TCP Congestion Control (contd.)

CS 168

http://cs168.io

Sylvia Ratnasamy

Today

- How TCP implements CC
- Modeling TCP throughput
- Critiquing TCP
- Router-assisted CC (briefly)

Recall: Sketch of TCP's solution

Each source independently runs the following:

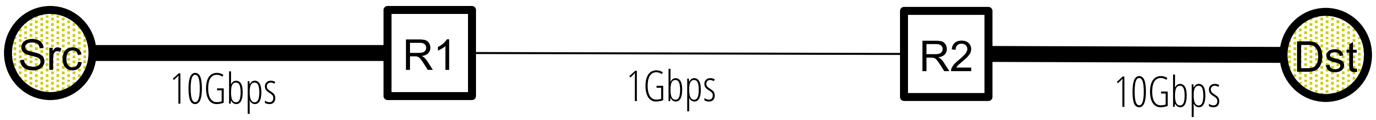
- Slow-start to find initial rate
- Try sending at a rate R for some time period
  - Did I experience ~~congestion~~ loss in this time period?
    - If yes, reduce R multiplicatively (2x)
    - If no, increase R additively (+1)
  - Repeat

A TCP sender implements rate adaptation by adjusting its CWND

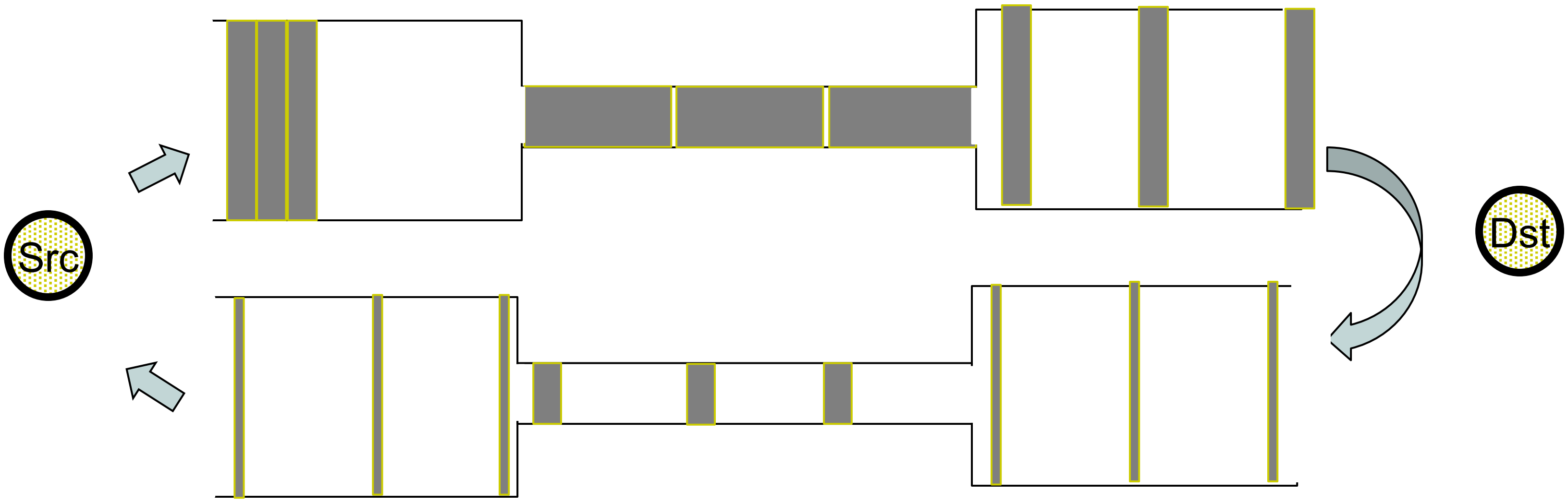
TCP Implementation

- Essential idea: update CWND based on events at sender
- Events → arrival (or absence) of **ACKs**
- Leads to TCP's "ACK clocked" transmission behavior
  - I.e., ACK events trigger the next transmission

ACK Clocking



ACK Clocking

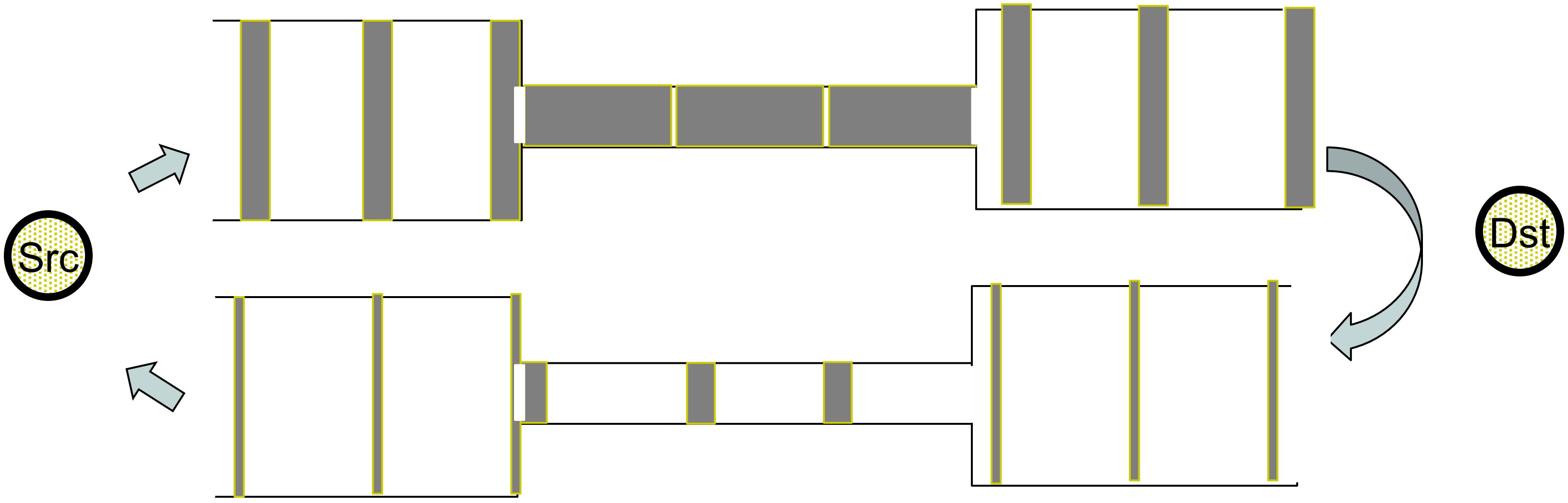


Consider: source sends a burst of packets

Packets are queued and "spread out" at slow link

ACKs maintain the spread on the return path

ACK Clocking



Sender clocks new packets with the spread

Now sending without queuing at the bottleneck link!

TCP Implementation

- Essential idea: update CWND based on events at sender
- Events → arrival (or absence) of **ACKs**
- Leads to TCP's "ACK clocked" transmission behavior
  - I.e., ACK events trigger the next transmission
- Without loss, #events per RTT ~ #packets per CWND

TCP Implementation

- **State at sender**
  - CWND (initialized to a 1 MSS)
  - SSTHRESH (initialized to a large constant)
  - dupACKcount (initialized to zero, as before)
  - Timer (as before)
- **Events at sender**
  - ACK (for new data)
  - dupACK (duplicate ACK for old data)
  - Timeout
- What about receiver?
  - Just send ACKs like before

Event: ACK (new data)

- If in slow start
  - CWND += 1 (MSS)

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

Event: ACK (new data)

- If in slow start
  - CWND += 1 (MSS)

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

- Plus the usual ...
  - Reset timer, dupACKcount
  - Send new data packets (if CWND allows)

Event: TimeOut

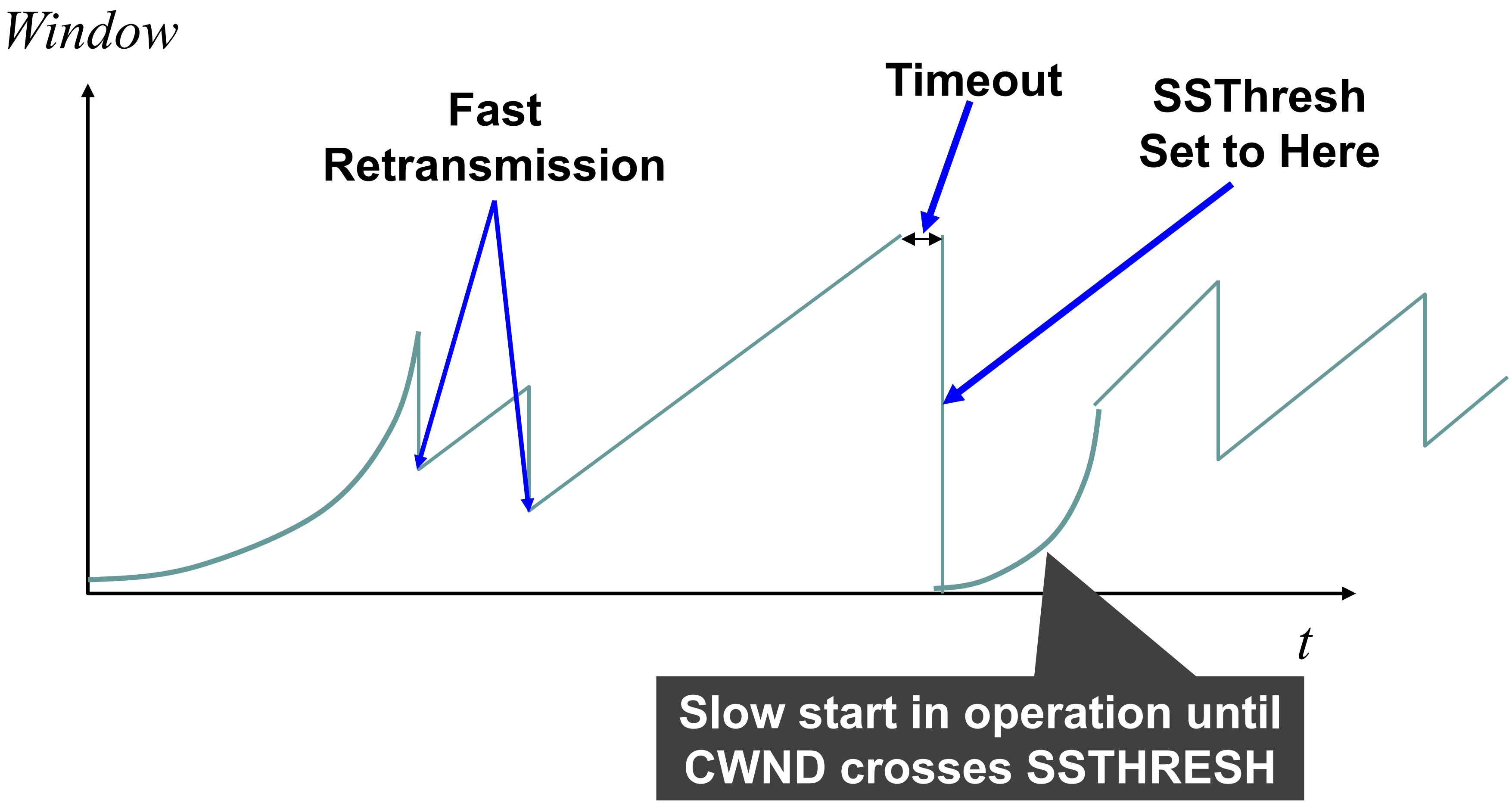
- On Timeout
  - SSTHRESH ← CWND/2
  - CWND ← 1
  - And retransmit packet (as always)

Event: dupACK

- dupACKcount ++
- If dupACKcount = 3 /* fast retransmit */
  - SSTHRESH = CWND/2
  - CWND = CWND/2 (but never less than 1)
  - And retransmit packet (as always)

Remain in AIMD
after fast retransmission…

Time Diagram



Any Questions?

One Final Phase: Fast Recovery

- The problem: congestion avoidance too slow in recovering from an isolated loss
- This last feature is an optimization to improve performance
  - Bit of a hack, but effective

Example

- Again: counting packets, not bytes
  - If you want example in bytes, assume MSS=1000 and add three zeros to all sequence numbers

- Consider a TCP connection with:
  - CWND=10 packets
  - Last ACK was for packet # 101
    - i.e., receiver expecting next packet to have seq. no. 101

- 10 packets [101, 102, 103,…, 110] are in flight
  - Packet 101 is dropped
  - What ACKs do they generate and how does the sender respond?

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 (no xmit)
- ACK 101 (due to 106) cwnd=5 (no xmit)
- ACK 101 (due to 107) cwnd=5 (no xmit)
- ACK 101 (due to 108) cwnd=5 (no xmit)
- ACK 101 (due to 109) cwnd=5 (no xmit)
- ACK 101 (due to 110) cwnd=5 (no xmit)
- ACK 111 (due to 101) ← only now can we transmit new packets
- Plus no packets in flight so no additional ACKs for another RTT

Note that you do not restart dupACK counter on same packet!

Two Questions

- Do you understand the problem?
  - Have to wait a long time before sending again
  - When you finally send, you immediately send a full window then wait an RTT

- How would you fix it?

Solution: Fast Recovery

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

- I.e., temporarily inflate CWND

- If dupACKcount = 3
  - SSTHRESH = CWND/2
  - CWND = SSTHRESH + 3
- While in fast recovery
  - CWND = CWND + 1 (MSS) for each additional duplicate ACK
  - This allows source to send an additional packet…
  - …to compensate for the packet that arrived (generating dupACK)
- Exit fast recovery after receiving new ACK
  - set CWND = SSTHRESH

Timeline (at sender)

In flight: ~~101~~, 102, 103, 104, 105, 106, 107, 108, 109, 110 101 111, 112, ...

- ACK 101 (due to 102) cwnd=10 dupACK#1
- ACK 101 (due to 103) cwnd=10 dupACK#2
- ACK 101 (due to 104) cwnd=10 dupACK#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 (and now sending 115)
- ACK 112 (due to 111) cwnd = 5 + 1/5 ← back in congestion avoidance

Updated Event-Actions

Event: ACK (new data)

- If in slow start
  - CWND += 1 (MSS)

Slow start phase

- If in fast recovery
  - CWND = SSTHRESH

Leaving Fast Recovery

- Else
  - CWND = CWND + 1/CWND

"Congestion Avoidance" phase (additive increase)

- Plus the usual...

Event: dupACK

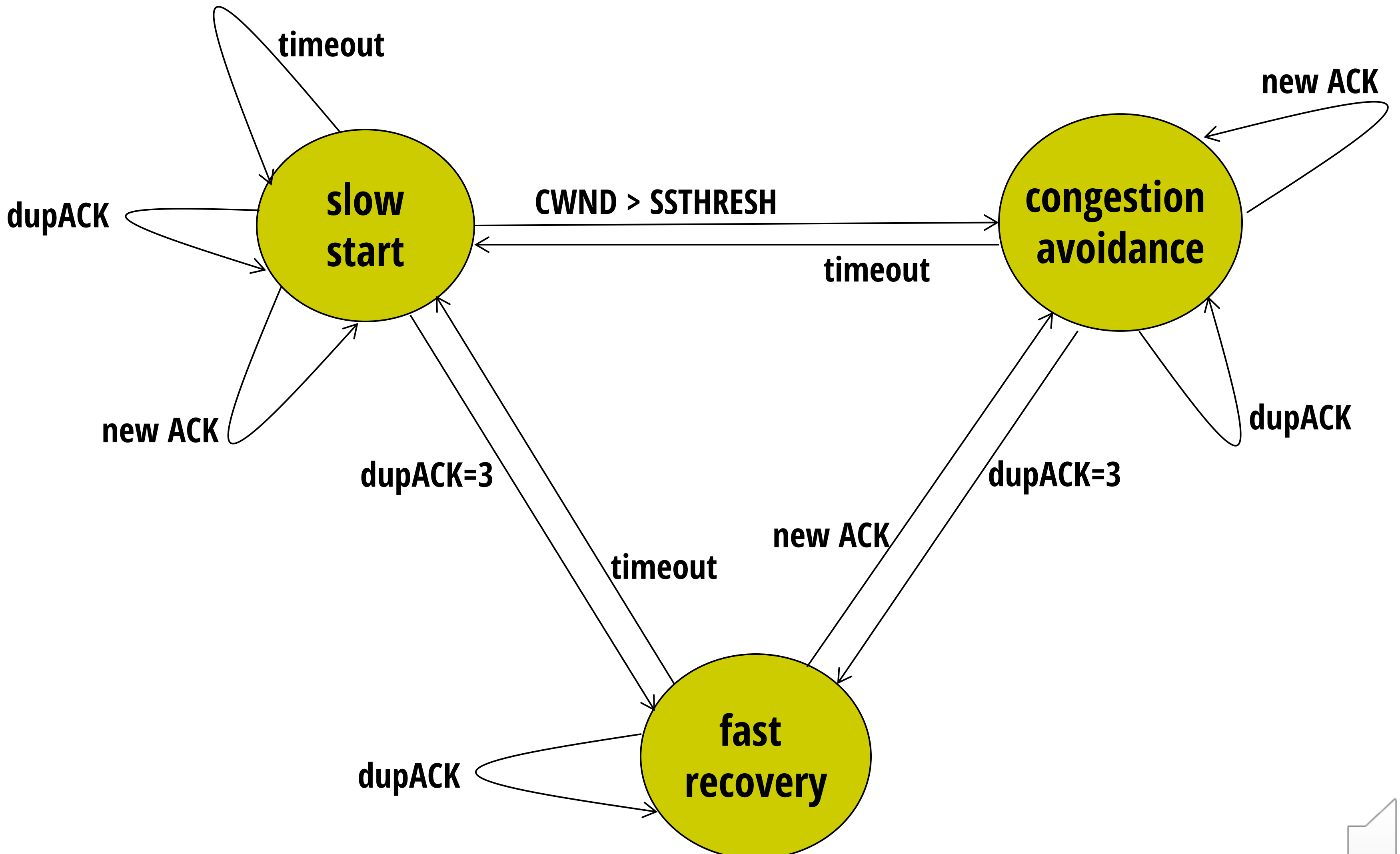
- dupACKcount ++

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

- If dupACKcount > 3 /* fast recovery */
  - CWND = CWND + 1 (MSS)

Next: TCP State Machine

TCP State Machine



Many variants

- 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"
  - ACKs describe byte ranges received

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?

TCP Throughput Equation

TCP Throughput

- Given a path, what TCP throughput can we expect?
- We'll derive a simple model that expresses TCP throughput in terms of path properties:
  - RTT
  - Loss rate, p

A Simple Model for TCP Throughput

- Assume loss occurs whenever CWND reaches W_{max}
- And is detected by duplicate ACKs (i.e., no timeouts)

- Hence, evolution of window size:
  - $\frac{1}{2}W_{max}$ (after detecting loss)
  - $\frac{1}{2}W_{max}$ +1 (one RTT later)
  - $\frac{1}{2}W_{max}$ +2 (two RTTs later)
  - $\frac{1}{2}W_{max}$ +3 (three RTTs later)
  - ...
  - W_{max} [drop]
  - $\frac{1}{2}W_{max}$
  - $\frac{1}{2}W_{max}$ +1

Note: number of RTTs between drops is $\frac{1}{2}W_{max}$

A Simple Model for TCP Throughput

- Assume loss occurs whenever CWND reaches W_{max}
- And is detected by duplicate ACKs (i.e., no timeouts)

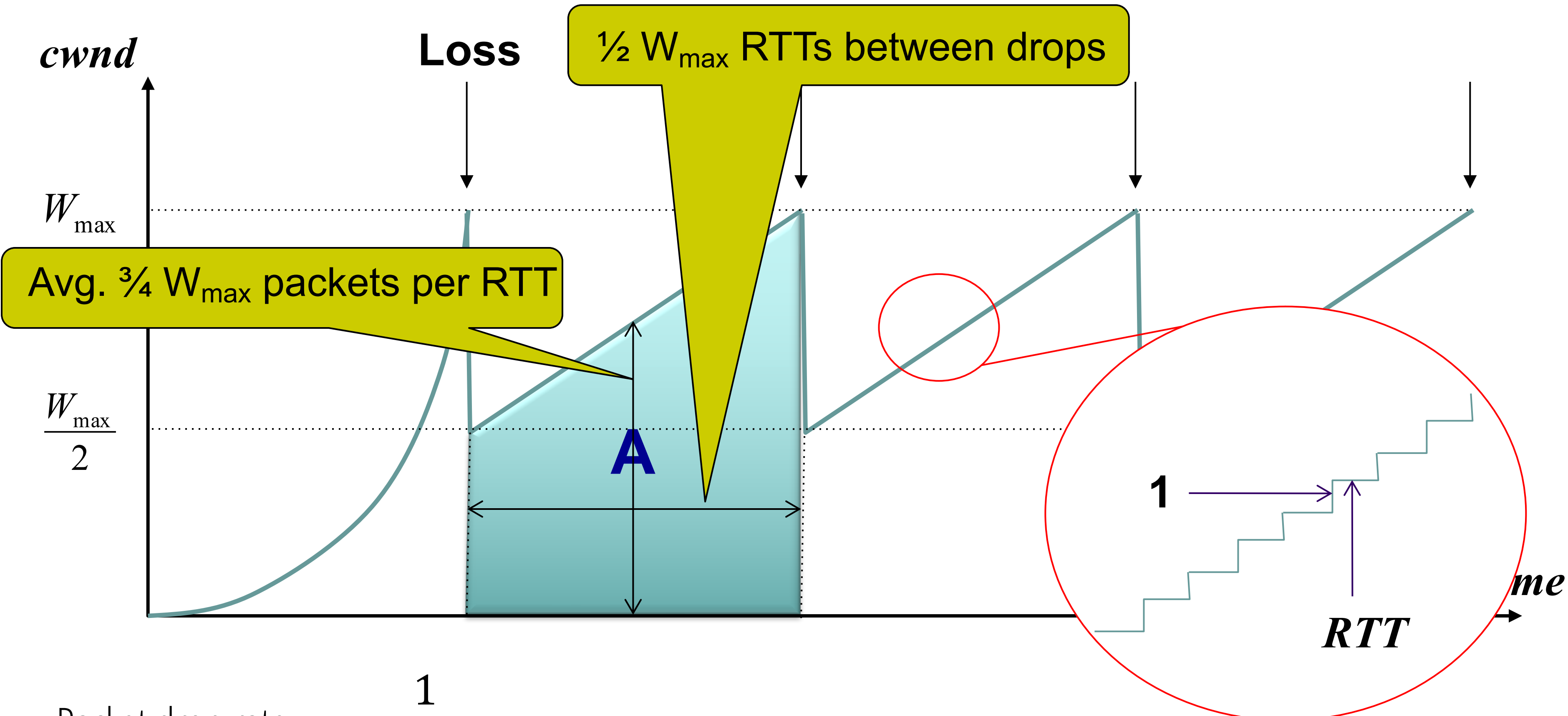
- Hence, evolution of window size:
  - Increase by 1 for each of $\frac{1}{2}W_{max}$ RTTs, then drop, then repeat

- Average window size per RTT = $\frac{3}{4}W_{max}$

- Average throughput = $\frac{3}{4}W_{max}$ x $\frac{MSS}{RTT}$

- Remaining step: express W_{max} in terms of loss rate p

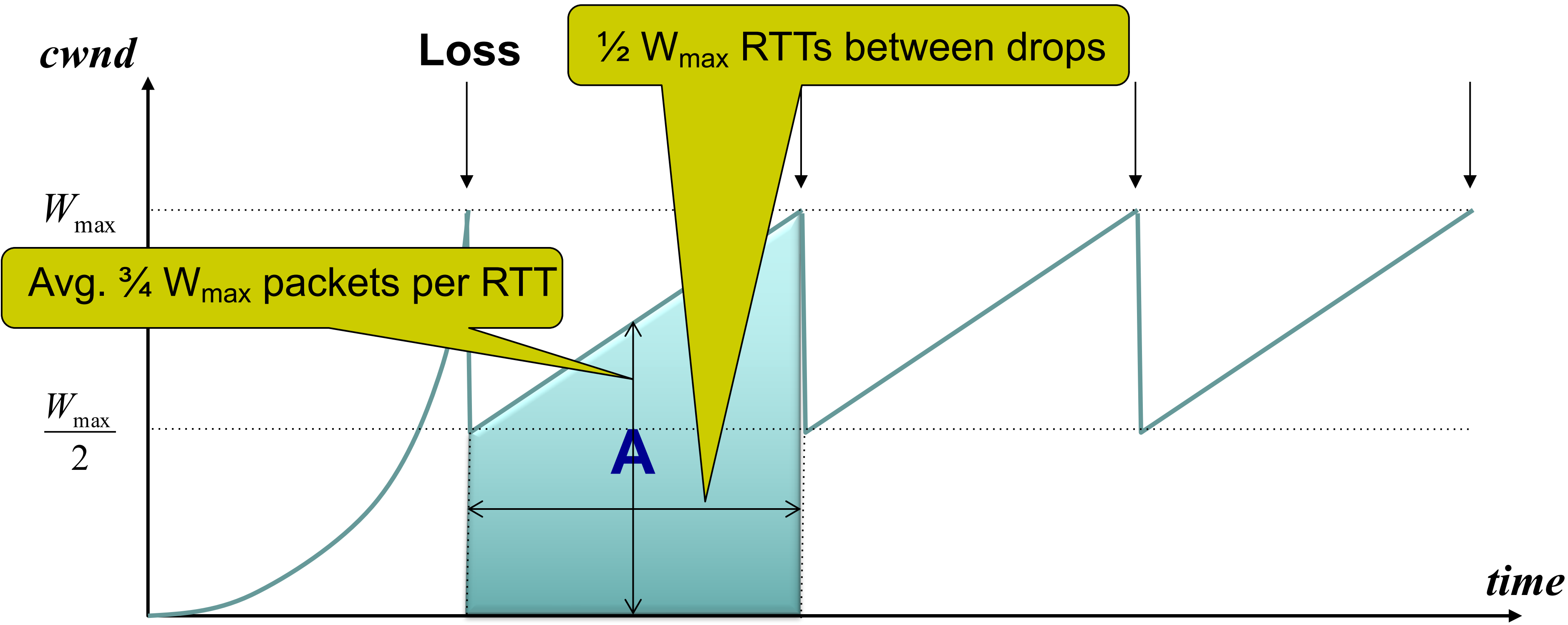
A Simple Model for TCP Throughput



Packet drop rate, $p = \frac{1}{A}$

On average, one of all packets in shaded region is lost (i.e., loss rate is 1/A, where A is #packets in shaded region)

A Simple Model for TCP Throughput



Packet drop rate, $p = \frac{1}{A}$ **A** = $\frac{3}{8} W_{max}^2$ $\rightarrow W_{max} = \frac{2\sqrt{2}}{\sqrt{3p}}$

Average Throughput = $\frac{\frac{3}{4} W_{max} \times MSS}{RTT} = \sqrt{\frac{3}{2}} \frac{\mathrm{MSS}}{\mathrm{RTT}\sqrt{p}}$

TCP Throughput

- Given a path, what TCP throughput can we expect?

- TCP throughput is proportional to $\frac{1}{\mathrm{RTT}}$ and $\frac{1}{\sqrt{p}}$
  - RTT is path round-trip time and p is the packet loss rate

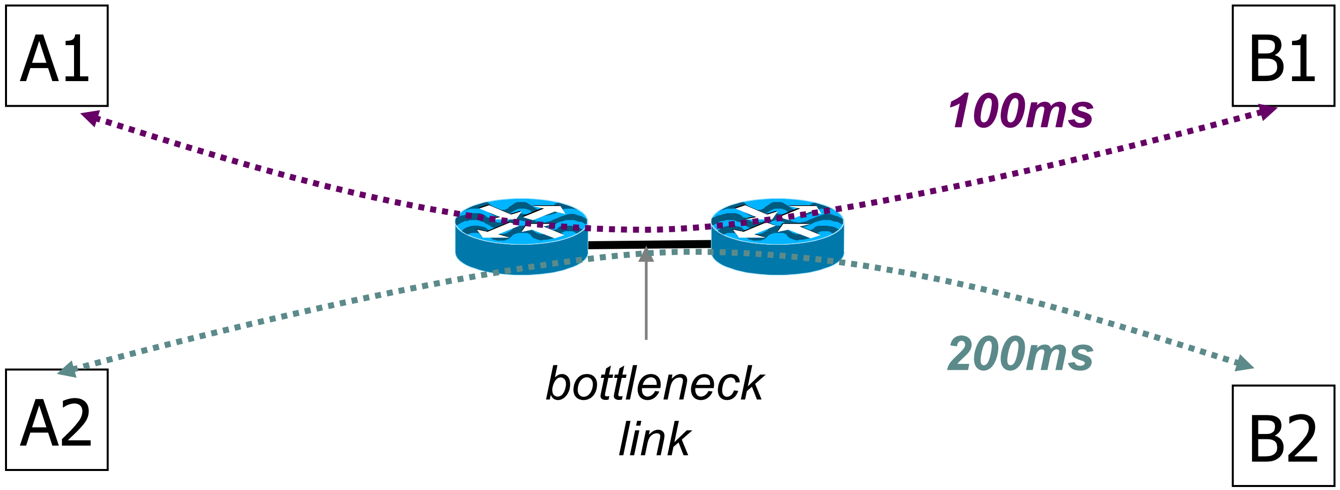
- Model makes many simplifying assumptions
  - Ignores slow-start, assumes fixed RTT, isolated loss, *etc.*

- But leads to some insights (coming up)

Implications (1): Different RTTs

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

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



Implications (2): *Rate*-based CC [RFC 5348]

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

- 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 RTT and 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

(3) Loss not due to congestion?

- TCP will confuse corruption with congestion
- Flow will cut its rate
  - Throughput ~ $\frac{1}{\sqrt{p}}$ even for non-congestion losses!

(4) How do short flows fare?

- 50% of flows have < 1500B to send; 80% < 100KB

- Implication (1): many flows never leave slow start!
  - Short flows never attain their fair share
  - In fact, short flows are likely to suffer unduly long transfer times

- 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

- A partial fix: use a higher initial CWND [RFC IW10]

(5) TCP fills up queues → long delays

- A flow deliberately overshoots capacity, until it experiences a drop
- Recall: loss follows delay (i.e,. queue *must* fill up)
- Means that delays are large, for *everyone*
  - Consider a flow transferring a 10GB file sharing a bottleneck link with 10 flows transferring 100B
  - Problem exacerbated when we have large amounts of memory on routers (a.k.a. “bufferbloat”)

(5) TCP fills up queues → long delays

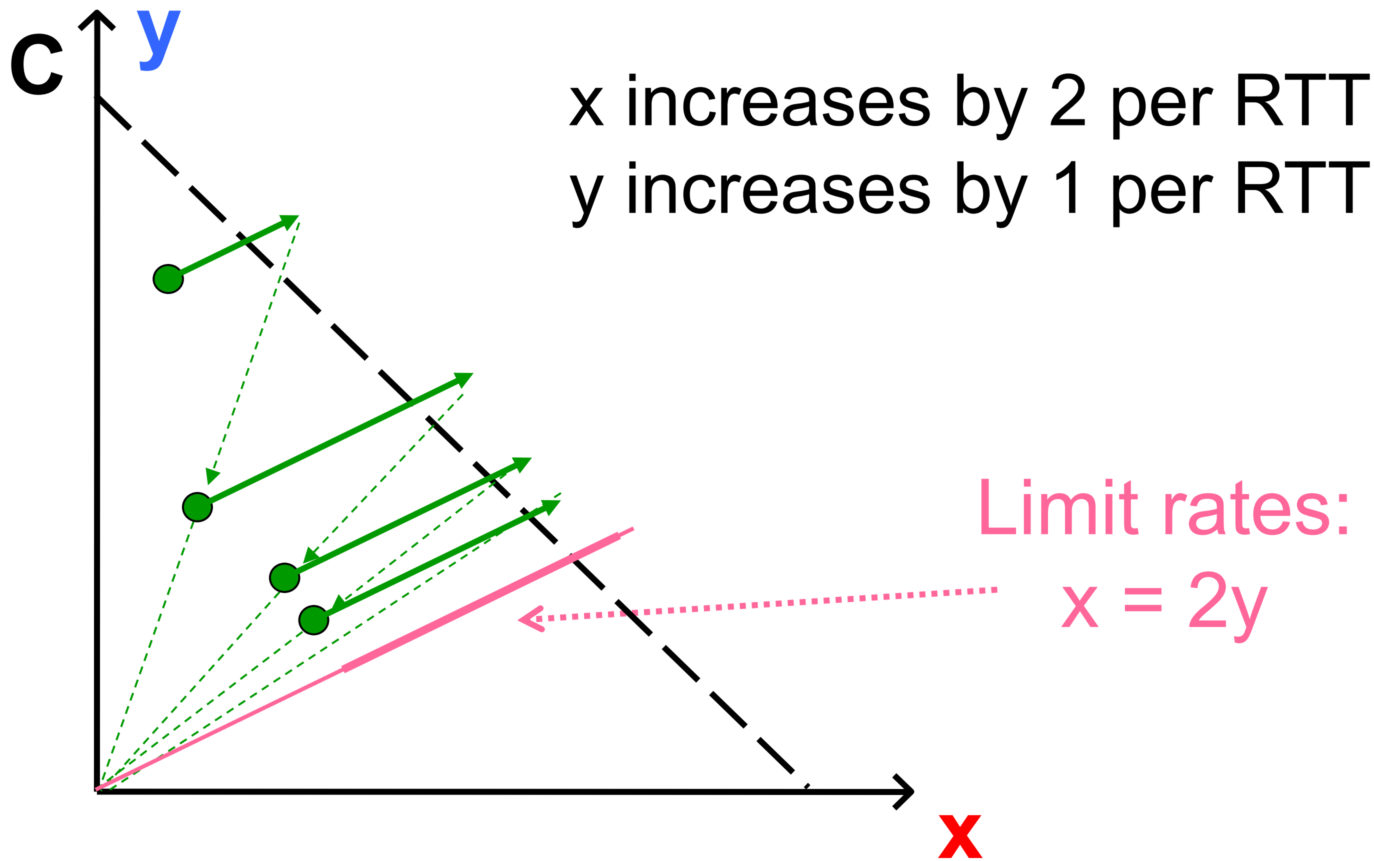
- Focus of Google's BBR algorithm[1]
- Basic idea (simplified):
  - Sender learns its minimum RTT (~ propagation RTT)
  - Decreases its rate when the observed RTT exceeds the minimum RTT

[1] *BBR: Congestion-Based Congestion Control; Cardwell et al, ACM Queue 2016*

(6) Cheating

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

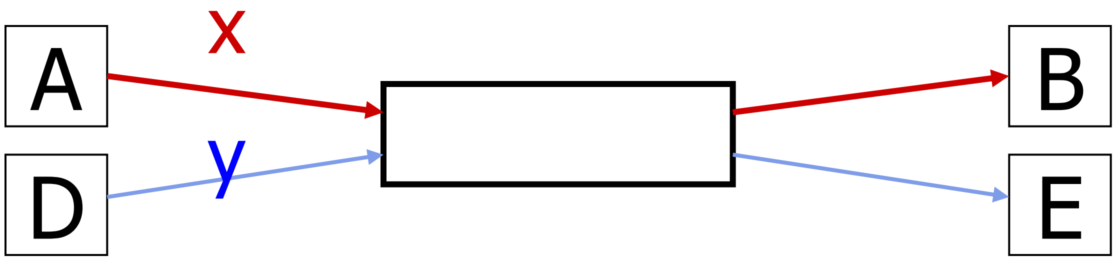
Increasing CWND Faster



(6) 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

(6) Cheating

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

Why hasn't the Internet suffered another congestion collapse?

- Even "cheaters" do back off!
  - Leads to unfairness, not necessarily collapse
- Hard to say whether unfair behavior is common

(7) 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 audio/video
- Sometimes we want reliability but not CC

Recap: TCP problems

Routers tell endhosts about congestion (fine- or coarse-grained feedback)

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

Routers enforce fair sharing

Could fix many of these with some help from routers!

Router-Assisted Congestion Control

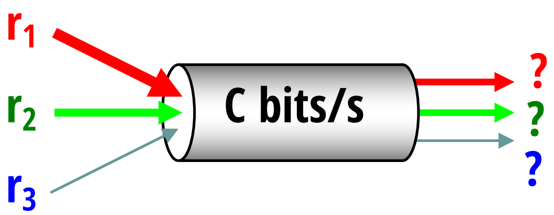
- Three ways routers can help
  - Enforce fairness
  - More precise rate adaptation
  - Detecting congestion

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

Fairness: General Approach

- Consider a single router’s actions
- Router classifies incoming packets into “flows”
  - (For now) let’s assume flows are TCP connections
- Each flow has its own FIFO queue in router
- Router picks a queue (i.e., flow) in a fair order; transmits packet from the front of the queue
- What does “fair” mean exactly?

Max-Min Fairness



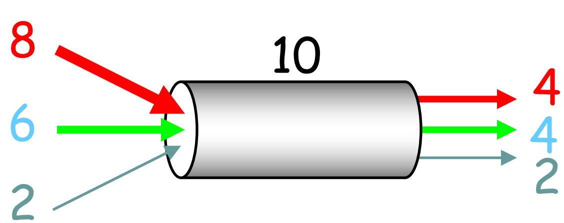
- Total available bandwidth C
- Each flow i has bandwidth demand r_i
- What is a fair allocation a_i of bandwidth to each flow i ?
- Max-min fair bandwidth allocations are:

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

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

Example

- C = 10; N = 3; r_1 = 8, r_2 = 6, r_3 = 2
- C/N = 10/3 = 3.33 $\rightarrow$
  - But r_3's need is only 2
  - Can service all of r_3
  - Allocate 2 to r_3 and remove it from 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

- 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”)
- Cannot do this in practice!
- 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 (called “deadlines”)

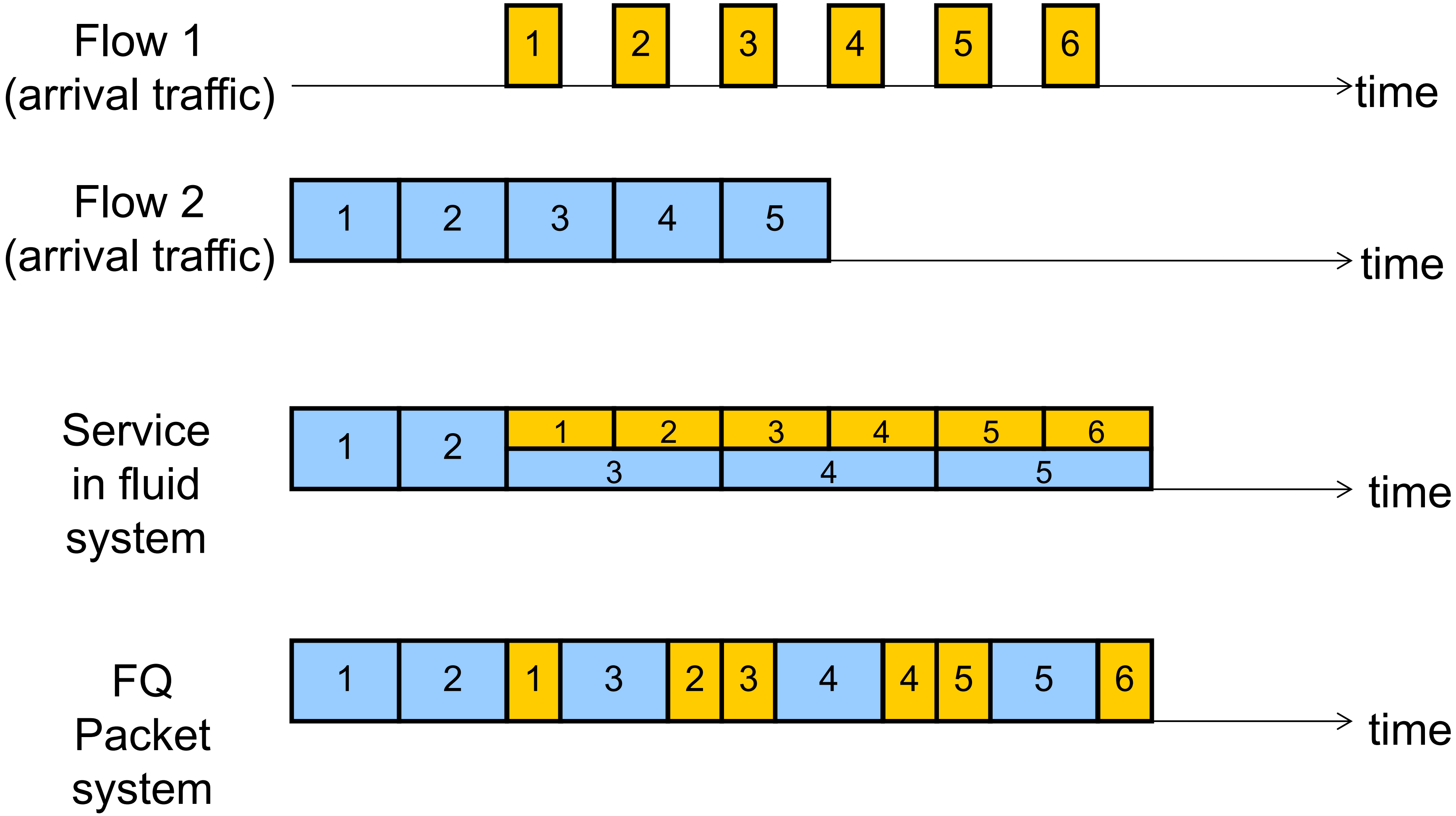
- Then serve packets in increasing order of their deadlines

- Think of it as an implementation of round-robin extended to the case where not all packets are equal sized

Analysis and Simulation of a Fair Queueing Algorithm

Alan Demers
Srinivasan Keshav†
Scott Shenker

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
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

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
  - Still only a partial solution (coming up)

Fair Queuing In Practice

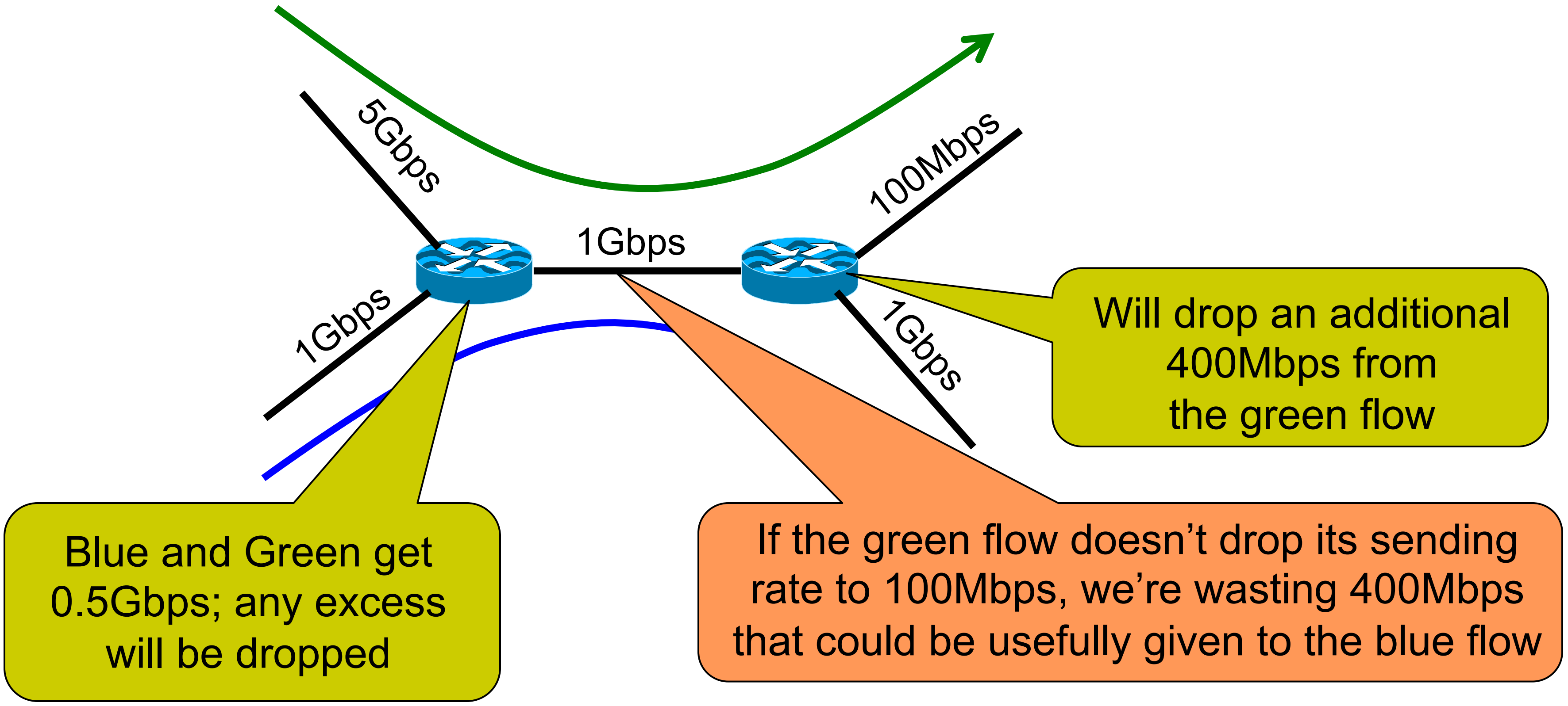
- "Pure" FQ too complex to implement at high speeds

- But several approximations exist
  - E.g., Deficit Round Robin (DRR)

- Today:
  - Routers typically implement approximate FQ (e.g., DRR)
  - For a small number of queues
  - Commonly used for coarser-grained isolation (e.g., per customer)

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
- FQ's benefit is its resilience to cheating, variations in RTT, details of delay, reordering, *etc.*
- But we still want end-hosts to discover/adapt to their fair share!

Per-flow 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?
  - Shouldn’t you be penalized for using more of scarce bandwidth?
- And at what granularity do we really want fairness?
  - TCP connection? Source-Destination pair? Source?
- Nonetheless, FQ/DRR is a great way to ensure **isolation**
  - Avoiding starvation even in the worst cases

Router-Assisted Congestion Control

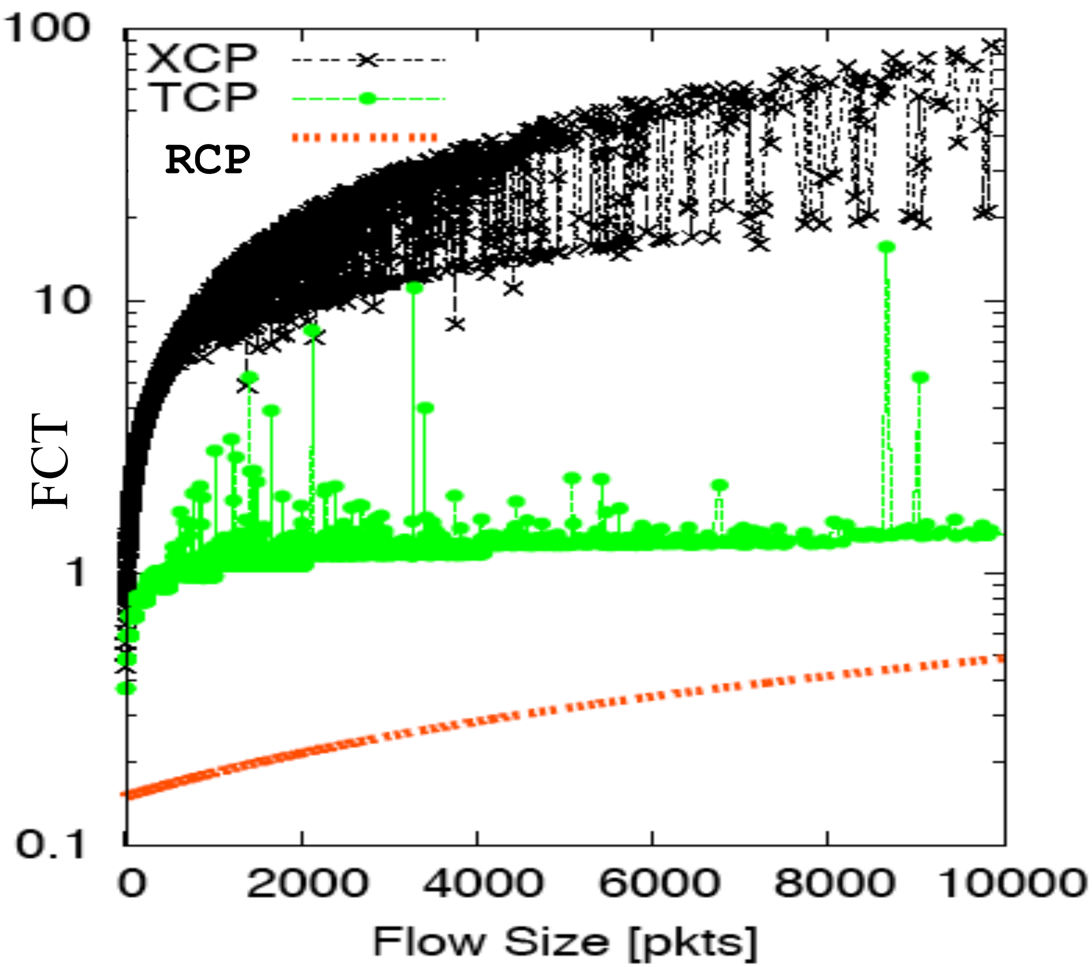
- Three ways routers can help
  - Enforce fairness
  - More precise rate adaptation
  - Detecting congestion

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

- Packets carry "rate field"
- Routers insert a flow's fair share *f* in packet header
- End-hosts set sending rate (or window size) to *f*
  - Essential idea behind the "Rate Control Protocol (RCP)"

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

Flow Completion Time (secs) vs. Flow Size



Router-Assisted Congestion Control

- Three ways routers can help
  - Enforce fairness
  - More precise rate adaptation
  - 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
- Host can react as though it was a drop

- Advantages:
  - Don't confuse corruption with congestion
  - Early indicator of congestion → avoid delays
  - Lightweight to implement

- Today:
  - Widely implemented in routers
  - Commonly used in datacenters

Final idea: Congestion-Based Charging

- Use ECN as congestion markers
- Whenever I get an ECN bit set, I have to pay $$
  - The more congested the network, the more I pay
- No debate over what a flow is, or what fair is…

Recap: Router-Assisted CC

- FQ: routers *enforce* per-flow fairness
- RCP: routers *inform* endhosts of their fair share
- ECN: routers set "I'm congested" bit in packets
- Congestion pricing: users pay based on congestion

Perspective: Router-Assisted CC

- Can be highly effective, approaching optimal perf.
- But deployment is more challenging
  - Need support at hosts and routers
  - Some require more complex book-keeping at routers
  - Some require deployment at *every* router
- Less challenging in datacenter contexts

Perspective: TCP CC

- Not perfect, a little ad-hoc
- But deeply practical/deployable
- Good enough to have raised the bar for the deployment of new, more optimal, approaches
- Though datacenters are reshaping the CC agenda
  - different needs and constraints (upcoming lecture)