Flow Control

- Part of TCP specification (even before 1988)
- Goal: not send more data than the receiver can handle
- Sliding window protocol
- Receiver uses window header field to tell sender how much space it has
Flow Control

• **Receiver:** $\text{AdvertisedWindow} = \text{MaxRcvBuffer} - ((\text{NextByteExpected} - 1) - \text{LastByteRead})$

• **Sender:** $\text{LastByteSent} - \text{LastByteAcked} \leq \text{AdvertisedWindow}$

  $\text{EffectiveWindow} = \text{AdvertisedWindow} - (\text{BytesInFlight})$

  $\text{LastByteWritten} - \text{LastByteAcked} \leq \text{MaxSendBuffer}$
Delayed Acknowledgments

• Goal: Piggy-back ACKs on data
  – Delay ACK for 200ms in case application sends data
  – If more data received, immediately ACK second segment
  – Note: never delay duplicate ACKs (if missing a segment)
Limitations of Flow Control

- Network may be the bottleneck
- Signal from receiver not enough!
- Sending too fast will cause queue overflows, heavy packet loss
- Flow control provides *correctness*
- Need more for performance: congestion control
Congestion Control

- Goal: pipes full
- Set rate accordingly to bottleneck
Congestion Control

• Goal: do not send more data than the network can take

• 3 Key Challenges
  – Determining the available capacity
  – Adjusting to changes in the available capacity
  – Sharing capacity between flows
TCP Congestion Control

• 3 Key Challenges
  – Determining the available capacity in the first place
  – Adjusting to changes in the available capacity
  – Sharing capacity between flows

• Idea
  – Each source determines network capacity for itself
  – Rate is determined by window size
  – Uses implicit feedback (drops, delay)
  – ACKs pace transmission (self-clocking)
Dealing with Congestion

• TCP keeps congestion and flow control windows
  – Transmit minimum of the two controls
• Sending rate: \( \sim \text{Window/RTT} \)
• The key here is how to set the congestion window to respond to congestion signals
Starting Up

• Before TCP Tahoe
  – On connection, nodes send full (rcv)window of packets
  – Retransmit packet immediately after its timer expires

• Result: window-sized bursts of packets in network
Bursts of Packets

Graph from Van Jacobson and Karels, 1988
Determining Initial Capacity

- Question: how do we set $w$ initially?
  - Should start at 1MSS (to avoid overloading the network)
  - Could increase additively until we hit congestion
  - May be too slow on fast network

- Start by doubling $w$ each RTT
  - Then will dump at most one extra window into network
  - This is called *slow start*

- *Slow start*, this sounds quite fast!
  - In contrast to initial algorithm: sender would dump entire *flow control* window at once
Startup behavior with Slow Start
Slow Start

Figure 3: Startup behavior of TCP without Slow-start

Figure 4: Startup behavior of TCP with Slow-start

From [Jacobson88]
Slow start implementation

• Let $w$ be the size of the window in bytes
  – We have $w$/MSS segments per RTT
• We are doubling $w$ after each RTT
  – We receive $w$/MSS ACKs each RTT
  – So we can set $w = w + \text{MSS}$ on every ack
• At some point we hit the network limit.
  – Experience loss
  – We are at most one window size above the limit
  – Remember this: ssthresh and reduce window
Chronology of a Slow-start

One Round Trip Time

One Packet Time

0R

1R

2R

3R

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15
Slow Start

• We double cwnd every round trip
• We are still sending min (cwnd,rcvwnd) pkts
• Continue until ssthresh estimate or pkt drop

http://en.wikipedia.org/wiki/Slow-start
Congestion Collapse

From [Chiu89]
Dealing with Congestion

• Assume losses are due to congestion
• After a loss, reduce congestion window
  – How much to reduce?
• Idea: conservation of packets at equilibrium
  – Want to keep roughly the same number of packets in network
  – Analogy with water in fixed-size pipe
  – Put new packet into network when one exits
So, if packets after the first burst are sent only in response to an ack, the sender’s packet spacing will exactly match the packet time on the slowest link in the path.
Congestion Signal

• If packet loss is (almost) due to congestion, we have a good candidate for the ‘networked is congested’ signal.

• Is it valid?
How much to reduce window?

• What happens under congestion?
  – Exponential increase in congestion

• Sources must decrease offered rate exponentially
  – i.e, multiplicative decrease in window size
  – TCP chooses to cut window in half
How to use extra capacity?

• Network signals congestion, but says nothing of underutilization
  – Senders constantly try to send faster, see if it works
  – So, increase window if no losses... By how much?

• Multiplicative increase?
  – Easier to saturate the network than to recover
  – Too fast, will lead to saturation, wild fluctuations

• Additive increase?
  – Won’t saturate the network
Chiu Jain Phase Plots

Fair: \( A = B \)

Efficient: \( A + B = C \)

Goal: fair and efficient!
Chiu Jain Phase Plots

Flow Rate A

Flow Rate B

Fair: \( A = B \)

Efficient: \( A + B = C \)

MIMD
Chiu Jain Phase Plots

Flow Rate A

Flow Rate B

Fair: $A = B$

Efficient: $A + B = C$

AIAD
Chiu Jain Phase Plots

Flow Rate A

Flow Rate B

Fair: A = B

Efficient: A + B = C

AIMD
AIMD Implementation

• In practice, send MSS-sized segments
  – Let window size in bytes be $w$ (a multiple of MSS)
• Increase:
  – After $w$ bytes ACKed, could set $w = w + \text{MSS}$
  – Smoother to increment on each ACK
    • $w = w + \text{MSS} \times \text{MSS}/w$
    • (receive $w$/MSS ACKs per RTT, increase by $\text{MSS}/(w$/MSS) for each)
• Decrease:
  – After a packet loss, $w = w/2$
  – But don’t want $w < \text{MSS}$
  – So react differently to multiple consecutive losses
  – Back off exponentially (pause with no packets in flight)
AIMD Trace

- AIMD produces sawtooth pattern of window size
  - Always probing available bandwidth
Putting it together

• TCP has two states: Slow Start (SS) and Congestion Avoidance (CA)
• A window size threshold governs the state transition
  – Window <= threshold: SS
  – Window > threshold: congestion avoidance
• States differ in how they respond to ACKs
  – Slow start: \( w = w + MSS \)
  – Congestion Avoidance: \( w = w + \frac{MSS^2}{w} \) (1 MSS per RTT)
• On loss event: set \( w = 1 \), slow start
How to Detect Loss

• Timeout

• Any other way?
  – Gap in sequence numbers at receiver
  – Receiver uses cumulative ACKs: drops => duplicate ACKs

• 3 Duplicate ACKs considered loss
Putting it all together

- **cwnd**
- **Slow Start**
- **Timeout**
- **AIMD**
- **ssthresh**
- **Timeout**
- **Slow Start**
- **AIMD**

Time
RTT

• We want an estimate of RTT so we can know a packet was likely lost, and not just delayed
• Key for correct operation
• Challenge: RTT can be highly variable
  – Both at long and short time scales!
• Both average and variance increase a lot with load
• Solution
  – Use exponentially weighted moving average (EWMA)
  – Estimate deviation as well as expected value
  – Assume packet is lost when time is well beyond reasonable deviation
Originally

- \( \text{EstRTT} = (1 - \alpha) \times \text{EstRTT} + \alpha \times \text{SampleRTT} \)
- \( \text{Timeout} = 2 \times \text{EstRTT} \)
- **Problem 1:**
  - in case of retransmission, ack corresponds to which send?
  - Solution: only sample for segments with no retransmission
- **Problem 2:**
  - does not take variance into account: too aggressive when there is more load!
Jacobson/Karels Algorithm (Tahoe)

- EstRTT = \((1 - \alpha) \times \text{EstRTT} + \alpha \times \text{SampleRTT}\)
  - Recommended \(\alpha\) is 0.125
- DevRTT = \((1 - \beta) \times \text{DevRTT} + \beta \times |\text{SampleRTT} - \text{EstRTT}|\)
  - Recommended \(\beta\) is 0.25
- Timeout = EstRTT + 4 DevRTT
- For successive retransmissions: use exponential backoff
Old RTT Estimation
Tahoe RTT Estimation

![Tahoe RTT Estimation Graph](image-url)
Slow start every time?

- Losses have large effect on throughput
- Fast Recovery (TCP Reno)
  - Same as TCP Tahoe on Timeout: $w = 1$, slow start
  - On triple duplicate ACKs: $w = w/2$
  - Retransmit missing segment (fast retransmit)
  - Stay in Congestion Avoidance mode
Figure 3.37 Resending a segment after triple duplicate ACK
Fast Recovery and Fast Retransmit
3 Challenges Revisited

• Determining the available capacity in the first place
  – Exponential increase in congestion window
• Adjusting to changes in the available capacity
  – Slow probing, AIMD
• Sharing capacity between flows
  – AIMD
• Detecting Congestion
  – Timeout based on RTT
  – Triple duplicate acknowledgments
• Fast retransmit/Fast recovery
  – Reduces slow starts, timeouts