Lecture 15: Congestion Control

Slides used with permissions from Edward W. Knightly, T. S. Eugene Ng, Ion Stoica, Hui Zhang
Abstract View

- We ignore internal structure of network and model it as having a single bottleneck link
Three Congestion Control Problems

• Adjusting to bottleneck bandwidth
• Adjusting to variations in bandwidth
• Sharing bandwidth between flows
Single Flow, Fixed Bandwidth

- Adjust rate to match bottleneck bandwidth
  - without any *a priori* knowledge
  - could be gigabit link, could be a modem
Single Flow, Varying Bandwidth

- Adjust rate to match instantaneous bandwidth
- Bottleneck can change because of a routing change
Multiple Flows

Two Issues:

• Adjust total sending rate to match bottleneck bandwidth
• Allocation of bandwidth between flows
General Approaches

• Send without care
  – many packet drops
  – could cause congestion collapse

• Reservations
  – pre-arrange bandwidth allocations
  – requires negotiation before sending packets

• Pricing
  – don’t drop packets for the high-bidders
  – requires payment model
General Approaches (cont’d)

• Dynamic Adjustment (TCP)
  – Every sender probe network to test level of congestion
  – speed up when no congestion
  – slow down when congestion
  – suboptimal, messy dynamics, simple to implement

  – Distributed coordination problem!
TCP Congestion Control

• TCP connection has window
  – controls number of unacknowledged packets

• Sending rate: ~Window/RTT

• Vary window size to control sending rate

• Introduce a new parameter called congestion window (cwnd) at the sender
  – Congestion control is mainly a sender-side operation
Congestion Window (cwnd)

- Limits how much data can be in transit
- Implemented as # of bytes
- Described as # packets in this lecture

MaxWindow = \text{min}(\text{cwnd}, \text{AdvertisedWindow})

\text{EffectiveWindow} = \text{MaxWindow} - (\text{LastByteSent} - \text{LastByteAcked})
Two Basic Components

• Detecting congestion

• Rate adjustment algorithm (change cwnd size)
  – depends on congestion or not
Detecting Congestion
Detecting Congestion

• Packet dropping is best sign of congestion
  – delay-based methods are hard and risky

• How do you detect packet drops? ACKs
  – TCP uses ACKs to signal receipt of data
  – ACK denotes last contiguous byte received
    • actually, ACKs indicate next segment expected

• Two signs of packet drops
  – No ACK after certain time interval: time-out
  – Several duplicate ACKs (ignore for now)
Detecting Congestion

• Packet dropping is best sign of congestion
  – delay-based methods are hard and risky

• How do you detect packet drops? ACKs
  – TCP uses ACKs to signal receipt of data
  – ACK denotes last contiguous byte received
    • actually, ACKs indicate next segment expected

• Two signs of packet drops
  – No ACK after certain time interval: time-out
  – Several duplicate ACKs (ignore for now)
Detecting Congestion

• Packet dropping is best sign of congestion
  – delay-based methods are hard and risky

• How do you detect packet drops? ACKs
  – TCP uses ACKs to signal receipt of data
  – ACK denotes last contiguous byte received
    • actually, ACKs indicate next segment expected

• Two signs of packet drops
  – No ACK after certain time interval: time-out
  – Several duplicate ACKs (ignore for now)

• May not work well for wireless networks, why?
Sliding (Congestion) Window

- Sliding window: each ACK = permission to send a new packet
  - Ex. cwnd = 3
Self-clocking

- If we have a large window, ACKs “self-clock” the data to the rate of the bottleneck link.
- Observe: received ACK spacing $\approx$ bottleneck bandwidth.
Rate Adjustment

• Basic structure:
  – Upon receipt of ACK (of new data): increase rate
    • Data successfully delivered, perhaps can send faster
  – Upon detection of loss: decrease rate

• But what increase/decrease functions should we use?
  – Depends on what problem we are solving
Fairness?

![Diagram showing a linear relationship between Connection 1 throughput and Connection 2 throughput with a point R]
Fairness?

Connection 1 throughput

Connection 2 throughput

equal bandwidth share
Fairness?

Two competing sessions:

Connection 1 throughput

Connection 2 throughput

equal bandwidth share
Fairness?

Two competing sessions:

- Additive increase (AI) gives slope of 1, as throughout increases
Fairness?

Two competing sessions:
- Additive increase (AI) gives slope of 1, as throughout increases
- Multiplicative decrease (MD) decreases throughput proportionally

![Graph showing equal bandwidth share between two connections.](image)
Fairness?

Two competing sessions:
- Additive increase (AI) gives slope of 1, as throughout increases
- Multiplicative decrease (MD) decreases throughput proportionally

Connection 1 throughput vs. Connection 2 throughput

Equal bandwidth share

Congestion avoidance: additive increase
Fairness?

Two competing sessions:

- Additive increase (AI) gives slope of 1, as throughout increases
- Multiplicative decrease (MD) decreases throughput proportionally
Fairness?

Two competing sessions:
- Additive increase (AI) gives slope of 1, as throughout increases
- Multiplicative decrease (MD) decreases throughput proportionally

Diagram:
- Equal bandwidth share
- Loss: decrease window by factor of 2
  - Connection 1 throughput vs. Connection 2 throughput
Fairness?

Two competing sessions:

- Additive increase (AI) gives slope of 1, as throughout increases
- Multiplicative decrease (MD) decreases throughput proportionally

![Graph showing equal bandwidth share between two connections](image)

Connection 1 throughput vs. Connection 2 throughput
Fairness?

Two competing sessions:

- Additive increase (AI) gives slope of 1, as throughout increases
- Multiplicative decrease (MD) decreases throughput proportionally

Graph:
- Axes: Connection 1 throughput vs. Connection 2 throughput
- Line: Equal bandwidth share
- Point: Fair and link fully utilized (rate R)
AIMD

A

B

C

D

E

x

y

C

y

x
AIMD

A

B

c

D

d

E

C

x

y

X

Y
AIMD

A
B
C

D
E

x
y

C

X

Y

X

Alan Mislove
amislove at ccs.neu.edu
Northeastern University
Limit rates:
\[ x = y \]
AIMD Sharing Dynamics

- No congestion $\Rightarrow$ rate increases by one packet/RTT every RTT
- Congestion $\Rightarrow$ decrease rate by factor 2

Rates equalize $\Rightarrow$ fair share
Limit rates: $x$ and $y$ depend on initial values.
AIAD Sharing Dynamics

- No congestion $\rightarrow$ $x$ increases by one packet/RTT every RTT
- Congestion $\rightarrow$ decrease $x$ by 1
TCP Model

• Derive an expression for the steady state throughput as a function of
  – RTT
  – Loss probability

• Assumptions
  – Each packet dropped with iid probability \( p \)

• Methodology: analyze “average” cycle in steady state
  – How many packets are transmitted per cycle?
  – What is the duration of a cycle?
Cycles in Steady State

- Denote $W$ as the (mean) maximum achieved window
- What is the slope of the line?
- What are the key values on the time axis?
Cycle Analysis

$W$ increase by 1 per RTT

$\text{pkts xmitted/cycle} = \text{area} = \left( \frac{W}{2} \right)^2 + \frac{1}{2} \left( \frac{W}{2} \right)^2 = \frac{3}{8} W^2$
Throughput

\[ \text{throughput} = \frac{\text{pkts xmitted/cycle}}{\text{time/cycle}} = \frac{3}{8} \frac{W^2}{RTT\left(\frac{W}{2}\right)} \]

- What is \( W \) as a function of \( p \)?
  How long does a cycle last until a drop?
Cycle Length

Let $\alpha$ index packet loss that ends cycle.

$$P(\alpha = k) = P(k - 1 \text{ pkts not lost, } k\text{th pkt lost})$$

$$= (1 - p)^{k-1} p$$

$$\Rightarrow E(\alpha) = \sum_{k=1}^{\infty} k(1 - p)^{k-1} p = \frac{1}{p}$$

$$\Rightarrow \frac{1}{p} = \frac{3}{8} W^2 \quad \Rightarrow \quad W = \sqrt{\frac{8}{3p}}$$
TCP Model

$$ \text{throughput } T(p) = \frac{1}{p} \sqrt{\frac{8}{2 \sqrt{3} p}} = \frac{1}{RTT \sqrt{\frac{2}{3} p}} $$

• Note role of RTT. Is it “fair”?  
• A “macroscopic” model  
• Achieving this throughput is referred to as “TCP Friendly”
Adapting cwin

• So far: sliding window + self-clocking of ACKs
• How to know the best cwnd (and best transmission rate)?

• Phases of TCP congestion control
  1. Slow start (getting to equilibrium)
     1. Want to find this very very fast and not waste time
  2. Congestion Avoidance
     – Additive increase - gradually probing for additional bandwidth
     – Multiplicative decrease - decreasing cwnd upon loss/timeout
Phases of Congestion Control

- **Congestion Window** \((cwnd)\)
  Initial value is 1 MSS (=maximum segment size) counted as bytes

- **Slow-start threshold Value** \((ss\_thresh)\)
  Initial value is the advertised window size

- **slow start** \((cwnd < ss\_thresh)\)
- **congestion avoidance** \((cwnd >= ss\_thresh)\)
TCP: Slow Start

• Goal: discover roughly the proper sending rate quickly

• Whenever starting traffic on a new connection, or whenever increasing traffic after congestion was experienced:
  • Initialize $cwnd = 1$
  • Each time a segment is acknowledged, increment $cwnd$ by one ($cwnd++$).

• Continue until
  – Reach ss_thresh
  – Packet loss
Slow Start Illustration

• The congestion window size grows very rapidly

• TCP slows down the increase of $cwnd$ when $cwnd \geq ss_{\text{thresh}}$

• Observe:
  – Each ACK generates two packets
  – slow start increases rate exponentially fast (doubled every RTT)!
Slow Start Illustration

- The congestion window size grows very rapidly

- TCP slows down the increase of $cwnd$ when $cwnd \geq ss\_thresh$

- Observe:
  - Each ACK generates two packets
  - slow start increases rate exponentially fast (doubled every RTT)!
Slow Start Illustration

• The congestion window size grows very rapidly

• TCP slows down the increase of cwnd when $cwnd \geq ss_{\text{thresh}}$

• Observe:
  – Each ACK generates two packets
  – slow start increases rate exponentially fast (doubled every RTT)!
Slow Start Illustration

- The congestion window size grows very rapidly

- TCP slows down the increase of $cwnd$ when $cwnd \geq ss\_thresh$

- Observe:
  - Each ACK generates two packets
  - slow start increases rate exponentially fast (doubled every RTT)!
Slow Start Illustration

- The congestion window size grows very rapidly
- TCP slows down the increase of congestion window size when \( cwnd >= ss\_thresh \)

Observe:
- Each ACK generates two packets
- Slow start increases rate exponentially fast (doubled every RTT)!
Slow Start Illustration

- The congestion window size grows very rapidly

- TCP slows down the increase of cwnd when $\text{cwnd} \geq \text{ss\_thresh}$

- Observe:
  - Each ACK generates two packets
  - slow start increases rate exponentially fast (doubled every RTT)!
Slow Start Illustration

- The congestion window size grows very rapidly

- TCP slows down the increase of $cwnd$ when $cwnd \geq \text{ss\_thresh}$

- Observe:
  - Each ACK generates two packets
  - slow start increases rate exponentially fast (doubled every RTT)!
Slow Start Illustration

- The congestion window size grows very rapidly

- TCP slows down the increase of cwnd when \( cwnd \geq ss_{thresh} \)

- Observe:
  - Each ACK generates two packets
  - slow start increases rate exponentially fast (doubled every RTT)!
Slow Start Illustration

- The congestion window size grows very rapidly

- TCP slows down the increase of $cwnd$ when $cwnd \geq ss_{\text{thresh}}$

- Observe:
  - Each ACK generates two packets
  - slow start increases rate exponentially fast (doubled every RTT)!
Slow Start Illustration

- The congestion window size grows very rapidly

- TCP slows down the increase of $cwnd$ when $cwnd \geq ss\_thresh$

- Observe:
  - Each ACK generates two packets
  - slow start increases rate exponentially fast (doubled every RTT)!
Congestion Avoidance (After Slow Start)

• Slow Start figures out roughly the rate at which the network starts getting congested
• Congestion Avoidance continues to react to network condition
  – Probes for more bandwidth, increase cwnd if more bandwidth available
  – If congestion detected, aggressive cut back cwnd
### Congestion Avoidance: Additive Increase

- After exiting slow start, slowly increase cwnd to probe for additional available bandwidth
  - Competing flows may end transmission
  - May have been “unlucky” with an early drop

- **If** $cwnd > ss\_thresh$ **then**
  - each time a segment is acknowledged
  - increment $cwnd$ by $1/cwnd$ ($cwnd += 1/cwnd$).

- $cwnd$ is increased by one only if all segments have been acknowledged
  - Increases by 1 per RTT, vs. doubling per RTT
Example of Slow Start + Congestion Avoidance

Assume that \textit{ss\_thresh} = 8
Detecting Congestion via Timeout

• If there is a packet loss, the ACK for that packet will not be received

• The packet will eventually timeout
  – No ack is seen as a sign of congestion
Congestion Avoidance: Multiplicative Decrease

• Timeout = congestion

• Each time when congestion occurs,
  – ss_thresh is set to half the current size of the congestion window:
    \[ ss\_thresh = \frac{cwnd}{2} \]
  – cwnd is reset to one:
    \[ cwnd = 1 \]
  – and slow-start is entered
TCP illustration

- Slow Start
- Timeout
- Congestion Avoidance
- ss_thresh

Time

cwnd

ss_thresh
Responses to Congestion (Loss)

• There are algorithms developed for TCP to respond to congestion
  – TCP Tahoe - the basic algorithm (discussed previously)
  – TCP Reno - Tahoe + fast retransmit & fast recovery
    • Most end hosts today implement TCP Reno

• and many more:
  – TCP Vegas (research: use timing of ACKs to avoid loss)
  – TCP SACK (future deployment: selective ACK)
TCP Reno

- Problem with Tahoe: If a segment is lost, there is a long wait until timeout
- Reno adds a **fast retransmit** and **fast recovery** mechanism

- Upon receiving 3 duplicate ACKs, retransmit the presumed lost segment ("fast retransmit")
- But do not enter slow-start. Instead enter congestion avoidance ("fast recovery")
Fast Retransmit

- Resend a segment after 3 duplicate ACKs
  - remember a duplicate ACK means that an out-of-sequence segment was received
  - ACK-n means packets 1, ..., n all received

- Notes:
  - duplicate ACKs due to packet reordering!
Fast Recovery

• After a fast-retransmit
  – $cwnd = \frac{cwnd}{2}$ (vs. 1 in Tahoe)
  – $ss_{thresh} = cwnd$
  – i.e. starts congestion avoidance at new $cwnd$
    • Not slow start from $cwnd = 1$

• After a timeout
  – $ss_{thresh} = \frac{cwnd}{2}$
  – $cwnd = 1$
  – Do slow start
  – Same as Tahoe
Fast Retransmit and Fast Recovery

- Retransmit after 3 duplicate ACKs
  - prevent expensive timeouts
- Slow start only once per session (if no timeouts)
- In steady state, \( cwnd \) oscillates around the ideal window size.
TCP Congestion Control Summary

• Measure available bandwidth
  – slow start: fast, hard on network
  – AIMD: slow, gentle on network

• Detecting congestion
  – timeout based on RTT
    • robust, causes low throughput
  – Fast Retransmit: avoids timeouts when few packets lost
    • can be fooled, maintains high throughput

• Recovering from loss
  – Fast recovery: don’t set cwnd=1 with fast retransmits
TCP Reno Quick Review

• Slow-Start if $cwnd < ss\_thresh$
  – $cwnd++$ upon every new ACK (exponential growth)
  – Timeout: $ss\_thresh = cwnd/2$ and $cwnd = 1$

• Congestion avoidance if $cwnd \geq ss\_thresh$
  – Additive Increase Multiplicative Decrease (AIMD)
  – ACK: $cwnd = cwnd + 1/cwnd$
  – Timeout: $ss\_thresh = cwnd/2$ and $cwnd = 1$

• Fast Retransmit & Recovery
  – 3 duplicate ACKS (interpret as packet loss)
  – Retransmit lost packet
  – $cwnd=cwnd/2$, $ss\_thresh = cwnd$
TCP Reno Saw Tooth Behavior

- Initial Slowstart
- Slowstart to pace packets
- Fast Retransmit and Recovery
- Timeouts may still occur
Summary

- TCP Reno is the *de facto* standard for congestion control on the Internet

- AIMD or “TCP friendliness” is expected of distributed applications