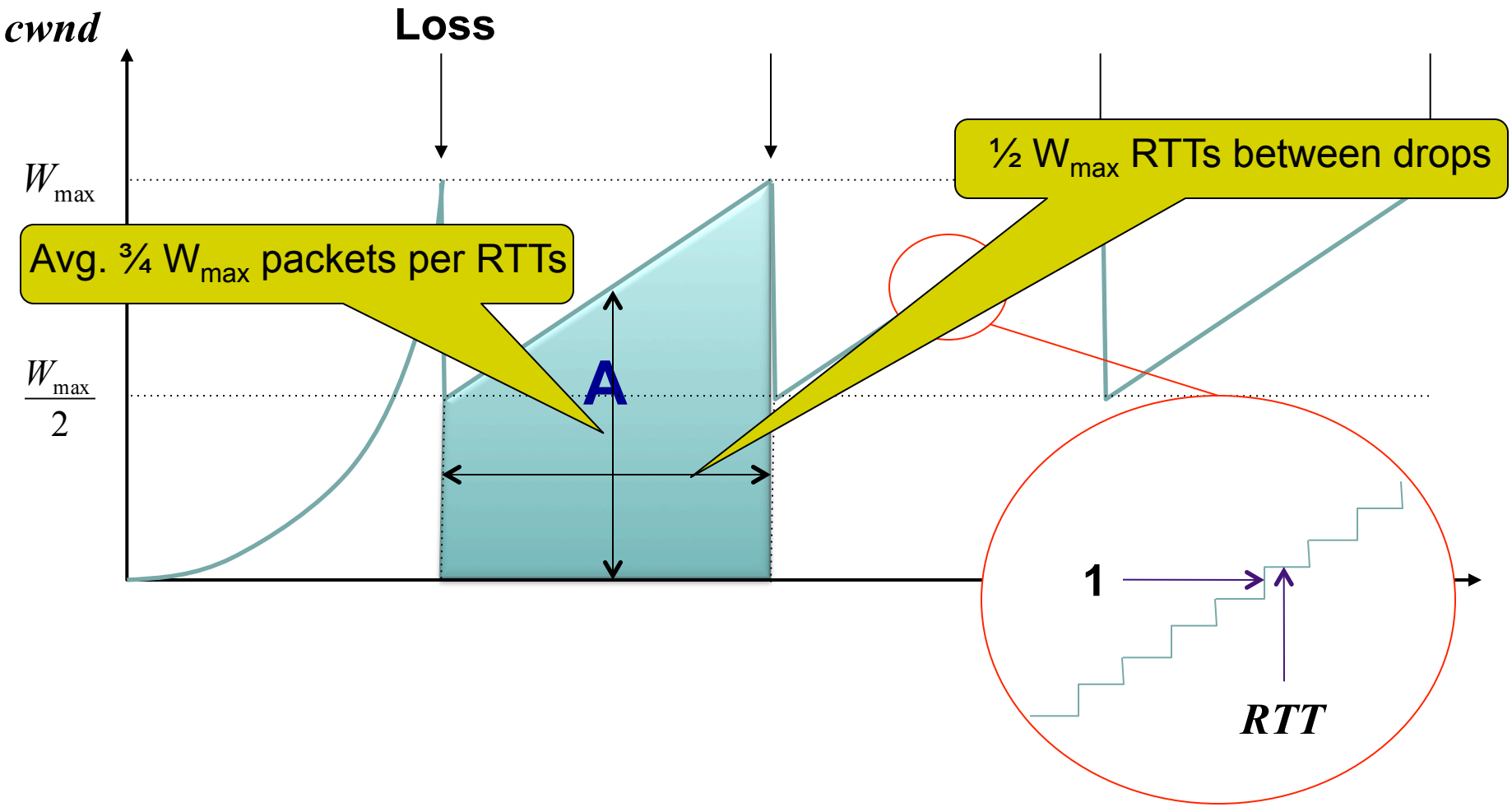
- TCP Congestion control: the gory details

Today

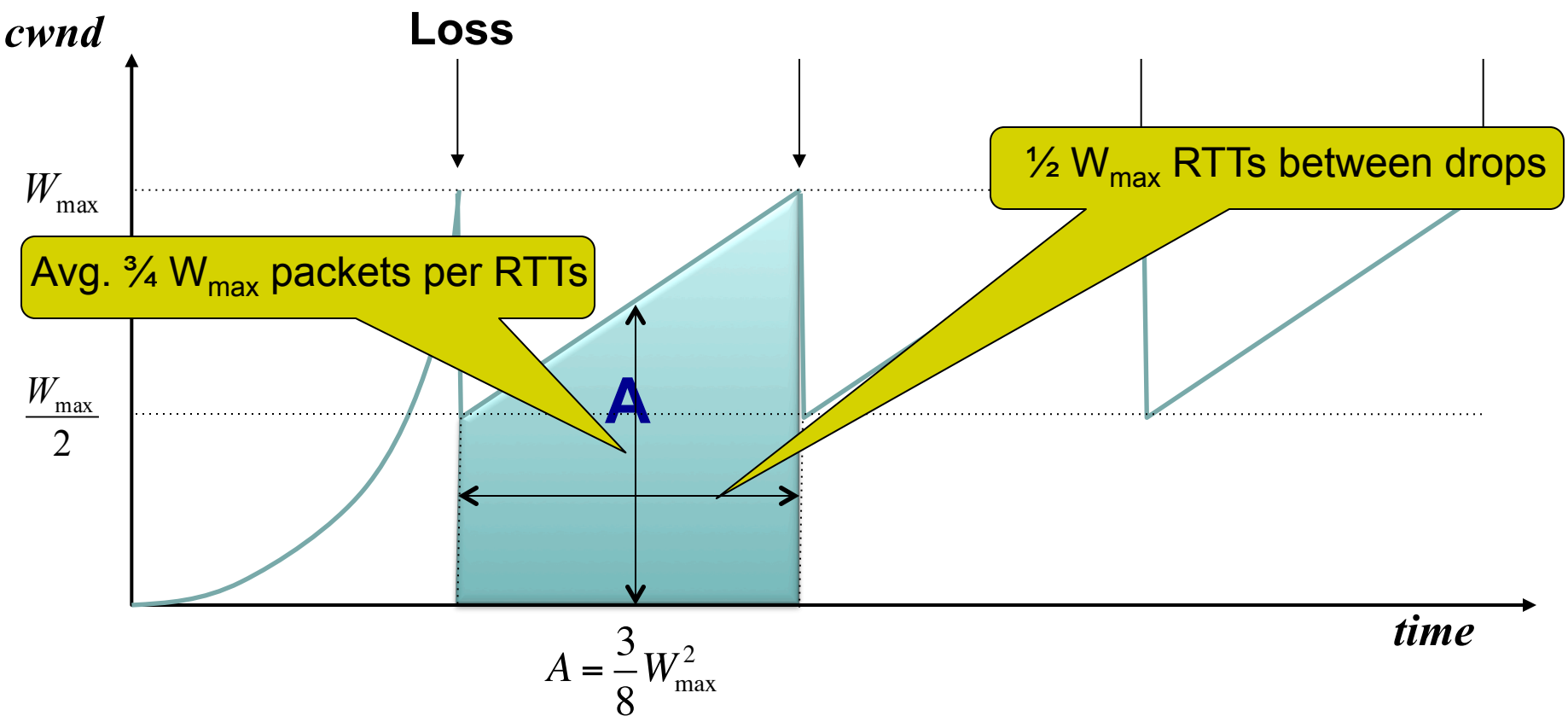
- Wrap-up CC details (fast retransmit)
- Critically examining TCP
- Advanced techniques

TCP Throughput Equation

A Simple Model for TCP Throughput



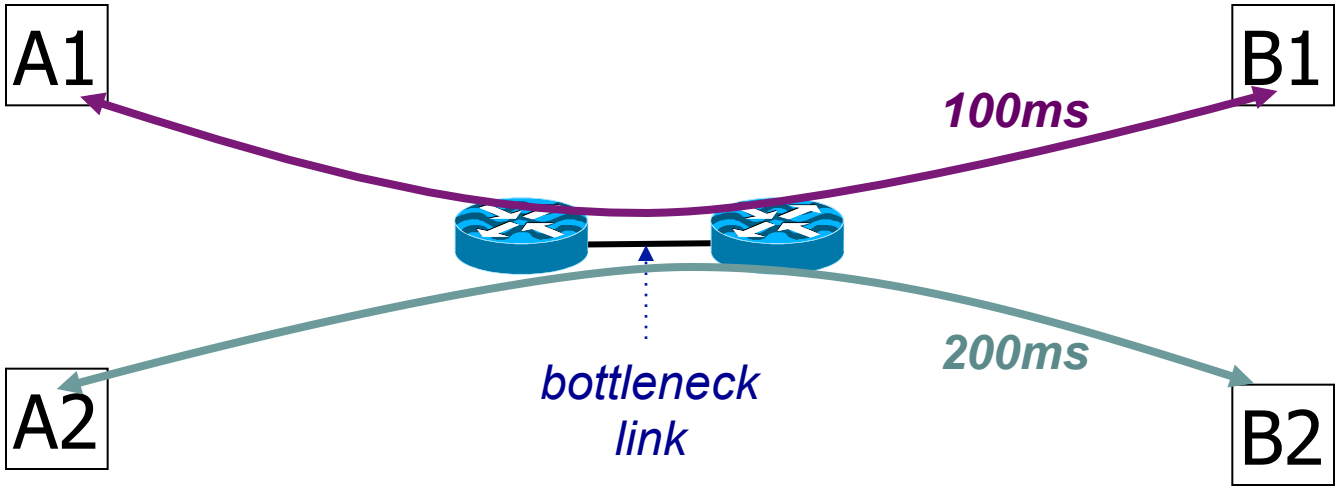
A Simple Model for TCP Throughput



Implications (1): Different RTTs

$$\text{Throughput} = \sqrt{\frac{3}{2}}\frac{1}{RTT\sqrt{p}}$$

- Flows get throughput inversely proportional to RTT
- TCP unfair in the face of heterogeneous RTTs!



Implications (2): High Speed TCP

$$\text{Throughput} = \sqrt{\frac{3}{2}} \frac{1}{RTT\sqrt{p}}$$

- Assume RTT = 100ms, MSS=1500bytes, BW=100Gbps
- What value of *p* is required to reach 100Gbps throughput?
  - ~ 2 x 10^{-12}
- How long between drops?
  - ~ 16.6 hours
- How much data has been sent in this time?
  - ~ 6 petabits
- These are not practical numbers!

Adapting TCP to High Speed

- Once past a threshold speed, increase CWND faster
  - A proposed standard [Floyd'03]: once speed is past some threshold, change equation to $p^{-.8}$ rather than $p^{-.5}$
  - Let the additive constant in AIMD depend on CWND
- Other approaches?
  - Multiple simultaneous connections (hack but works today)
  - Router-assisted approaches (will see shortly)

Implications (3): *Rate*-based CC

$$\text{Throughput} = \sqrt{\frac{3}{2}} \frac{1}{RTT\sqrt{p}}$$

- TCP throughput is "choppy"
  - repeated swings between W/2 to W
- Some apps would prefer sending at a steady rate
  - e.g., streaming apps
- A solution: "Equation-Based Congestion Control"
  - ditch TCP's increase/decrease rules and just follow the equation
  - measure drop percentage *p*, and set rate accordingly
- Following the TCP equation ensures we're "TCP friendly"
  - i.e., use no more than TCP does in similar setting

Other Limitations of TCP Congestion Control

(4) Loss not due to congestion?

- TCP will confuse corruption with congestion
- Flow will cut its rate
  - Throughput ~ 1/sqrt(p) where p is loss prob.
  - Applies even for non-congestion losses!
- We'll look at proposed solutions shortly…

(5) How do short flows fare?

- 50% of flows have < 1500B to send; 80% < 100KB
- Implication (1): short flows never leave slow start!
  - short flows never attain their fair share
- Implication (2): too few packets to trigger dupACKs
  - Isolated loss may lead to timeouts
  - At typical timeout values of ~500ms, might severely impact flow completion time

(6) TCP fills up queues → long delays

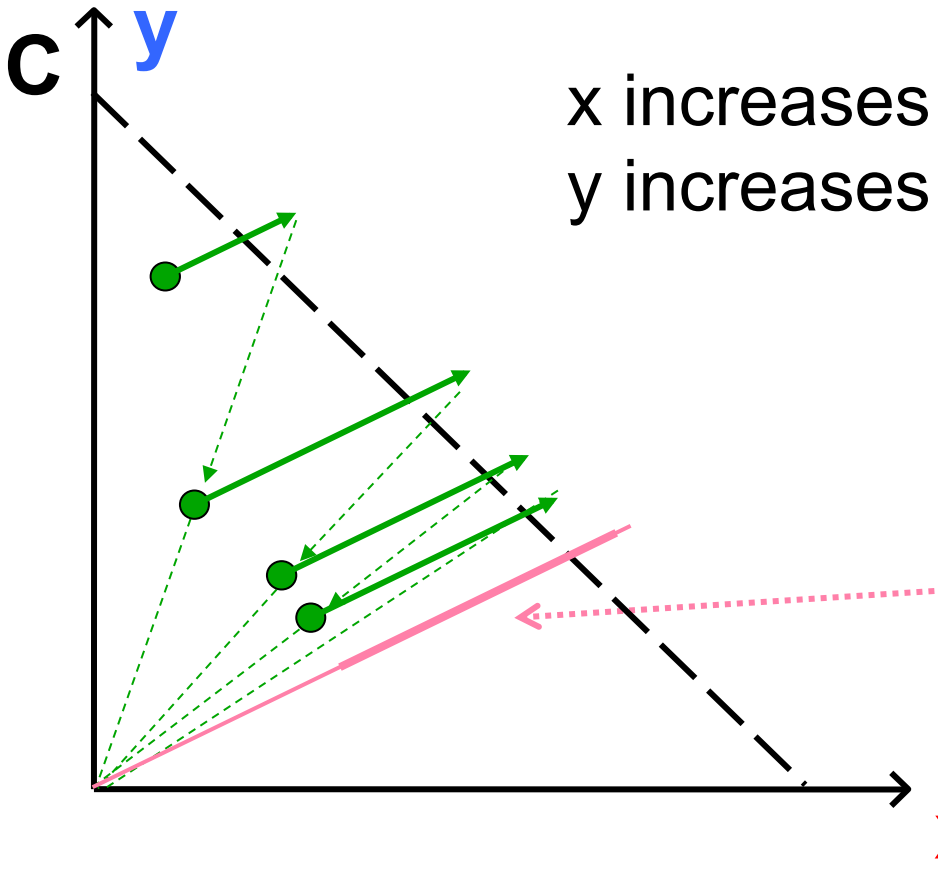
- A flow deliberately overshoots capacity, until it experiences a drop

- Means that delays are large, and are large for *everyone*
  - Consider a flow transferring a 10GB file sharing a bottleneck link with 10 flows transferring 100B

(7) Cheating

- Three easy ways to cheat
  - Increasing CWND faster than +1 MSS per RTT

Increasing CWND Faster



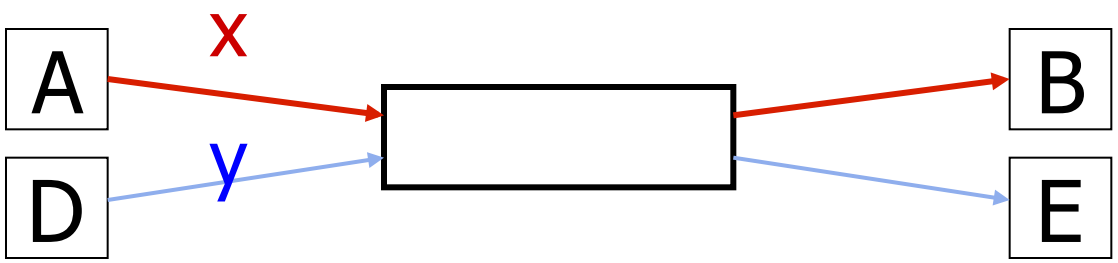
x increases by 2 per RTT
y increases by 1 per RTT

Limit rates:
x = 2y

(7) Cheating

- Three easy ways to cheat
  - Increasing CWND faster than +1 MSS per RTT
  - Opening many connections

Open Many Connections



Assume

- A starts 10 connections to B
- D starts 1 connection to E
- Each connection gets about the same throughput

Then A gets 10 times more throughput than D

(7) Cheating

- Three easy ways to cheat
  - Increasing CWND faster than +1 MSS per RTT
  - Opening many connections
  - Using large initial CWND

- Why hasn't the Internet suffered a congestion collapse yet?

(8) CC intertwined with reliability

- Mechanisms for CC and reliability are tightly coupled
  - CWND adjusted based on ACKs and timeouts
  - Cumulative ACKs and fast retransmit/recovery rules
- Complicates evolution
  - Consider changing from cumulative to selective ACKs
  - A failure of modularity, not layering
- Sometimes we want CC but not reliability
  - e.g., real-time applications
- Sometimes we want reliability but not CC (?)

Recap: TCP problems

- Misled by non-congestion losses
- Fills up queues leading to high delays
- Short flows complete before discovering available capacity
- AIMD impractical for high speed links
- Sawtooth discovery too choppy for some app
- Unfair under heterogeneous RTTs
- Tight coupling with reliability mechanisms
- Endhosts can cheat

Routers tell endpoints if they're congested

Routers tell endpoints what rate to send at

Routers enforce fair sharing

Could fix many of these with some help from routers!

Router-Assisted Congestion Control

- Three tasks for CC:
  - Isolation/fairness
  - Adjustment
  - Detecting congestion

How can routers ensure each flow gets its “fair share”?

Fairness: General Approach

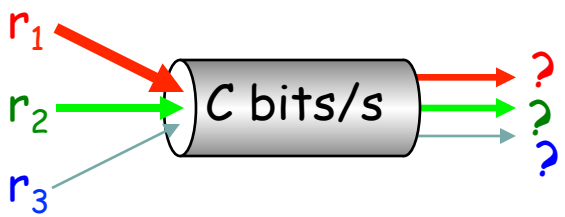
- Routers classify packets into “flows”
  - (For now) let’s assume flows are TCP connections
- Each flow has its own FIFO queue in router
- Router services flows in a fair fashion
  - When line becomes free, take packet from next flow in a fair order
- What does “fair” mean exactly?

Max-Min Fairness

- Given set of bandwidth demands r_i and total bandwidth C, max-min bandwidth allocations are:

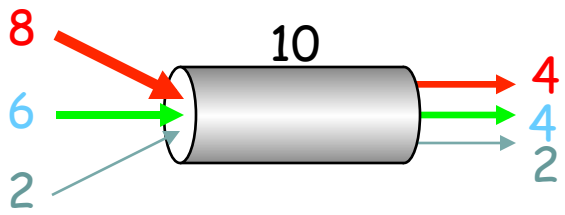
$$a_i = \min(f, r_i)$$

where f is the unique value such that Sum(a_i) = C



Example

- $C = 10$; $r_1 = 8$, $r_2 = 6$, $r_3 = 2$; $N = 3$
- $C/3 = 3.33 \rightarrow$
  - But r_3's need is only 2
  - Can service all of r_3
  - Remove r_3 from the accounting: $C = C - r_3 = 8$; $N = 2$
- $C/2 = 4 \rightarrow$
  - Can't service all of r_1 or r_2
  - So hold them to the remaining fair share: $f = 4$



$f = 4$:
min(8, 4) = 4
min(6, 4) = 4
min(2, 4) = 2

Max-Min Fairness

- Given set of bandwidth demands r_i and total bandwidth C, max-min bandwidth allocations are:

$$a_i = \min(f, r_i)$$

- where f is the unique value such that $\text{Sum}(a_i) = C$

- Property:
  - If you don't get full demand, no one gets more than you

- This is what round-robin service gives if all packets are the same size

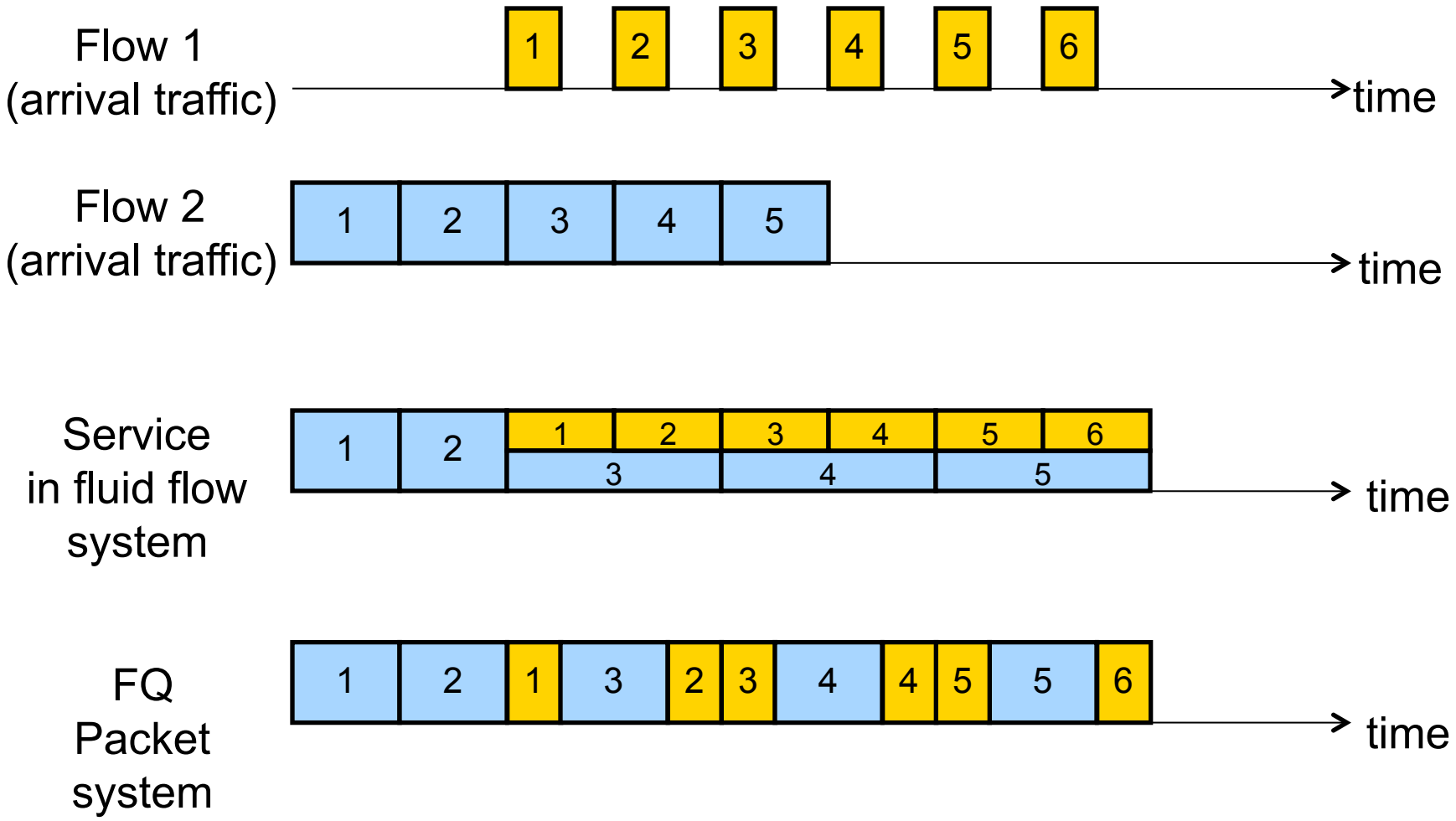
How do we deal with packets of different sizes?

- Mental model: Bit-by-bit round robin (“fluid flow”)
- Can you do this in practice?
- No, packets cannot be preempted
- But we can approximate it
  - This is what “fair queuing” routers do

Fair Queuing (FQ)

- For each packet, compute the time at which the last bit of a packet would have left the router *if* flows are served bit-by-bit
- Then serve packets in the increasing order of their deadlines

Example


Flow 1
(arrival traffic)
1
2
3
4
5
6
time
Flow 2
(arrival traffic)
1
2
3
4
5
time
Service
in fluid flow
system
1
2
1
2
3
4
5
6
3
4
5
time
FQ
Packet
system
1
2
1
3
2
3
4
4
5
5
6
time

Fair Queuing (FQ)

- Think of it as an implementation of round-robin generalized to the case where not all packets are equal sized
- Weighted fair queuing (WFQ): assign different flows different shares
- Today, some form of WFQ implemented in almost all routers
  - Not the case in the 1980-90s, when CC was being developed
  - Mostly used to isolate traffic at larger granularities (e.g., per-prefix)

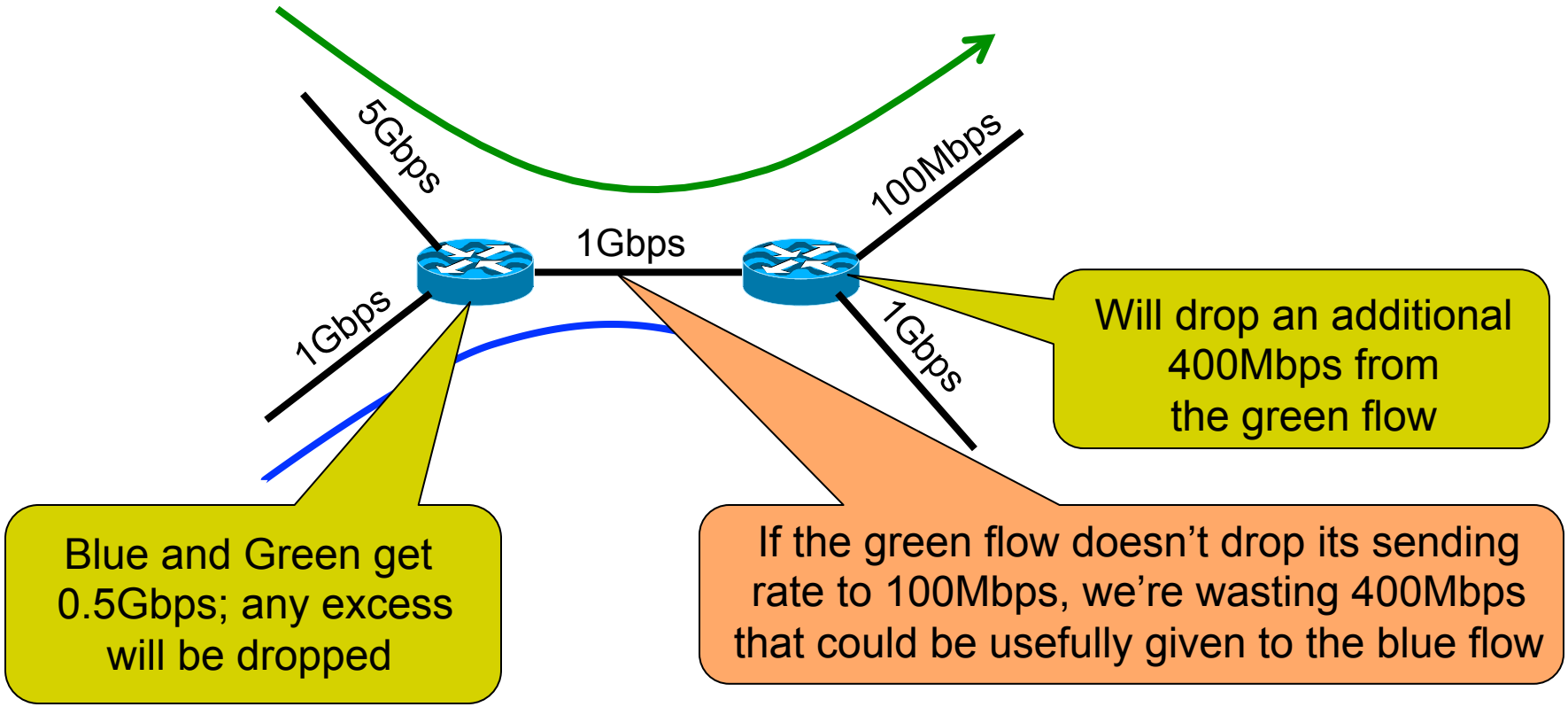
FQ vs. FIFO

- FQ advantages:
  - Isolation: cheating flows don't benefit
  - Bandwidth share does not depend on RTT
  - Flows can pick any rate adjustment scheme they want

- Disadvantages:
  - More complex than FIFO: per flow queue/state, additional per-packet book-keeping

FQ in the big picture

- FQ does not eliminate congestion → it just manages the congestion



FQ in the big picture

- FQ does not eliminate congestion → it just manages the congestion
  - robust to cheating, variations in RTT, details of delay, reordering, retransmission, *etc.*
- But congestion (and packet drops) still occurs
- And we still want end-hosts to discover/adapt to their fair share!
- What would the end-to-end argument say w.r.t. congestion control?

Fairness is a controversial goal

- What if you have 8 flows, and I have 4?
  - Why should you get twice the bandwidth
- What if your flow goes over 4 congested hops, and mine only goes over 1?
  - Why shouldn't you be penalized for using more scarce bandwidth?
- And what is a flow anyway?
  - TCP connection
  - Source-Destination pair?
  - Source?

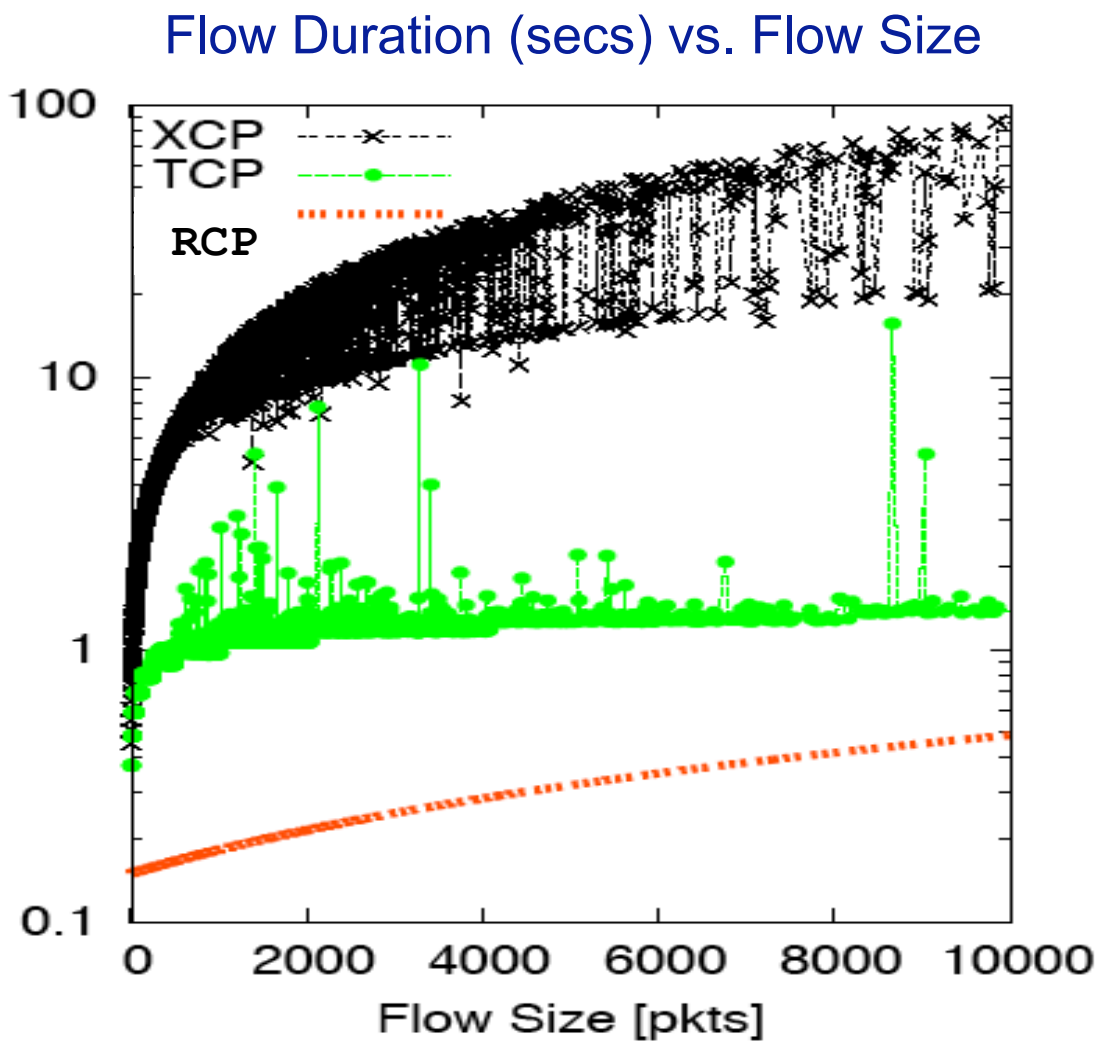
Router-Assisted Congestion Control

- CC has three different tasks:
  - Isolation/fairness
  - Rate adjustment
  - Detecting congestion

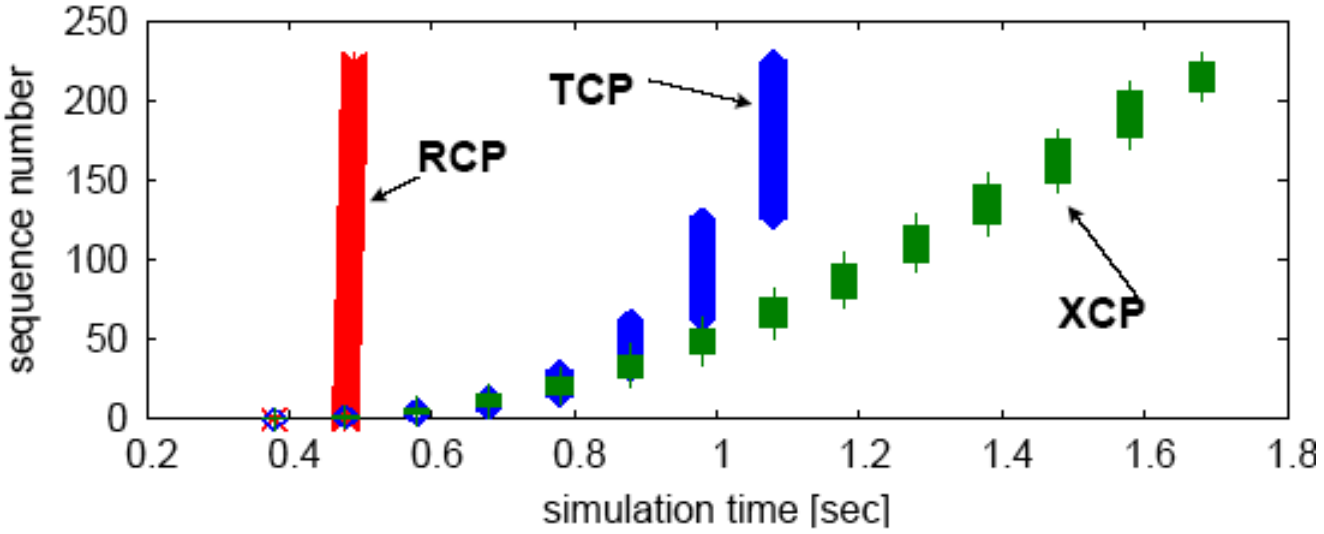
Why not just let routers tell endhosts what rate they should use?

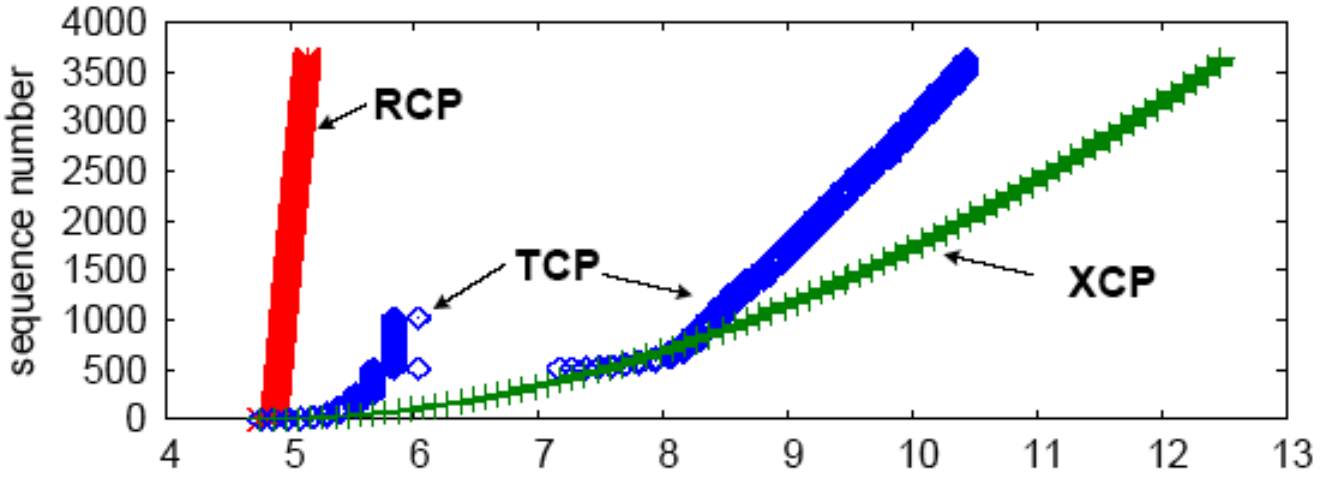
- Packets carry “rate field”
- Routers insert “fair share” *f* in packet header
- End-hosts set sending rate (or window size) to *f*
  - hopefully (still need some policing of endhosts!)
- This is the basic idea behind the “Rate Control Protocol” (RCP) from Dukkipati *et al.* ’07

Flow Completion Time: TCP vs. RCP (Ignore XCP)



Why the improvement?





Router-Assisted Congestion Control

- CC has three different tasks:
  - Isolation/fairness
  - Rate adjustment
  - Detecting congestion

Explicit Congestion Notification (ECN)

- Single bit in packet header; set by congested routers
  - If data packet has bit set, then ACK has ECN bit set
- Many options for when routers set the bit
  - tradeoff between (link) utilization and (packet) delay
- Congestion semantics can be exactly like that of drop
  - I.e., endhost reacts as though it saw a drop

- Advantages:
  - Don’t confuse corruption with congestion; recovery w/ rate adjustment
  - Can serve as an early indicator of congestion to avoid delays
  - Easy (easier) to incrementally deploy
    - Today: defined in RFC 3168 using ToS/DSCP bits in the IP header

One final proposal: Charge people for congestion!

- Use ECN as congestion markers
- Whenever I get an ECN bit set, I have to pay $$
- Now, there's no debate over what a flow is, or what fair is…
- Idea started by Frank Kelly at Cambridge
  - "optimal" solution, backed by much math
  - Great idea: simple, elegant, effective
  - Unclear that it will impact practice

Recap

- TCP:
  - somewhat hacky
  - but practical/deployable
  - good enough to have raised the bar for the deployment of new, more optimal, approaches
  - though the needs of datacenters might change the status quo (future lecture)