Network Congestion: Context

• Best-effort network does not “block” calls
  – So, they can easily become overloaded
  – Congestion == “Load higher than capacity”

• Examples of congestion
  – Link layer: Ethernet frame collisions
  – Network layer: full IP packet buffers

• Excess packets are simply dropped
  – And the sender can simply retransmit
Problem: Congestion Collapse

- Easily leads to *congestion collapse*
  - Senders retransmit the lost packets
  - Leading to even *greater* load
  - ... and even *more* packet loss

Increase in load that results in a *decrease* in useful work done.
Detect and Respond to Congestion

• What does the end host see?
• What can the end host change?
• Distributed Resource Sharing
Detecting Congestion

• Link layer
  – Carrier sense multiple access
  – Seeing your own frame collide with others

• Network layer
  – Observing end-to-end performance
  – Packet delay or loss over the path
Responding to Congestion

• Upon detecting congestion
  – Decrease the sending rate

• But, what if conditions change?
  – If more bandwidth becomes available,
  – ... unfortunate to keep sending at a low rate

• Upon not detecting congestion
  – Increase sending rate, a little at a time
  – See if packets get through
TCP seeks “Fairness”
Phase Plots

User 1's Allocation $x_1$ vs. User 2's Allocation $x_2$

Fairness Line $x_1 = x_2$
Phase Plots

User 1’s Allocation \( x_1 \)

User 2’s Allocation \( x_2 \)

Overload

Fairness Line
\( x_1 = x_2 \)

Max Throughput

Under utilization
Phase Plots

User 1’s Allocation $x_1$

User 2’s Allocation $x_2$

Fairness Line $x_1 = x_2$

Overload

Under utilization

Max Throughput
Phase Plots

User 1’s Allocation $x_1$

User 2’s Allocation $x_2$

Fairness Line $x_1 = x_2$

Max Throughput

Overload

Under utilization
Phase Plots

User 1’s Allocation $x_1$

User 2’s Allocation $x_2$

Under utilization

Max Throughput

Fairness Line $x_1 = x_2$

Overload
Phase Plots

User 1’s Allocation $x_1$

User 2’s Allocation $x_2$

Overload

Fairness Line $x_1 = x_2$

Optimal point

Efficiency Line

Under utilization
Additive Increase/Decrease

User 1's Allocation \( x_1 \)

User 2's Allocation \( x_2 \)

Fairness Line \( x_1 = x_2 \)

Efficiency Line

AIAD
Multiplicative Increase/Decrease

Multiplicative Increase/Decrease

User 1’s Allocation $x_1$

User 2’s Allocation $x_2$

MIMD

Fairness Line $x_1 = x_2$

Efficiency Line

15
Additive Increase / Multiplicative Decrease

User 1’s Allocation $x_1$

User 2’s Allocation $x_2$

Fairness Line $x_1 = x_2$

Efficiency Line

AIMD
TCP Congestion Control

- Additive increase, multiplicative decrease
  - On packet loss, divide congestion window in half
  - On success for last window, increase window linearly
Why Multiplicative?

• Respond aggressively to bad news
  – Congestion is (very) bad for everyone
  – Need to react aggressively

Examples of exponential backoff:
  – TCP: divide sending rate in \textit{half}
  – Ethernet: \textit{double} retransmission timer

• Nice theoretical properties
  – Makes efficient use of network resources
TCP Congestion Control
Congestion in a Drop-Tail FIFO Queue

- Access to the bandwidth: first-in first-out queue
  - Packets transmitted in the order they arrive

- Access to the buffer space: drop-tail queuing
  - If the queue is full, drop the incoming packet
How it Looks to the End Host

• Delay: Packet experiences high delay
• Loss: Packet gets dropped along path

• How does TCP sender learn this?
  – Delay: Round-trip time estimate
  – Loss: Timeout and/or duplicate acknowledgments
TCP Congestion Window

• Each TCP sender maintains a congestion window
  – Max number of bytes to have in transit (not yet ACK’d)

• Adapting the congestion window
  – Decrease upon losing a packet: backing off
  – Increase upon success: optimistically exploring
  – Always struggling to find right transfer rate

• Tradeoff
  – Pro: avoids needing explicit network feedback
  – Con: continually under- and over-shoots “right” rate
Additive Increase, Multiplicative Decrease

• How much to adapt?
  – Additive increase: On success of last window of data, increase window by 1 Max Segment Size (MSS)
  – Multiplicative decrease: On loss of packet, divide congestion window in half

• Much quicker to slow down than speed up?
  – Over-sized windows (causing loss) are much worse than under-sized windows (causing lower throughput)
  – AIMD: A necessary condition for stability of TCP
Leads to the TCP “Sawtooth”

Window

Loss

halved

Time
Receiver Window vs. Congestion Window

• Flow control
  – Keep a *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
  – Sender TCP window = \( \min \{ \text{congestion window}, \text{receiver window} \} \)
Starting a New Flow
How Should a New Flow Start?

Start slow (a small CWND) to avoid overloading network

But, could take a long time to get started!
“Slow Start” Phase

• Start with a small congestion window
  – Initially, CWND is 1 MSS
  – So, initial sending rate is MSS / RTT

• Could be pretty wasteful
  – Might be much less than 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 rate exponentially until the first loss
Slow Start in Action

Double CWND per round-trip time
Slow Start and the TCP Sawtooth

- TCP originally had *no* congestion control
  - Source would start by sending entire receiver window
  - Led to congestion collapse!
  - “Slow start” is, comparatively, slower
Two Kinds of Loss in TCP

- **Timeout vs. Triple Duplicate ACK**
  - Which suggests network is in worse shape?

- **Timeout**
  - If entire window was lost, buffers may be full
  - ...blasting entire CWND would cause another burst
  - ...be aggressive: start over with a low CWND

- **Triple duplicate ACK**
  - Might be do to bit errors, or “micro” congestion
  - ...react less aggressively (halve CWND)
Repeating Slow Start After Timeout

Window

timeout

$\text{timeout}$
Repeating Slow Start After Timeout

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

Slow start until reaching half of previous cwnd.
Repeating Slow Start After Idle Period

• Suppose a TCP connection goes idle for a while

• Eventually, the network conditions change
  – Maybe many more flows are traversing the link

• Dangerous to start transmitting at the old rate
  – Previously-idle TCP sender might blast network
  – ... causing excessive congestion and packet loss

• So, some TCP implementations repeat slow start
  – Slow-start restart after an idle period
Fairness
TCP Achieves a Notion of Fairness

• Effective utilization is not only goal
  – We also want to be *fair* to various flows

• Simple definition: equal bandwidth shares
  – N flows that each get 1/N of the bandwidth?

• But, what if flows traverse different paths?
  – Result: bandwidth shared in proportion to RTT
What About Cheating?

• Some senders are more fair than others
  – Using multiple TCP connections in parallel (BitTorrent)
  – Modifying the TCP implementation in the OS
    • Some cloud services start TCP at > 1 MSS
  – Use the User Datagram Protocol

• What is the impact?
  – Good senders *slow down* to make room for you
  – You get an *unfair share* of the bandwidth
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

• Fundamental tensions
  – Feedback from the network?
  – Enforcement of “TCP friendly” behavior?