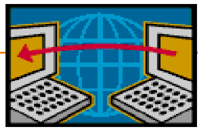


Congestion Control

EE122 Fall 2012

Scott Shenker

http://inst.eecs.berkeley.edu/~ee122/

Materials with thanks to Jennifer Rexford, Ion Stoica, Vern Paxson and other colleagues at Princeton and UC Berkeley

Announcements

- **No office hours on Thursday!**

TCP Refresher

Same slides, but crucial for rest of lecture

TCP Header

Starting sequence number (byte offset) of data carried in this segment

This is the number of the first byte of data in packet!

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

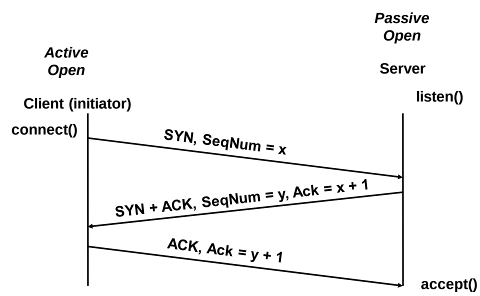
TCP Header

Acknowledgment gives seq # **just beyond** highest seq. received **in order**.

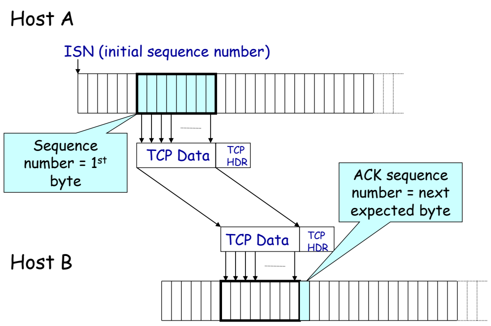
"What's Next"

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

3-Way Handshaking



Sequence Numbers



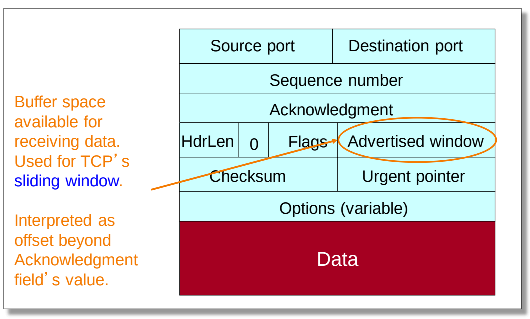
7

Data and ACK in same packet

- The sequence number refers to data in packet
  - Packet from A carrying data to B
- The ACK refers to received data in other direction
  - A acking data that it received from B

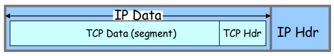
8

TCP Header



9

TCP Segment



- IP packet
  - No bigger than Maximum Transmission Unit (MTU)
  - E.g., up to 1,500 bytes on an Ethernet
- TCP packet
  - IP packet with a TCP header and data inside
  - TCP header ≥ 20 bytes long
- TCP **segment**
  - No more than Maximum Segment Size (MSS) bytes
  - E.g., up to 1460 consecutive bytes from the stream
  - MSS = MTU – (IP header) – (TCP header)

10

Congestion Control Overview

Everything in this lecture is oversimplified.

Lots of details omitted.

But the basic points remain valid….

11

Flow Control vs Congestion Control

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

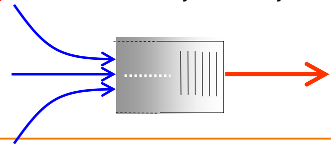
12

Huge Literature on Problem

- In mid-80s Jacobson “saved” the Internet with CC
- One of very few net topics where theory helps; many frustrated mathematicians in networking
- Less of a research focus now in the wide area
  - But still actively researched in datacenter networks
  - And commercial activity in wide area (e.g., Google)
- …but still far from academically settled
  - E.g. battle over “fairness” with Bob Briscoe…

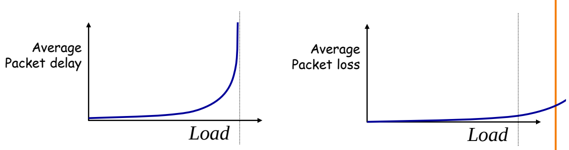
Congestion is Natural

- **Because Internet traffic is bursty!**
- If two packets arrive at the same time
  - The node can only transmit one
  - … and either buffers or drops the other
- If many packets arrive in a short period of time
  - The node cannot keep up with the arriving traffic
  - … delays, and the buffer may eventually overflow



Load and Delay

Typical queuing system with bursty arrivals



Must balance utilization versus delay and loss

Who Takes Care of Congestion?

- Network?
- End hosts?
- Both?

Answer

- End **hosts** adjust sending rate
- Based on feedback from **network**
- Hosts probe network to test level of congestion
  - Speed up when no congestion
  - Slow down when congestion

Drawbacks

- Suboptimal (always above or below optimal point)
- Relies on end system cooperation
- Messy dynamics
  - All end systems adjusting at the same time
  - Large, complicated dynamical system
  - Miraculous it works at all!

Basics of TCP Congestion Control

- Congestion window (CWND)
  - Maximum # of unacknowledged bytes to have in flight
  - Congestion-control equivalent of receiver window
  - MaxWindow = min{congestion window, receiver window}
    - Typically assume receiver window much bigger than cwnd
- Adapting the congestion window
  - Increase upon lack of congestion: optimistic exploration
  - Decrease upon detecting congestion

19

Detecting Congestion

- Network could tell source (ICMP Source Quench)
  - Risky, because during times of overload the signal itself could be dropped (and add to congestion)!
- Packet delays go up (knee of load-delay curve)
  - Tricky: noisy signal (delay often varies considerably)
- **Packet loss**
  - **Fail-safe** signal that TCP already has to detect
  - Complication: non-congestive loss (checksum errors)

20

Not All Losses the Same

- Duplicate ACKs: isolated loss
  - Still getting ACKs
- Timeout: possible disaster
  - Not enough dupacks
  - Must have suffered several losses

21

How to Adjust CWND?

- Consequences of over-sized window much worse than having an under-sized window
  - Over-sized window: packets dropped and retransmitted
  - Under-sized window: somewhat lower throughput
- Approach:
  - Gentle increase when uncongested (exploration)
  - Rapid decrease when congested

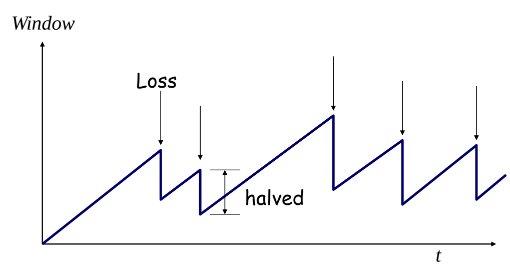
22

AIMD

- Additive increase
  - On success of last window of data, increase by one MSS
- Multiplicative decrease
  - On loss of packet, divide congestion window in **half**

23

Leads to the TCP "Sawtooth"



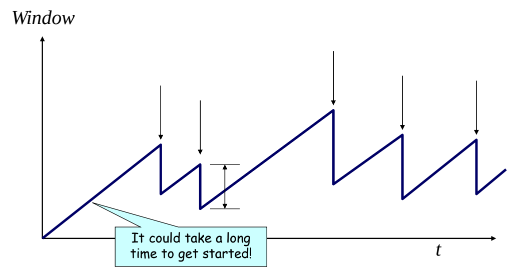
24

Slow-Start

In what follows refer to cwnd in units of MSS

AIMD Starts Too Slowly!

Need to start with a small CWND to avoid overloading the network.



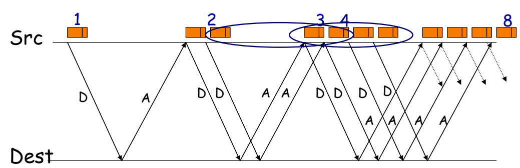
"Slow Start" Phase

- 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 (actually "fast start")
  - Sender starts at a slow rate (hence the name)
  - … but increases **exponentially** until first loss

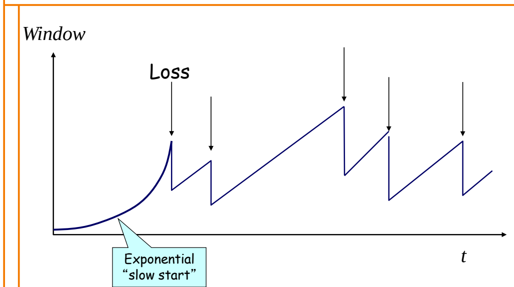
Slow Start in Action

Double CWND per round-trip time

Simple implementation:
on each ack, CWND += MSS



Slow Start and the TCP Sawtooth



Why is it called slow-start? Because TCP originally had no congestion control mechanism. The source would just start by sending a whole window's worth of data.

This has been incredibly successful

- Leads to the theoretical puzzle:

If TCP congestion control is the answer, then what was the question?

- Not about optimizing, but about robustness
  - Hard to capture…

Congestion Control Details

Increasing CWND

- **Increase by MSS for every successful window**

- Increase a fraction of MSS per received ACK
- # packets (thus ACKs) per window: CWND / MSS
- Increment per ACK:

  CWND += MSS / (CWND / MSS)

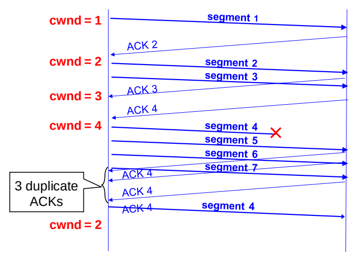
- Termed: Congestion Avoidance
  - Very gentle increase

Fast Retransmission

- Sender sees 3 dupACKs

- Multiplicative decrease: CWND halved

CWND with Fast Retransmit



Loss Detected by Timeout

- Sender starts a timer that runs for RTO seconds
- **Restart timer whenever ack for new data arrives**

- If timer expires:
  - Set **SSTHRESH** ← CWND **/ 2** (“Slow-Start Threshold”)
  - Set **CWND** ← MSS
  - Retransmit **first** lost packet
  - Execute Slow Start <u>until</u> **CWND** > **SSTHRESH**
  - After which switch to Additive Increase

Summary of Decrease

- Cut CWND <u>half</u> on loss detected by dupacks
  - “**fast retransmit**”

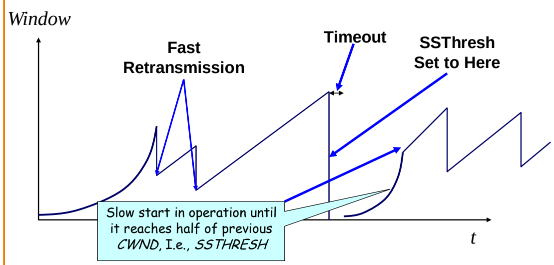
- Cut CWND <u>all the way to 1 MSS</u> on **timeout**
  - Set ssthresh to cwnd/2

- Never drop CWND below 1 MSS

Summary of Increase

- "Slow-start": increase cwnd by MSS for each ack
- Leave slow-start regime when either:
  - cwnd > SSThresh
  - Packet drop
- Enter AIMD regime
  - Increase by MSS for each window's worth of acked data

Repeating Slow Start After Timeout



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

More Advanced Fast Restart

- Set ssthresh to cwnd/2
- Set cwnd to cwnd/2 + 3
  - for the 3 dup acks already seen
- Increment cwnd by 1 MSS for each additional duplicate ACK
- After receiving new ACK, reset cwnd to ssthresh

Throughput Equation

In what follows refer to cwnd in units of MSS

Calculation on Simple Model

- Assume loss occurs whenever cwnd reaches W
  - Recovery by fast retransmit
- Window: W/2, W/2+1, W/2+2, …W, W/2, …
  - W/2 RTTs, then drop, then repeat
- Average throughput: .75W(MSS/RTT)
  - One packet dropped out of (W/2)*(3W/4)
  - Packet drop rate p = $(8/3)\ W^{-2}$
- Throughput = (MSS/RTT) sqrt(3/2p)

Some implications

- Flows get throughput inversely proportional to RTT
  - Fairness issue?
- One can dispense with TCP and just match eqtn:
  - Equation-based congestion control
  - Measure drop percentage p, and set rate accordingly
  - Useful for streaming applications

How does this work at high speed?

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

43

Adapting TCP to High Speed

- One approach: once speed is past some threshold, change equation to $p^{-.8}$ rather than $p^{-.5}$
- We will discuss other approaches next time…

44

Why AIMD?

In what follows refer to cwnd in units of MSS

45

Three Congestion Control Challenges

- Single flow adjusting to bottleneck bandwidth
  - Without any *a priori* knowledge
  - Could be a Gbps link; could be a modem
- Single flow adjusting to variations in bandwidth
  - When bandwidth decreases, must lower sending rate
  - When bandwidth increases, must increase sending rate
- Multiple flows sharing the bandwidth
  - Must avoid overloading network
  - And share bandwidth "fairly" among the flows

46

Problem #1: Single Flow, Fixed BW

- Want to get a first-order estimate of the available bandwidth
  - Assume bandwidth is fixed
  - Ignore presence of other flows
- Want to start slow, but rapidly increase rate until packet drop occurs ("slow-start")
- Adjustment:
  - cwnd initially set to 1 (MSS)
  - cwnd++ upon receipt of ACK

47

Problems with Slow-Start

- Slow-start can result in many losses
  - Roughly the size of cwnd ~ BW*RTT
- Example:
  - At some point, cwnd is enough to fill "pipe"
  - After another RTT, cwnd is double its previous value
  - All the excess packets are dropped!
- Need a more gentle adjustment algorithm once have rough estimate of bandwidth
  - Rest of design discussion focuses on this

48

Problem #2: Single Flow, Varying BW

Want to track available bandwidth

- Oscillate around its current value
- If you never send more than your current rate, you won't know if more bandwidth is available

Possible variations: (in terms of change per RTT)

- Multiplicative increase or decrease:
  cwnd□ cwnd * / a
- Additive increase or decrease:
  cwnd□ cwnd +- b

49

Four alternatives

- AIAD: gentle increase, gentle decrease
- AIMD: gentle increase, drastic decrease
- MIAD: drastic increase, gentle decrease
  - too many losses: eliminate
- MIMD: drastic increase and decrease

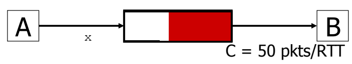
50

Problem #3: Multiple Flows

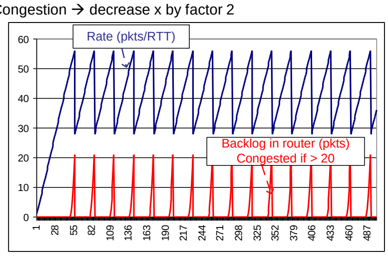
- Want steady state to be "fair"
- Many notions of fairness, but here just require two identical flows to end up with the same bandwidth
- This eliminates MIMD and AIAD
  - As we shall see…
- AIMD is the only remaining solution!
  - Not really, but close enough….

51

Buffer and Window Dynamics

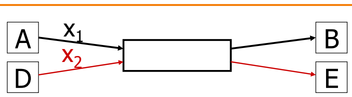


- No congestion → x increases by one packet/RTT every RTT
- Congestion → decrease x by factor 2

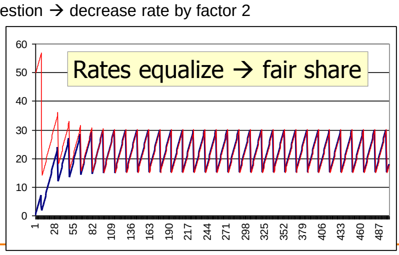


52

AIMD Sharing Dynamics



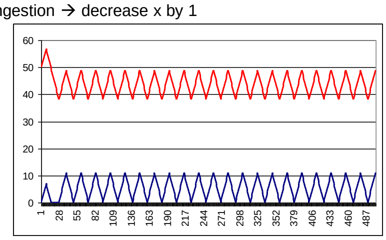
- No congestion → rate increases by one packet/RTT every RTT
- Congestion → decrease rate by factor 2



53

AIAD Sharing Dynamics

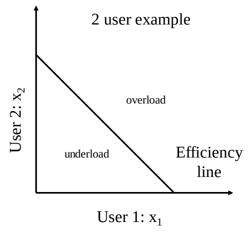
- No congestion → x increases by one packet/RTT every RTT
- Congestion → decrease x by 1



54

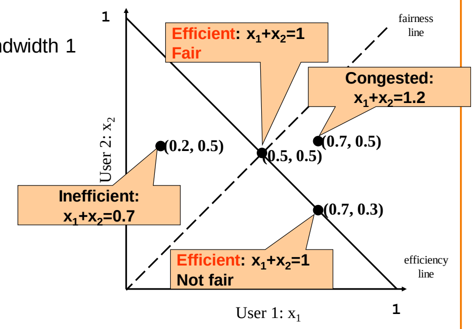
Simple Model of Congestion Control

- Two TCP connections
  - Rates x_1 and x_2
- Congestion when sum>1
- Efficiency: sum near 1
- Fairness: x’s converge



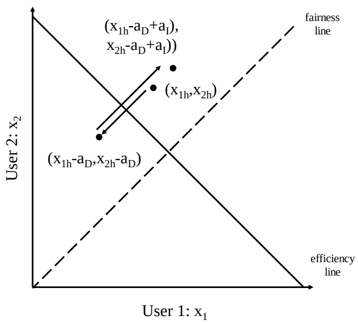
Example

- Total bandwidth 1



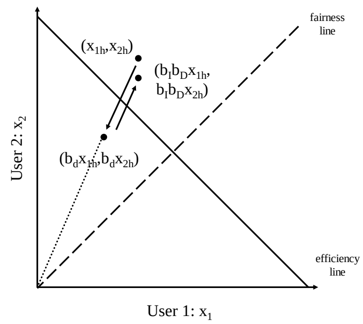
AIAD

- Increase: $x + a_I$
- Decrease: $x - a_D$
- Does not converge to fairness



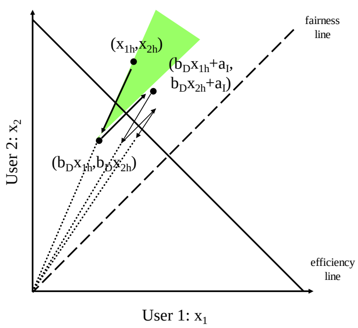
MIMD

- Increase: $x*b_I$
- Decrease: $x*b_D$
- Does not converge to fairness



AIMD

- Increase: $x+a_D$
- Decrease: $x*b_D$
- Converges to fairness



AIMD is only “fair” choice

- **But how fair is it?**
- Bandwidth depends on RTT
- Hosts that send more flows get more bandwidth

Thursday: Advanced Topics in CC

61