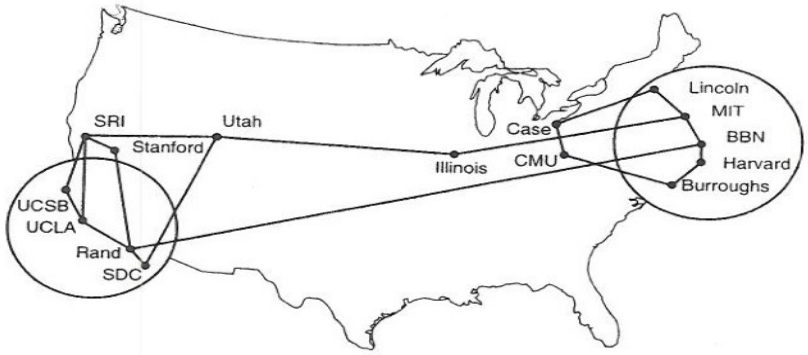
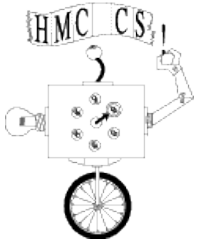




Congestion Control

Outline

- Queuing Discipline
- Reacting to Congestion
- Avoiding Congestion

Readings:

Chapter 6

Goals of Today's Lecture

- Principles of congestion control
  - Learning that congestion is occurring
  - Adapting to alleviate the congestion
- TCP congestion control
  - Additive-increase, multiplicative-decrease
  - Slow start and slow-start restart
- Related TCP mechanisms
  - Nagle's algorithm and delayed acknowledgments
- Active Queue Management (AQM)
  - Random Early Detection (RED)
  - Explicit Congestion Notification (ECN)

Resource Allocation vs. Congestion Control

- Resource allocation
  - How nodes meet competing demands for resources
  - E.g., link bandwidth and buffer space
  - When to say no, and to whom
- Congestion control
  - How nodes prevent or respond to overload conditions
  - E.g., persuade hosts to stop sending, or slow down
  - Typically has notions of fairness (i.e., sharing the pain)

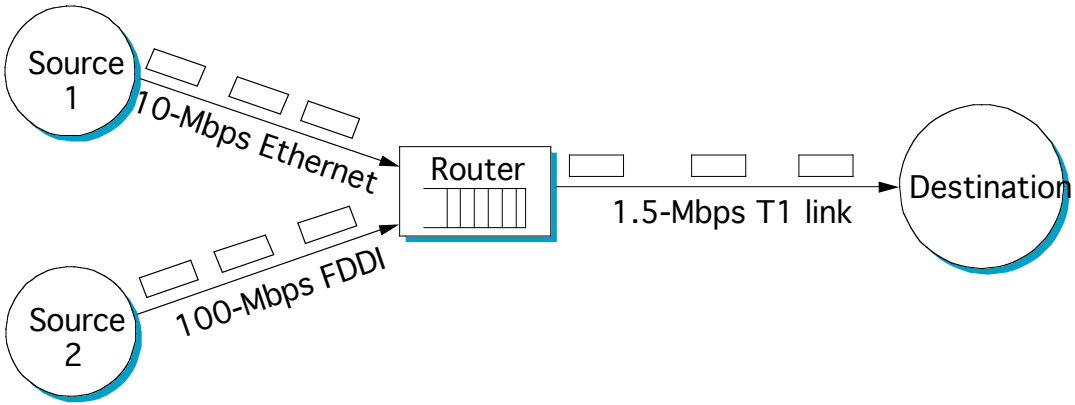
Flow Control vs. Congestion Control

- Flow control
  - Keeping *one fast sender* from overwhelming *a slow receiver*
- Congestion control
  - Keep a *set of senders* from overloading the *network*

- Different concepts, but similar mechanisms
  - TCP flow control: receiver window
  - TCP congestion control: congestion window
  - TCP window: min{congestion window, receiver window}

Issues

- Two sides of the same coin
  - pre-allocate resources so at to avoid congestion
  - control congestion if (and when) it occurs

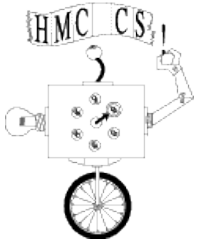


- Two points of implementation
  - hosts at the edges of the network (transport protocol)
  - routers inside the network (queuing discipline)

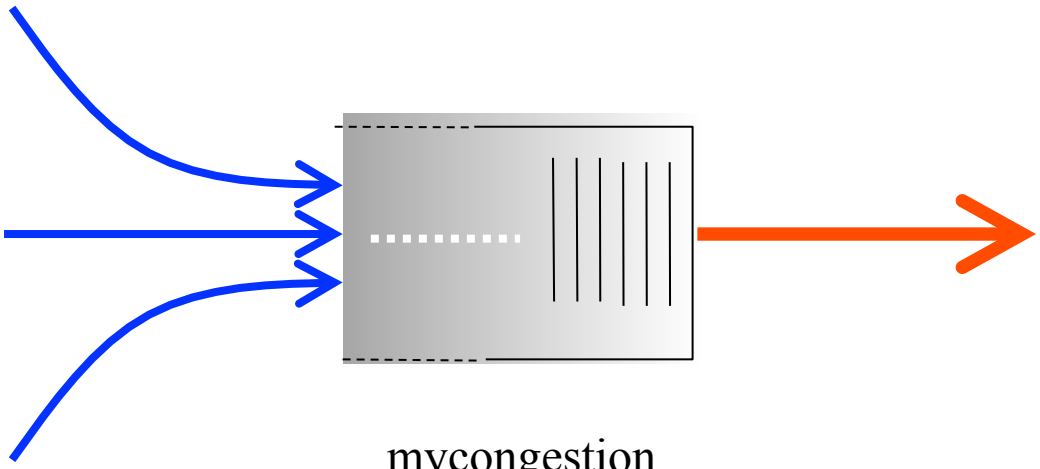
Three Key Features of Internet Service Model

- Packet switching
  - A given source may have enough capacity to send data
  - … and yet the packets may encounter an overloaded link
- Connectionless flows
  - No notions of connections inside the network
  - … and no advance reservation of network resources
  - Still, you can view related packets as a group ("flow")
  - … e.g., the packets in the same TCP transfer
- Best-effort service
  - No guarantees for packet delivery or delay
  - No preferential treatment for certain packets

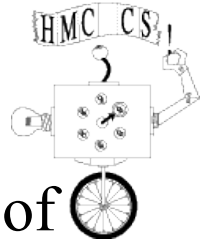
Congestion is Unavoidable



- Two packets arrive at the same time
  - The node can only transmit one
  - … and either buffer or drop the other
- If many packets arrive in a short period of time
  - The node cannot keep up with the arriving traffic
  - … and the buffer may eventually overflow

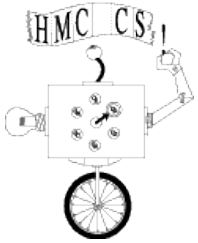


Congestion Collapse



- Definition: Increase in network load results in a decrease of useful work done
- Many possible causes
  - Spurious retransmissions of packets still in flight
    - Classical congestion collapse
    - Solution: better timers and TCP congestion control
  - Undelivered packets
    - Packets consume resources and are dropped elsewhere in network
    - Solution: congestion control for ALL traffic

What Do We Want, Really?

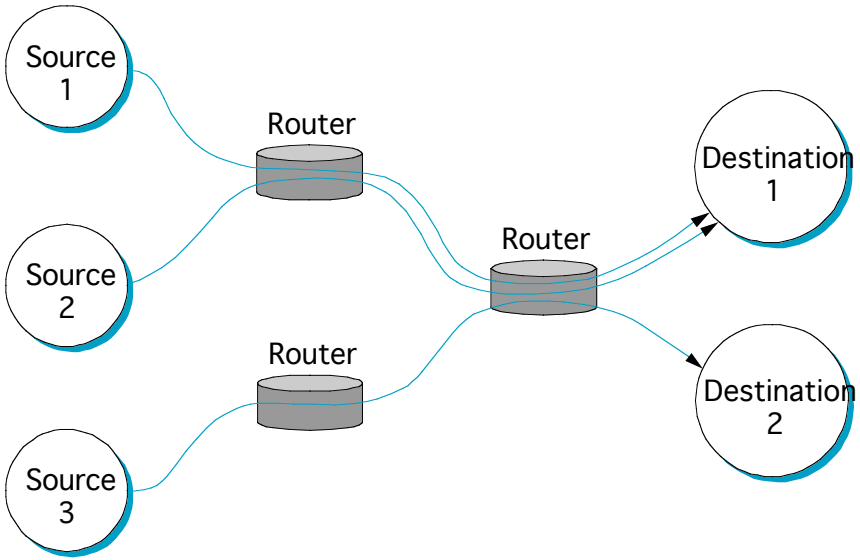


- High throughput
  - Throughput: measured performance of a system
  - E.g., number of bits/second of data that get through
- Low delay
  - Delay: time required to deliver a packet or message
  - E.g., number of msec to deliver a packet

- These two metrics are sometimes at odds
  - E.g., suppose you drive a link as hard as possible
  - … then, throughput will be high, but delay will be, too because of buffering along the way

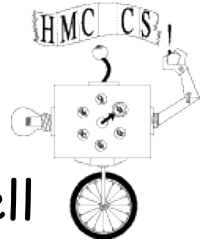
Framework

- Connectionless flows
  - sequence of packets sent between source/destination pair
  - maintain *soft state* at the routers

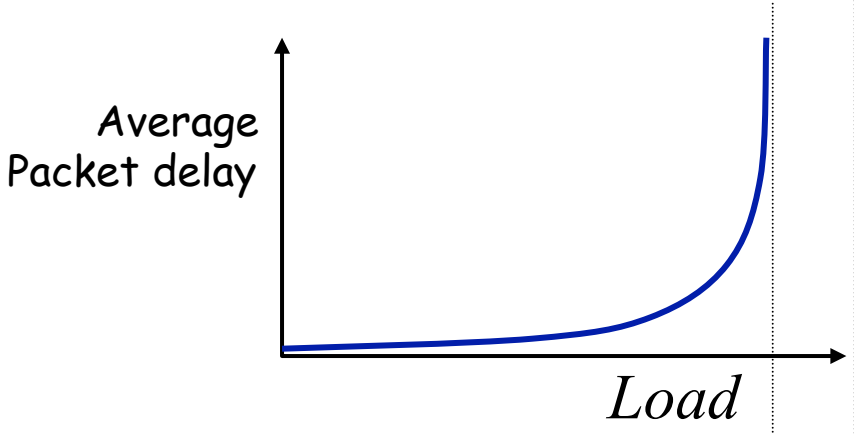


- Taxonomy
  - router-centric versus host-centric
  - reservation-based versus feedback-based
  - window-based versus rate-based

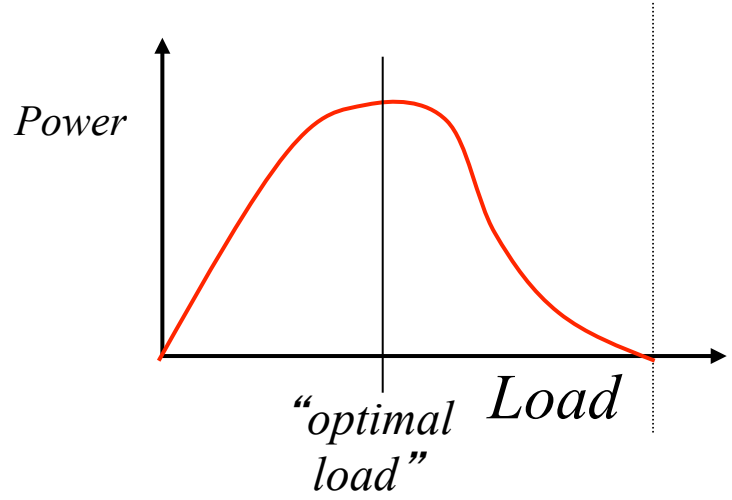
Load, Delay, and Power



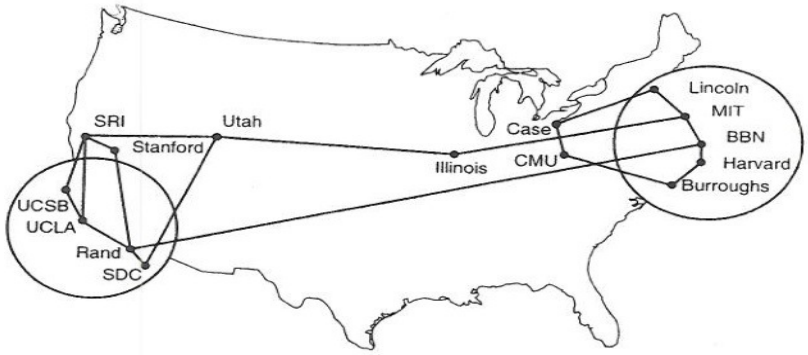
Typical behavior of queuing systems with random arrivals:

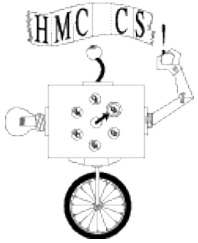


A simple metric of how well the network is performing:
Power = throughput / delay



Goal: maximize power

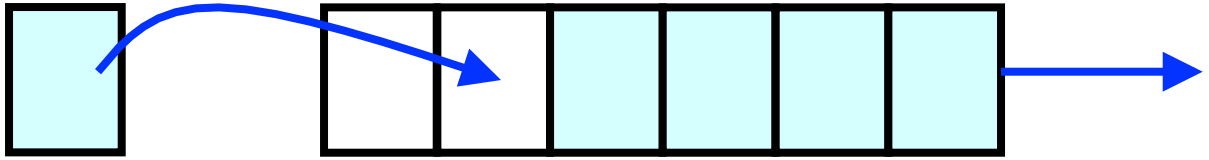




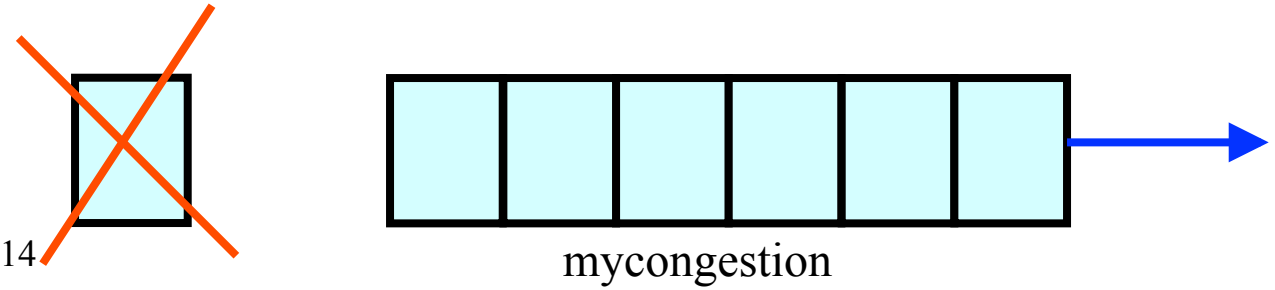
Queueing & Fairness

Simple Resource Allocation

- Simplest approach: FIFO queue and drop-tail
- Link bandwidth: first-in first-out queue
  - Packets transmitted in the order they arrive

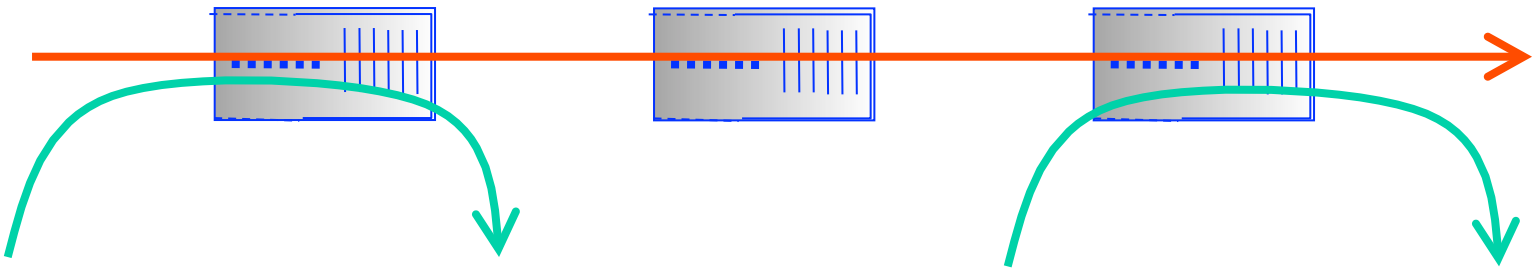


- Buffer space: drop-tail queuing
  - If the queue is full, drop the incoming packet



Fairness

- Effective utilization is not the only goal
  - We also want to be *fair* to the various flows
  - … but what the heck does *that* mean?
- Simple definition: equal shares of the bandwidth
  - N flows that each get 1/N of the bandwidth?
  - But, what if the flows traverse different paths?

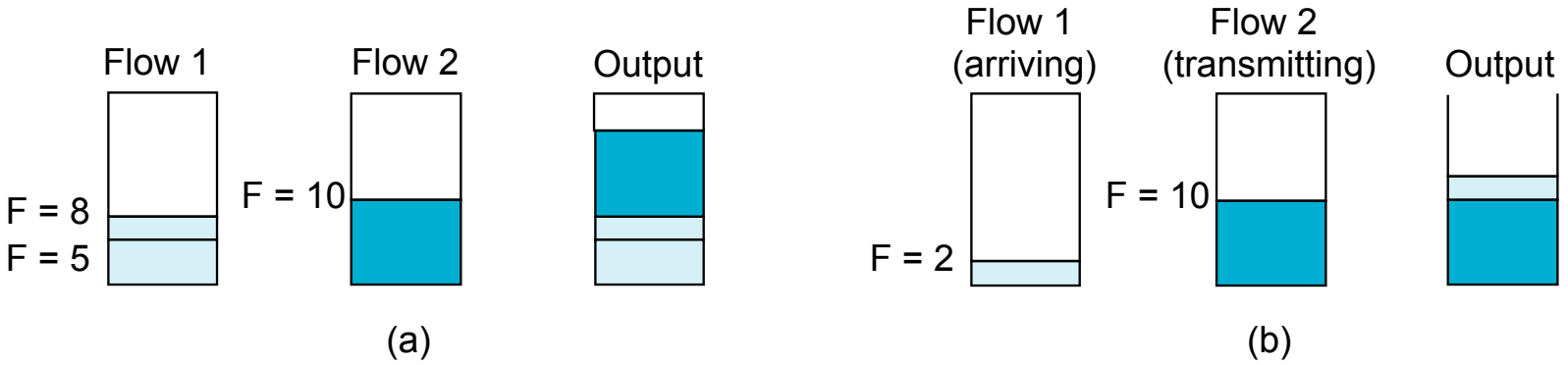


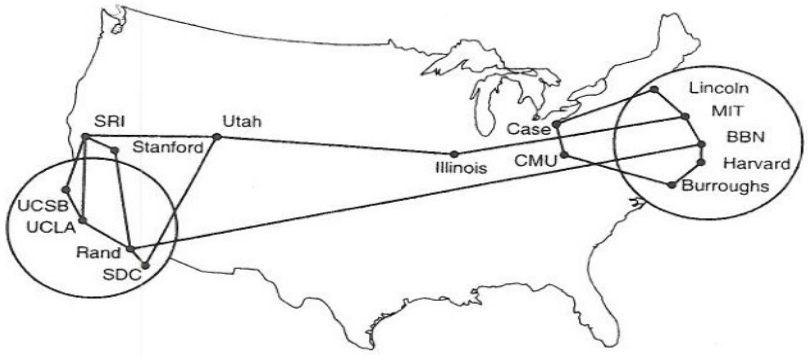
FQ Algorithm

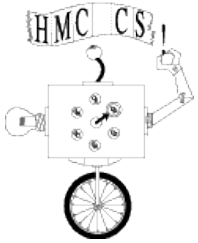
- Suppose clock ticks each time a bit is transmitted
- Let P_i denote the length of packet i
- Let S_i denote the time when start to transmit packet i
- Let F_i denote the time when finish transmitting packet i
- $F_i = S_i + P_i$
- When does router start transmitting packet i?
  - if arrives before router finished packet i - 1 from this flow, then immediately after last bit of i - 1 (F_{i-1})
  - if no current packets for this flow, then start transmitting when arrives (call this A_i)
- Thus: $F_i = MAX\,(F_{i-1}, A_i) + P_i$

FQ Algorithm (cont)

- For multiple flows – The Problem
  - calculate F_i for each packet that arrives on each flow
  - treat all F_i's as timestamps
  - next packet to transmit is one with lowest timestamp
- Not perfect: can't preempt current packet
- Example







Approaches to TCP Congestion Control

TCP Congestion Control

- Idea
  - assumes best-effort network (FIFO or FQ routers) each source determines network capacity for itself
  - uses implicit feedback
  - ACKs pace transmission (*self-clocking*)
- Challenge
  - determining the available capacity in the first place
  - adjusting to changes in the available capacity
  - When do we care about capacity?

Idea of TCP Congestion Control

- Each source determines the available capacity
  - … so it knows how many packets to have in transit
- Congestion window
  - Maximum # of unacknowledged bytes to have in transit
  - The congestion-control equivalent of receiver window
  - MaxWindow = min{congestion window, receiver window}
  - Send at the rate of the slowest component
- Adapting the congestion window
  - Decrease size upon losing a packet: backing off
  - Increase upon success: optimistically exploring limits

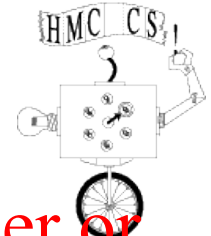
Additive Increase/Multiplicative Decrease

- Objective: adjust to changes in the available capacity
- New state variable per connection: **`CongestionWindow`**
  - limits how much data source has in transit

```
MaxWin = MIN(CongestionWindow, AdvertisedWindow)
EffWin = MaxWin - (LastByteSent - LastByteAcked)
```

- Idea:
  - increase **`CongestionWindow`** when congestion goes down
  - decrease **`CongestionWindow`** when congestion goes up

AIMD (cont)

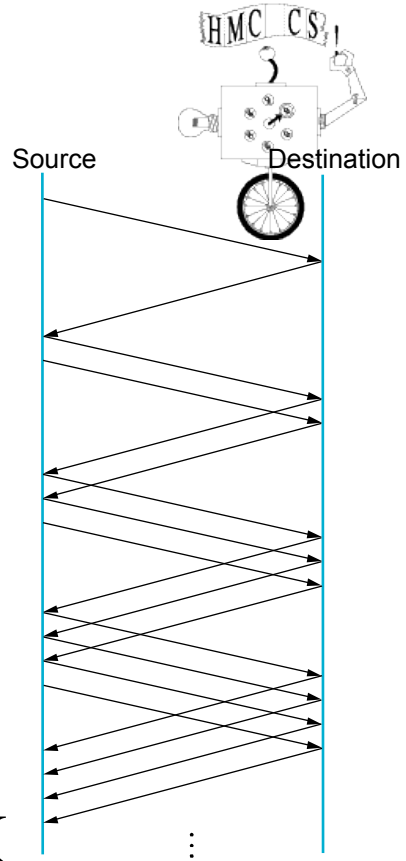


- Question: how does the source determine whether or not the network is congested?

- Answer: a timeout occurs
  - timeout signals that a packet was lost
  - packets are seldom lost due to transmission error (true?)
  - lost packet implies congestion (Does Wireless change this?)

AIMD (cont)

- Algorithm
  - increment **`CongestionWindow`** by one packet per RTT (*linear increase*)
  - divide **`CongestionWindow`** by two whenever a timeout occurs (*multiplicative decrease*)



- In practice: increment a little for each ACK

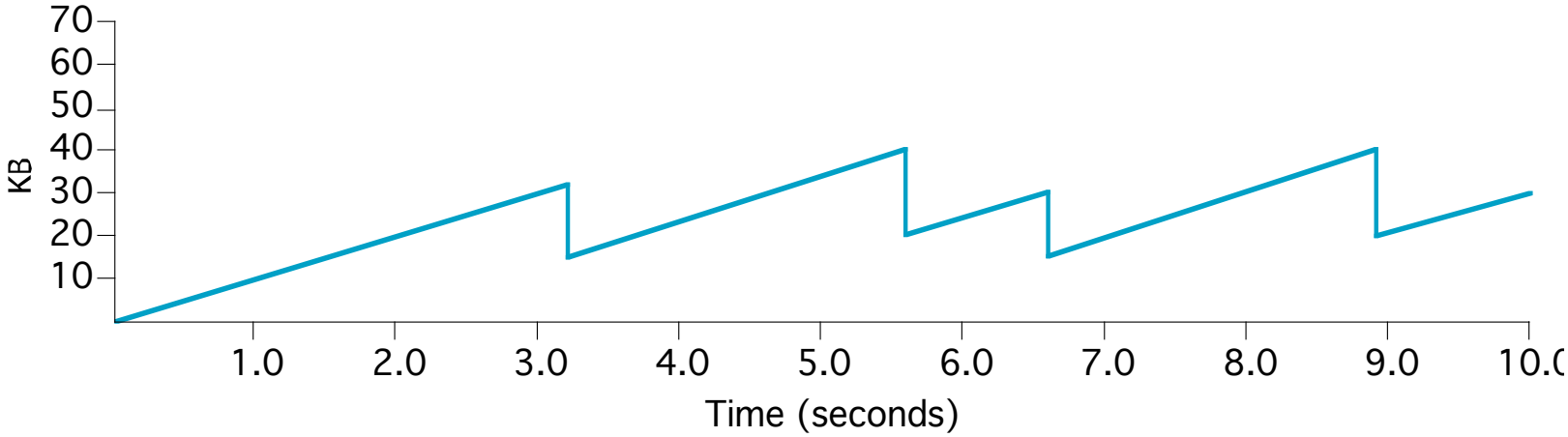
```
Increment = (MSS * MSS)/CongestionWindow
CongestionWindow += Increment
```

Additive Increase/Multiplicative Decrease

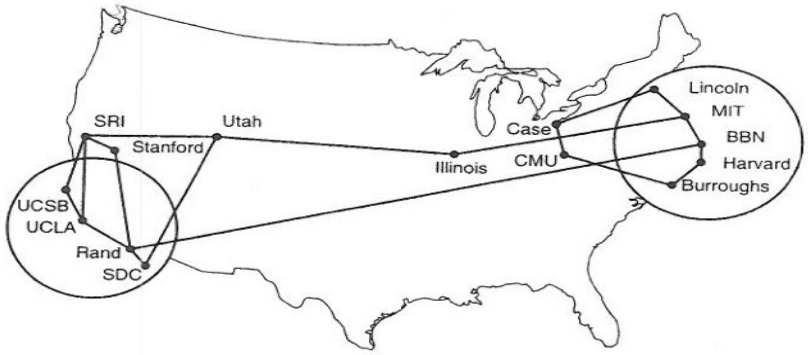
- How much to increase and decrease?
  - Increase linearly, decrease multiplicatively
  - A necessary condition for stability of TCP
  - Consequences of over-sized window are much worse than having an under-sized window
    - Over-sized window: packets dropped and retransmitted
    - Under-sized window: somewhat lower throughput
- Multiplicative decrease
  - On loss of packet, divide congestion window in half
- Additive increase
  - On success for last window of data, increase linearly

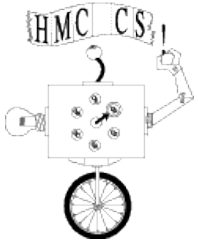
AIMD (cont)

- Trace: sawtooth behavior
- Start with small Congestion Window to avoid overloading nextwork



Note, the ½ of the Window


SRI
Utah
Stanford
Illinois
Case
CMU
Lincoln
MIT
BBN
Harvard
Burroughs
UCSB
UCLA
Rand
SDC


HMC CS

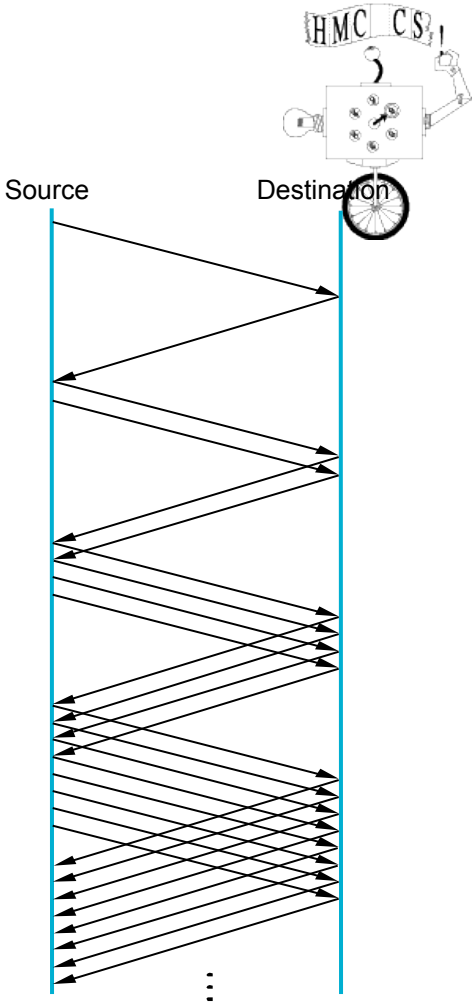
Slow Start

"Slow Start" Phase

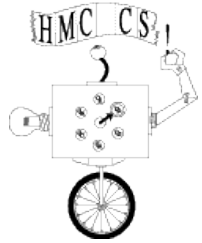
- Start with a small congestion window
  - Initially, CWND is 1 MSS
  - So, initial sending rate is MSS/RTT
- That could be pretty wasteful
  - Might be much less than the actual bandwidth
  - Linear increase takes a long time to accelerate
- Slow-start phase (really "fast start")
  - Sender starts at a slow rate (hence the name)
  - … but increases the rate exponentially
  - … until the first loss event

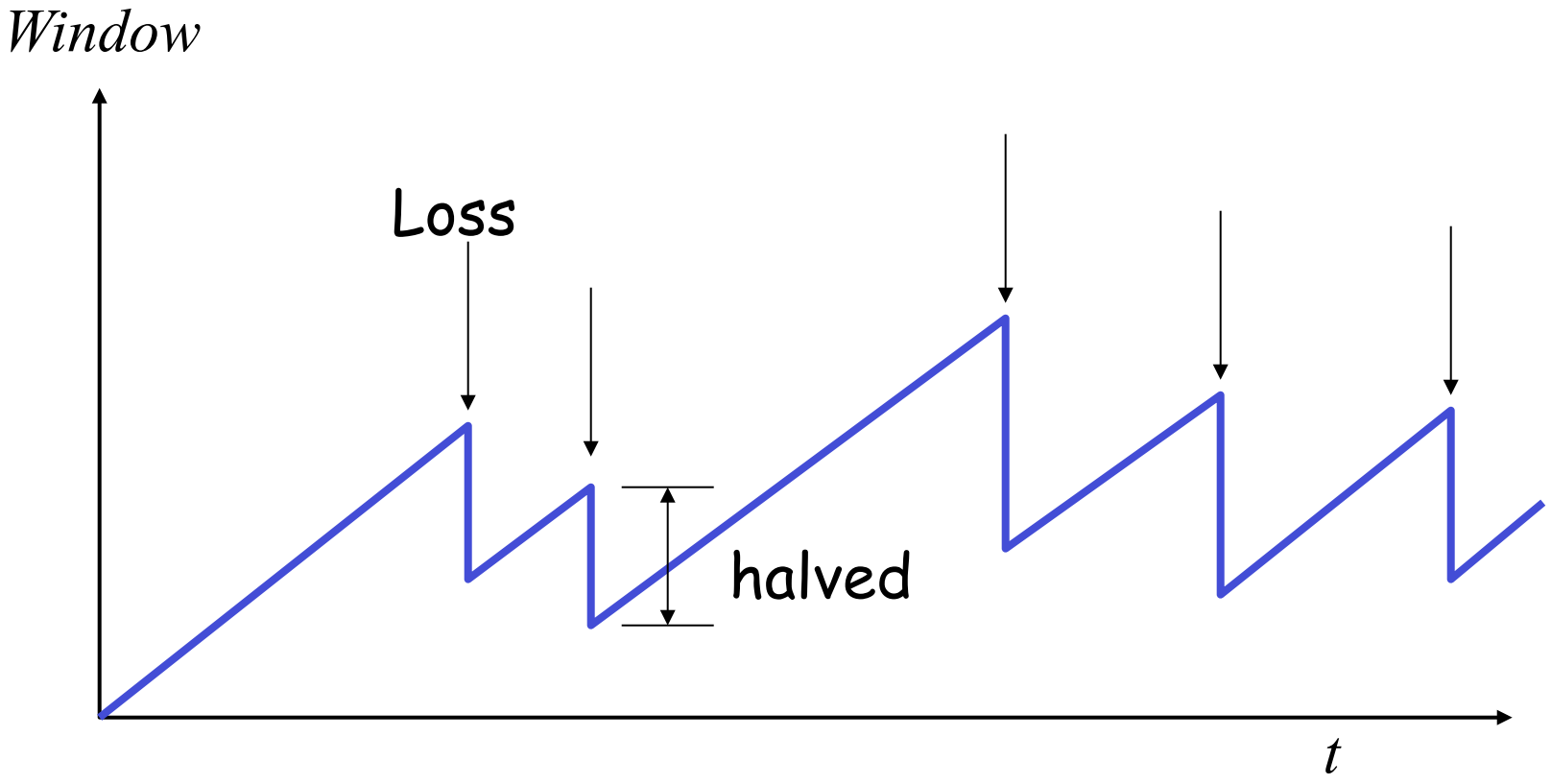
Slow Start

- Objective: determine the available capacity.
- Idea:
  - begin with **`CongestionWindow`** = 1 packet
  - double **`CongestionWindow`** each RTT (increment by 1 packet for each ACK)



Leads to the TCP "Sawtooth"

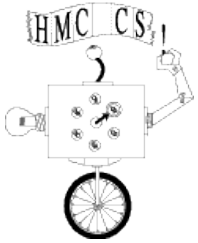




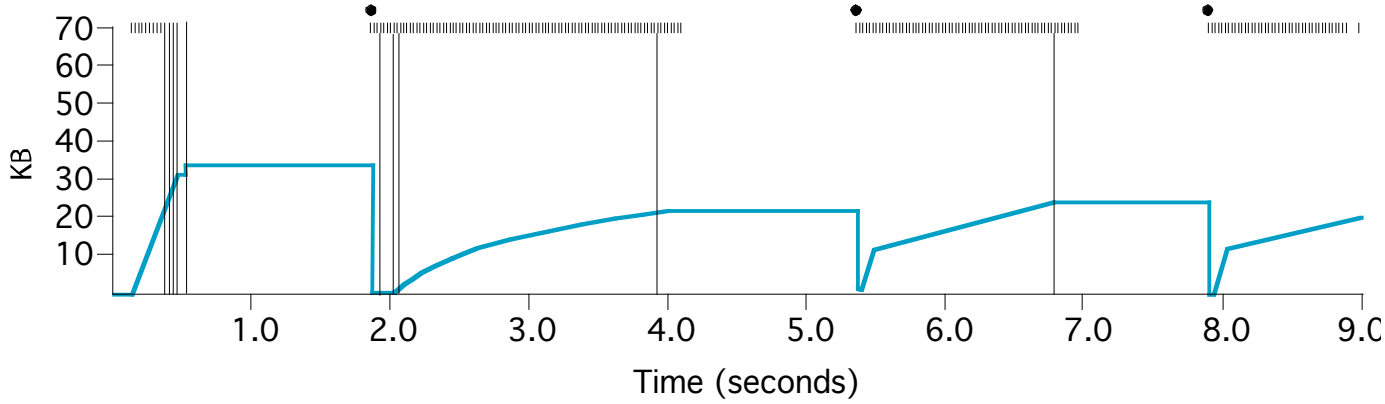
Practical Details

- Congestion window
  - Represented in bytes, not in packets (Why?)
  - Packets have MSS (Maximum Segment Size) bytes
- Increasing the congestion window
  - Increase by MSS on success for last window of data
  - In practice, increase a fraction of MSS per received ACK
    - # packets per window: CW / MSS
    - Increment per ACK: MSS * (MSS / CW)
- Decreasing the congestion window
  - Never drop congestion window below 1 MSS

Slow Start (cont)

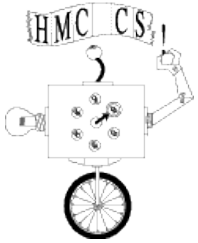


- Exponential growth, but slower than all at once
- Used…
  - when first starting connection
  - when connection goes dead waiting for timeout
- Trace



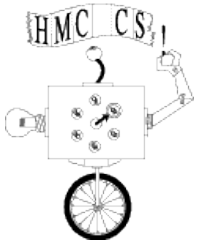
- Problem: lose up to half a **`CongestionWindow`**'s worth of data

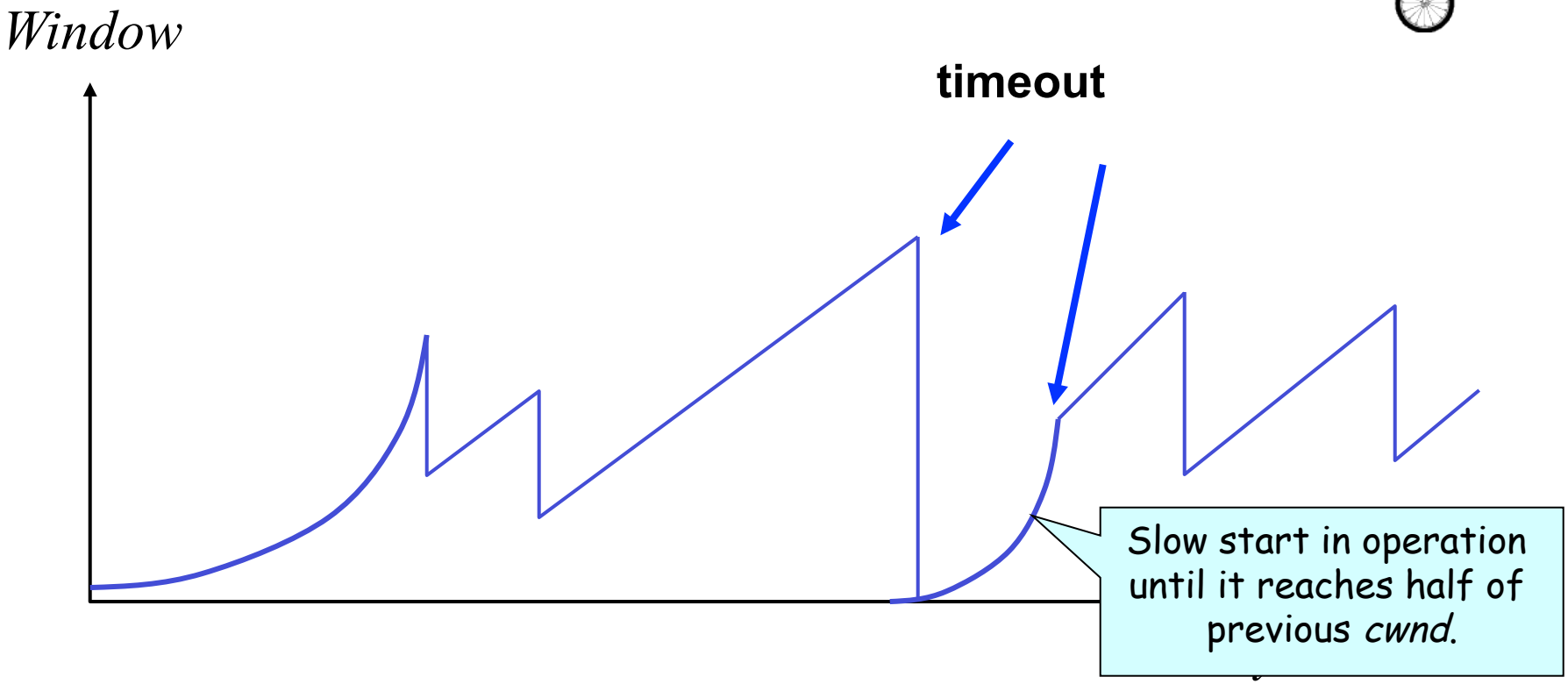
Two Kinds of Loss in TCP



- Triple duplicate ACK
  - Packet n is lost, but packets n+1, n+2, etc. arrive
  - Receiver sends duplicate acknowledgments
  - … and the sender retransmits packet n quickly
  - Do a multiplicative decrease and keep going
- Timeout
  - Packet n is lost and detected via a timeout
  - E.g., because all packets in flight were lost
  - After the timeout, blasting away for the entire CW
  - … would trigger a very large burst in traffic
  - So, better to start over with a low CW

Repeating Slow Start After Timeout



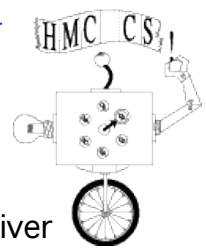


Slow-start restart: Go back to CW of 1, but take advantage of knowing the previous value of CW.

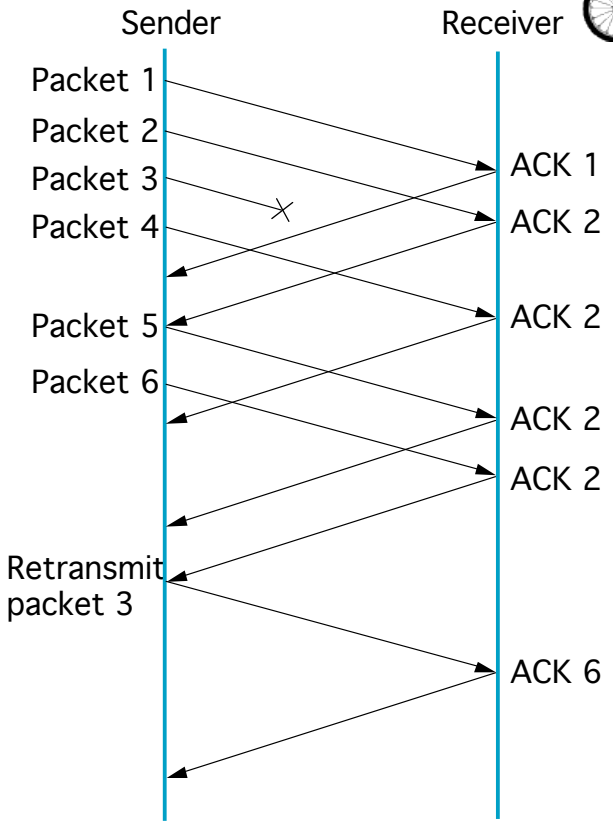
Repeating Slow Start After Idle Period

- Suppose a TCP connection goes idle for a while
  - E.g., Telnet session where you don't type for an hour
- Eventually, the network conditions change
  - Maybe many more flows are traversing the link
  - E.g., maybe everybody has come back from lunch!
- Dangerous to start transmitting at the old rate
  - Previously-idle TCP sender might blast the network
  - … causing excessive congestion and packet loss
- So, some TCP implementations repeat slow start
  - Slow-start restart after an idle period

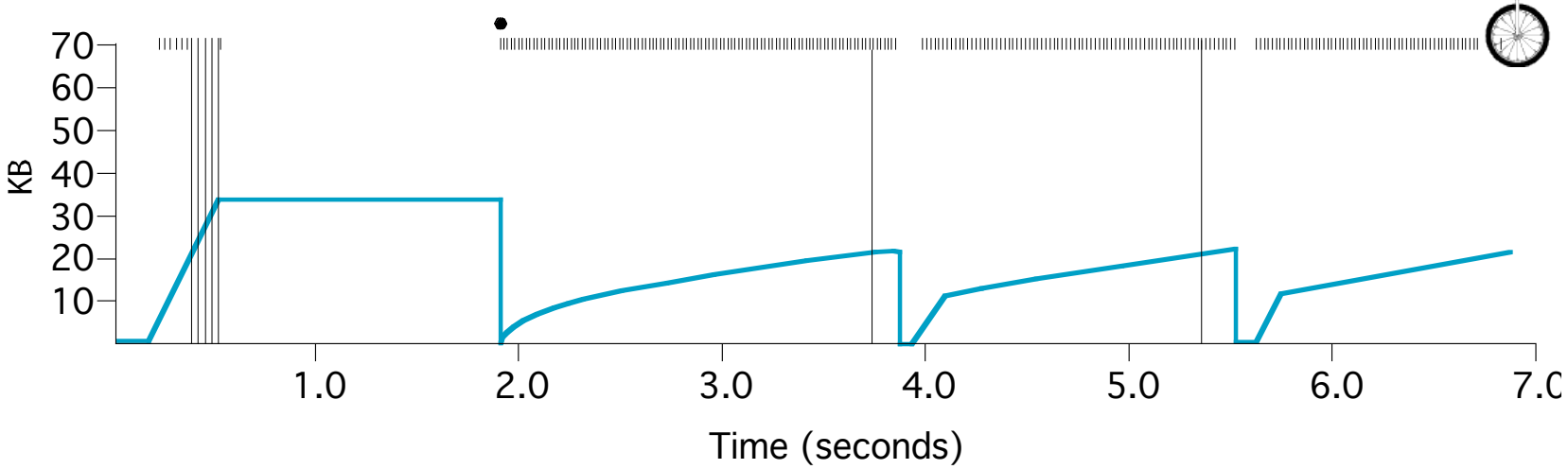
Fast Retransmit and Fast Recovery



- Problem: coarse-grain TCP timeouts lead to idle periods
- Fast retransmit: use duplicate ACKs to trigger retransmission



Results

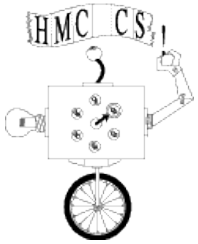


- Fast recovery
  - skip the slow start phase
  - go directly to half the last successful **`CongestionWindow (ssthresh)`**

Other TCP Mechanisms

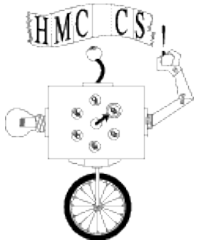
Nagle's Algorithm and Delayed ACK

Motivation for Nagle's Algorithm

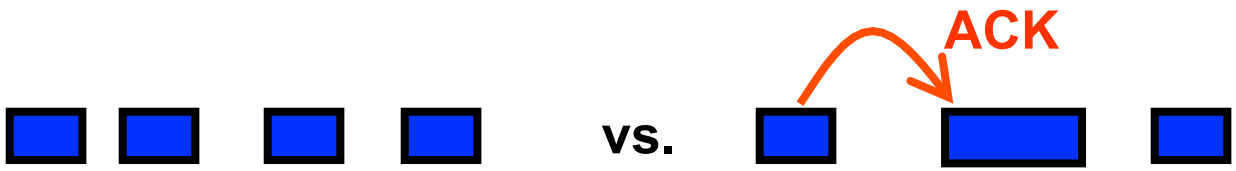


- Interactive applications
  - Telnet and rlogin
  - Generate many small packets (e.g., keystrokes)
- Small packets are wasteful
  - Mostly header (e.g., 40 bytes of header, 1 of data)
- Appealing to reduce the number of packets
  - Could force every packet to have some minimum size
  - … but, what if the person doesn't type more characters?
- Need to balance competing trade-offs
  - Send larger packets
  - … but don't introduce much delay by waiting

Nagle's Algorithm



- Wait if the amount of data is small
  - Smaller than Maximum Segment Size (MSS)
- And some other packet is already in flight
  - I.e., still awaiting the ACKs for previous packets
- That is, send at most one small packet per RTT
  - … by waiting until all outstanding ACKs have arrived

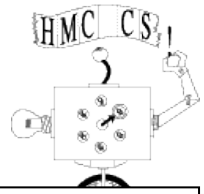


- Influence on performance
  - Interactive applications: enables batching of bytes
  - Bulk transfer: transmits in MSS-sized packets anyway

Motivation for Delayed ACK

- TCP traffic is often bidirectional
  - Data traveling in both directions
  - ACKs traveling in both directions
- ACK packets have high overhead
  - 40 bytes for the IP header and TCP header
  - … and zero data traffic
- Piggybacking is appealing
  - Host B can send an ACK to host A
  - … as part of a data packet from B to A

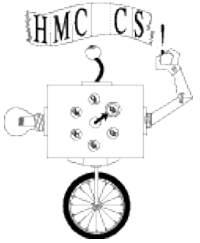
TCP Header Allows Piggybacking

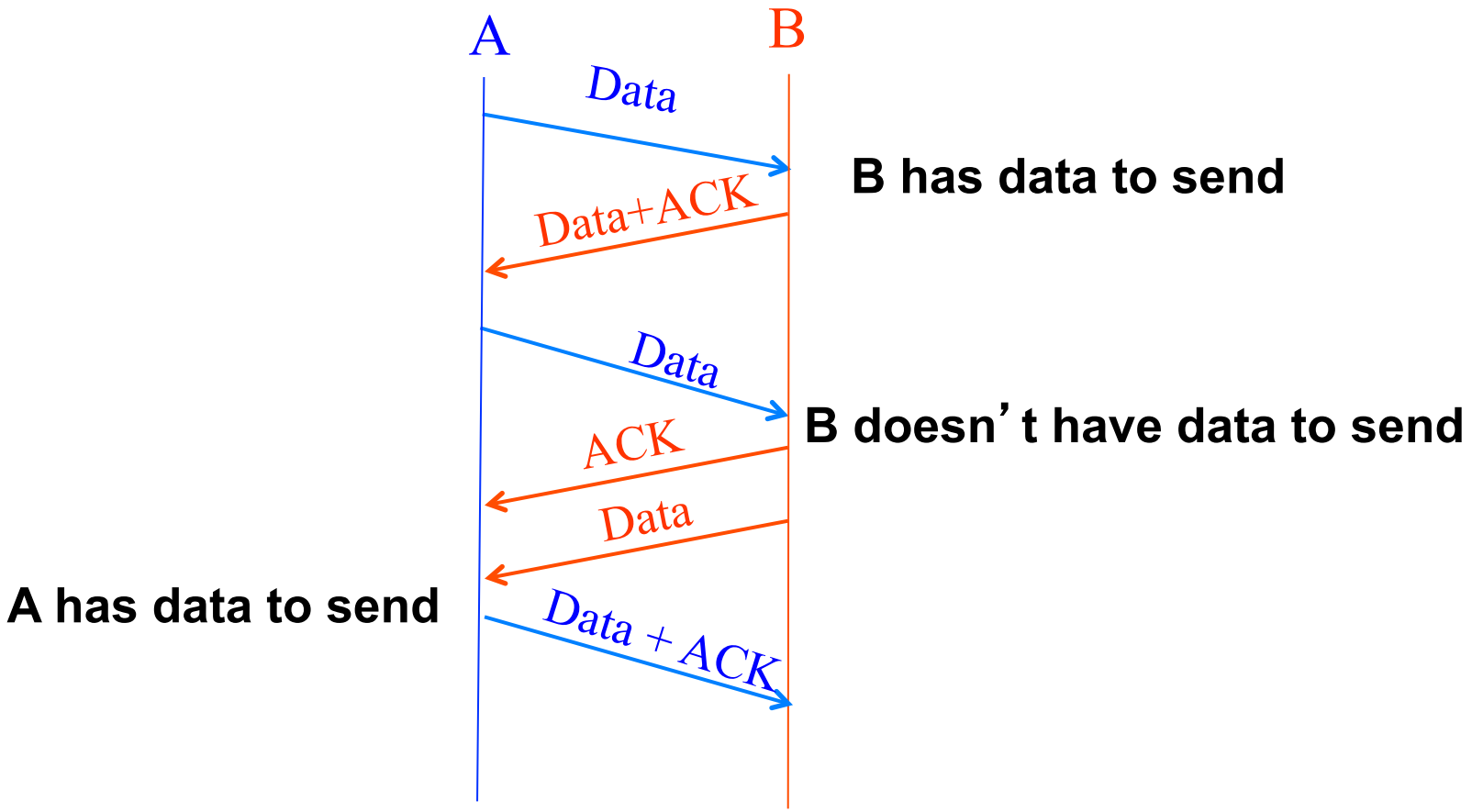


Flags: SYN
FIN
RST
PSH
URG
ACK

Source port			Destination port
Sequence number			
Acknowledgment			
HdrLen	0	Flags	Advertised window
Checksum			Urgent pointer
Options (variable)			
Data			

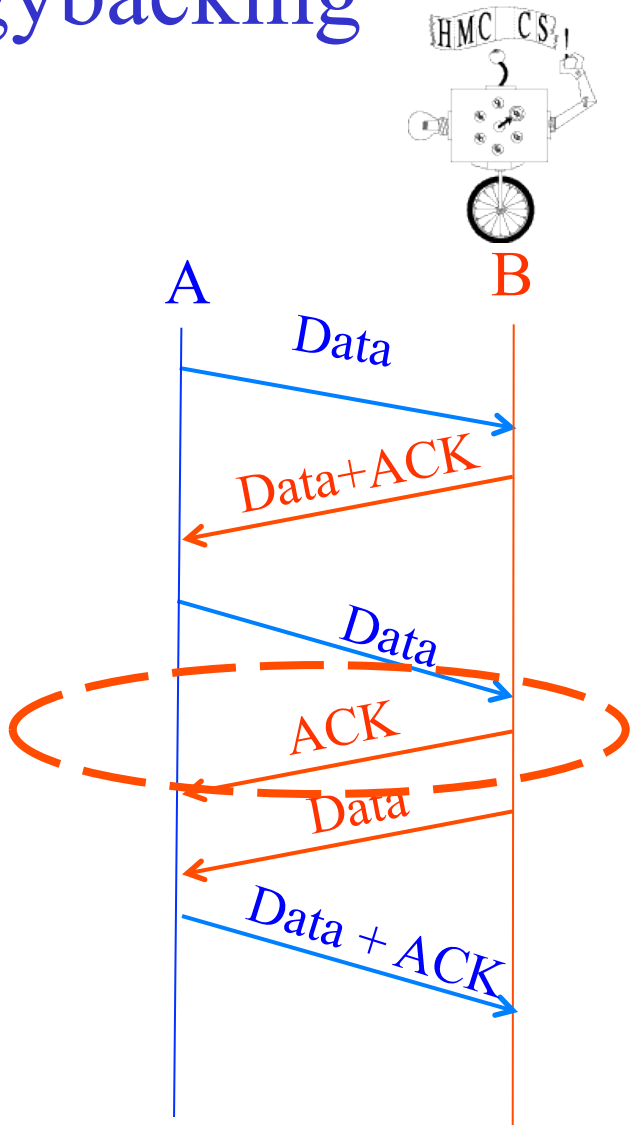
Example of Piggybacking





Increasing Likelihood of Piggybacking

- Increase piggybacking
  - TCP allows the receiver to *wait* to send the ACK
  - … in the hope that the host will have data to send
- Example: rlogin or telnet
  - Host A types characters at a UNIX prompt
  - Host B receives the character and executes a command
  - … and then data are generated
  - Would be nice if B could send the ACK with the new data



Delayed ACK

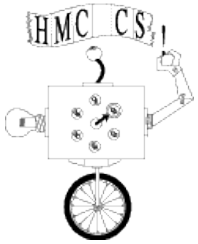
- Delay sending an ACK
  - Upon receiving a packet, the host B sets a timer
    - Typically, 200 msec or 500 msec
  - If B's application generates data, go ahead and send
    - And piggyback the ACK bit
  - If the timer expires, send a (non-piggybacked) ACK
- Limiting the wait
  - Timer of 200 msec or 500 msec
  - ACK every other full-sized packet

Other Mechanisms

Random Early Detection (RED)

TCP Vegas

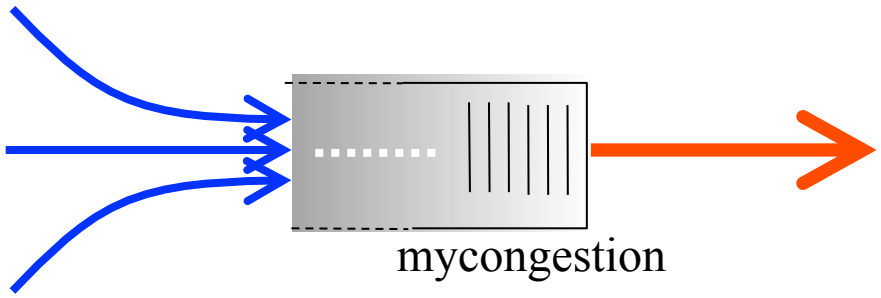
Congestion Avoidance



- TCP's strategy
  - control congestion once it happens
  - repeatedly increase load in an effort to find the point at which congestion occurs, and then back off
- Alternative strategy
  - predict when congestion is about to happen
  - reduce rate before packets start being discarded
  - call this congestion *avoidance*, instead of congestion *control*
- Two possibilities
  - router-centric: DECbit and RED Gateways
  - host-centric: TCP Vegas

Bursty Loss From Drop-Tail Queuing

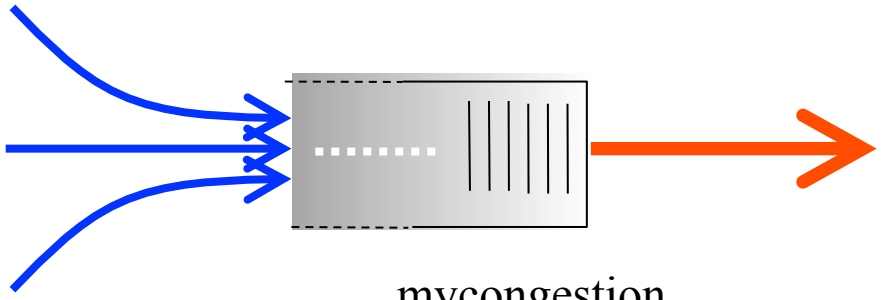
- TCP depends on packet loss
  - Packet loss is the indication of congestion
  - In fact, TCP *drives* the network into packet loss
  - … by continuing to increase the sending rate
- Drop-tail queuing leads to *bursty* loss
  - When a link becomes congested…
  - … many arriving packets encounter a full queue
  - And, as a result, many flows divide sending rate in half
  - … and, many individual flows lose multiple packets



mycongestion

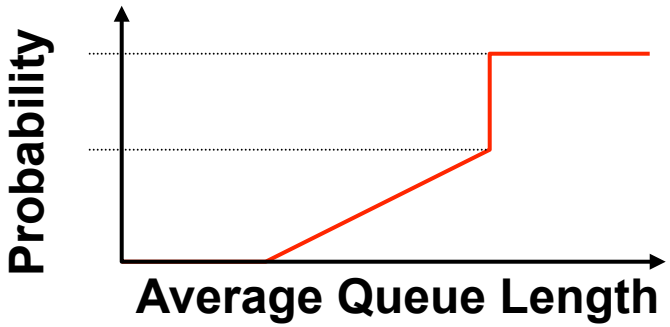
Slow Feedback from Drop Tail

- Feedback comes when buffer is completely full
  - … even though the buffer has been filling for a while
- Plus, the filling buffer is increasing RTT
  - … and the variance in the RTT
- Might be better to give early feedback
  - Get one or two flows to slow down, not all of them
  - Get these flows to slow down before it is too late



Random Early Detection (RED)

- Basic idea of RED
  - Router notices that the queue is getting backlogged
  - … and randomly drops packets to signal congestion
- Packet drop probability
  - Drop probability increases as queue length increases
  - If buffer is below some level, don't drop anything
  - … otherwise, set drop probability as function of queue



mycongestion

Properties of RED

- Drops packets before queue is full
  - In the hope of reducing the rates of some flows
- Drops packet in proportion to each flow's rate
  - High-rate flows have more packets
  - … and, hence, a higher chance of being selected
- Drops are spaced out in time
  - Which should help desynchronize the TCP senders
- Tolerant of burstiness in the traffic
  - By basing the decisions on *average* queue length
- Notification is implicit

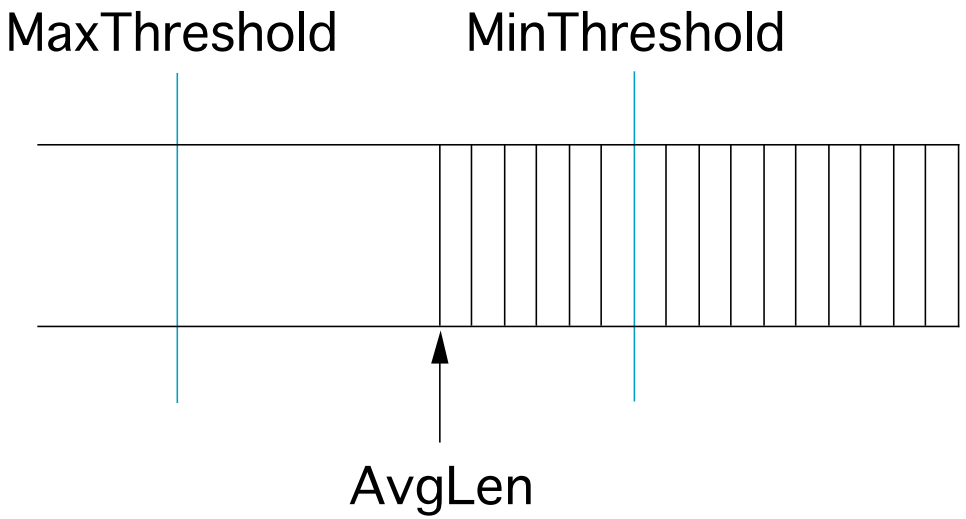
RED Details

- Compute average queue length

```
AvgLen = (1 - Weight) * AvgLen +
                Weight * SampleLen
```

0 < **`Weight`** < 1 (usually 0.002)

`SampleLen` is queue length each time a packet arrives

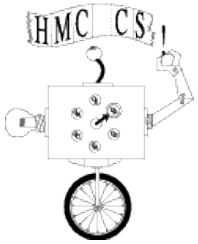


RED Details (cont)

- Two queue length thresholds

```
if AvgLen <= MinThreshold then
   enqueue the packet
if MinThreshold < AvgLen < MaxThreshold then
   calculate probability P
   drop arriving packet with probability P
if ManThreshold <= AvgLen then
   drop arriving packet
```

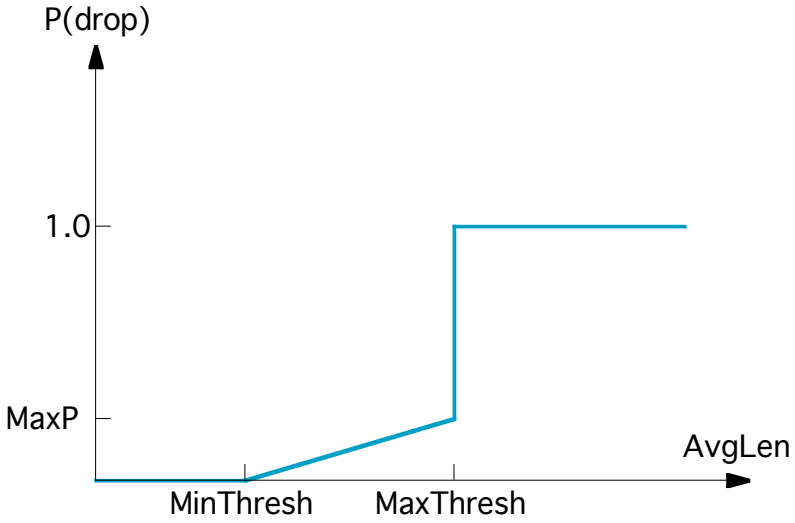
RED Details (cont)



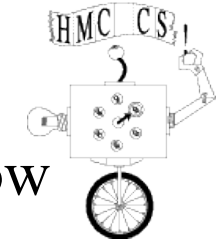
- Computing probability P

```
TempP = MaxP * (AvgLen -
  MinThreshold)/
   (MaxThreshold - MinThreshold)
P = TempP/(1 - count * TempP)
```

- Drop Probability Curve



Tuning RED



- Probability of dropping a particular flow's packet(s) is roughly proportional to the share of the bandwidth that flow is currently getting
- **`MaxP`** is typically set to 0.02, meaning that when the average queue size is halfway between the two thresholds, the gateway drops roughly one out of 50 packets.
- If traffic id bursty, then **`MinThreshold`** should be sufficiently large to allow link utilization to be maintained at an acceptably high level
- Difference between two thresholds should be larger than the typical increase in the calculated average queue length in one RTT; setting **`MaxThreshold`** to twice **`MinThreshold`** is reasonable for traffic on today's Internet
- Penalty Box for Offenders

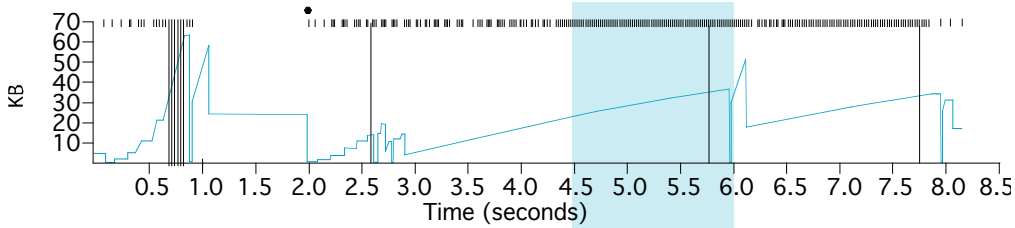
Problems With RED

- Hard to get the tunable parameters just right
  - How early to start dropping packets?
  - What slope for the increase in drop probability?
  - What time scale for averaging the queue length?
- Sometimes RED helps but sometimes not
  - If the parameters aren't set right, RED doesn't help
  - And it is hard to know how to set the parameters
- RED is implemented in practice
  - But, often not used due to the challenges of tuning right
- Many variations
  - With cute names like "Blue" and "FRED"… ☺
  - What is the change in Philosophy?

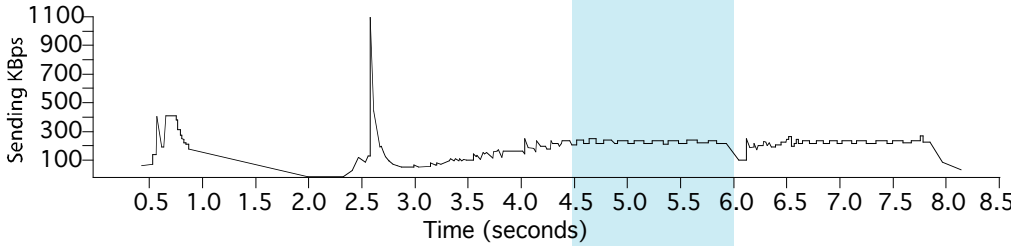
TCP Vegas

- Idea: source watches for some sign that router's queue is building up and congestion will happen too; e.g.,
  - RTT growth indicates congestion…
  - sending rate flattens

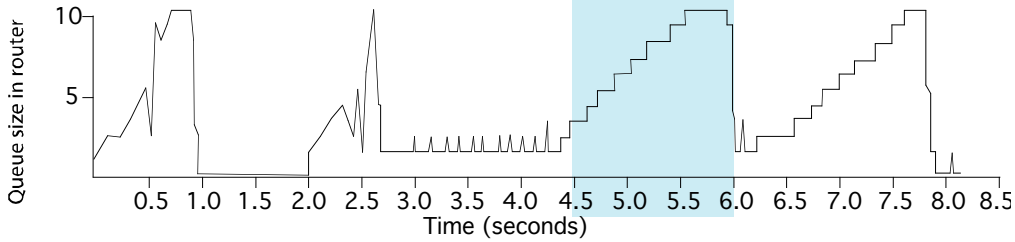
Congestion Window



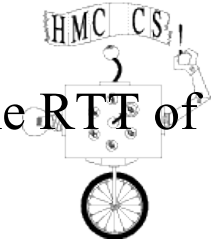
Observed Throughput
Why no increase?
No bandwidth



Router Buffer Space



Algorithm



- Let **`BaseRTT`** be the minimum of all measured RTTs (commonly the RTT of the first packet)

- If not overflowing the connection, then

```
ExpectRate = CongestionWindow/BaseRTT
```

as RTT increases, **`ExpectRate`**, descreases

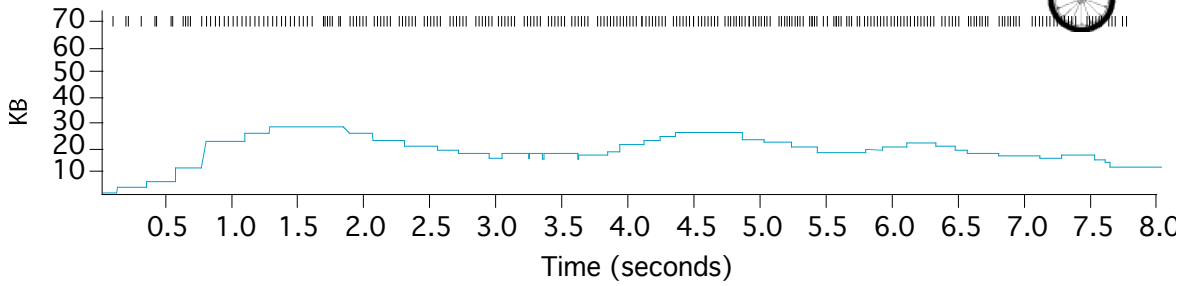
- Source calculates sending rate (**`ActualRate`**) once per RTT – More Work
- Source compares **`ActualRate`** with **`ExpectRate`**

```
Diff = ExpectedRate - ActualRate (Always Positive)
if Diff < α  -- Set to 1 buffer
    increase CongestionWindow linearly
else if Diff > β --Set to 3 buffer
    decrease CongestionWindow linearly
else
    leave CongestionWindow unchanged
```

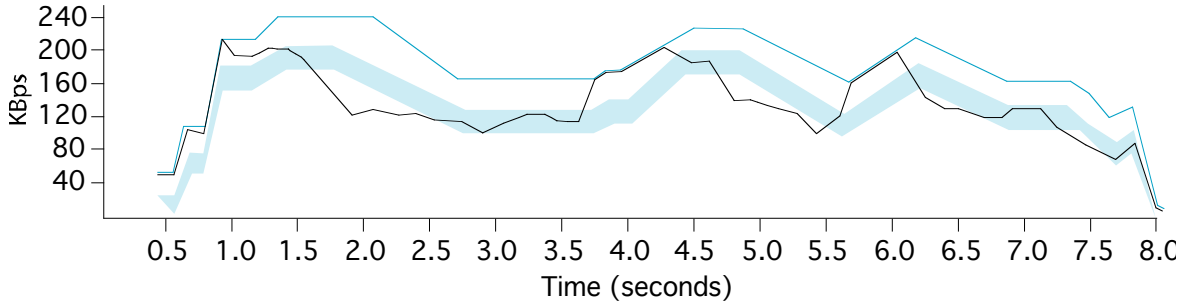
Algorithm (cont)

- Parameters
  - α = 1 packet
  - β = 3 packets

Congestion Window



Expected = color
Actual = black
Band = α, β



Even faster retransmit

- keep fine-grained timestamps for each packet
- check for timeout on first duplicate ACK

Conclusions

- Congestion is inevitable
  - Internet does not reserve resources in advance
  - TCP actively tries to push the envelope
- Congestion can be handled
  - Additive increase, multiplicative decrease
  - Slow start, and slow-start restart
- Active Queue Management can help
  - Random Early Detection (RED)
  - Explicit Congestion Notification (ECN)
- Hosts can manage