Congestion Avoidance

Class 19
Review of General Principles

- **Co-existence** - cooperation with existing TCP mechanisms for adapting to congestion and with routers’ current practice of dropping packets in periods of congestion.

- **Congestion persistence** - the time scales that we are concerned with are congestion events that may last longer than a round-trip time.

- **Long-time flow** - we are interested in managing the congestion caused by flows that send enough packets so that they are still active when network feedback reaches them.

- **Asymmetric routing** - the path followed by data packets may be different from the path followed by the acknowledgment packets in the reverse direction.
Congestion Control Techniques

• Slow start - recap
• Dynamic window sizing
• Fast retransmit
• Fast recovery
Strict AIMD

- AIMD is useful when operating at near NW capacity - *sustained ops*
  - AIMD is too conservative to use always
- If just starting up, AIMD is too slow
  - Start-up defined as initial tx or following idle period
TCP Congestion Management Techniques

- **TCP Tahoe (BNR1):**
  - slow start, congestion avoidance, fast retransmit
  - Tahoe cooperates w/ other Tahoe TCPs to share bandwidth
- **TCP Reno (BNR2):**
  - fast recovery, header prediction (seg seqnum prediction)
  - Delayed ACKs
  - *widely implemented:* Most real TCPs in 2000 are “Reno”
- **TCP Vegas:**
  - Not widely accepted yet; for source-based Avoidance
Illustration of Slow Start and Congestion Avoidance

- Exponential Increase
- Round-trip times
- cwnd
- threshold
- timeout occurs
Slow Start

When a TCP sender initiates a connection, it doesn’t know if there is congestion present in the network ➔ Uses Slow Start

Starts by sending a packet, doubling the number of packets each RTT, until half of the receiver’s window is filled or packets are lost
Slow start phase

- initialize:
  - $\text{CongestionWindow} = 1$
  - for (each ACK)
    - $\text{CongestionWindow} \; \text{++}$
  - until
    - loss detection OR
    - $\text{CongestionWindow} > \text{ssthresh}$
How is Slow Start Slow?

Earlier TCP burst at a steady-rate of segments until routers overflowed.

Slow-start TCP sends only a few segments initially, but "catches up" rapidly - so it tests the waters.
Slow Start Situations

1. Upon connection initiation
   - No idea of network congestion level
     - No ideal of # of packets allowed in transit
   - Doubles CWindow until there’s a loss
     - Timeout results in cwnd = cwnd/2

2. Connection “goes dead” while waiting for timeout
   - Source sent as much data as advertisedWindow allowed
     - waiting for ACK
   - Eventually a single ACK is received, reopening the advertisedWindow
   - Source now knows something about the network
     - Has new & useful value for CongestionWindow (cwnd/2)
Slow Start Phases

- **Slow Start Phase**
  - Increase CongestionWindow *exponentially up to* a threshold (ssthresh)

- **Congestion Avoidance Phase**
  - Increase cwnd *linearly after* ssthresh

- **On Timeout**
  - CongestionWindow is reduced to the *initial value of 1 MSS*
  - Set ssthresh = 1/2 * current CongestionWindow
  - more precisely,
    - ssthresh = maximum of MIN(cwnd, receiver’s advertised window)/2 and 2 MSS
  - Slow Start is (re-)initiated
Slow Start Phases

- **Congestion avoidance**
- **Slow start**
- **Linear AIMD**
- **Slow start threshold (ssthresh i.e. CongestionThreshold)**
Congestion Avoidance Phase

/* CongestionWindow > threshold */
Until (loss detection) {
    After receiving ACK for all the segments in transit:
        CongestionWindow ++
}

ssthresh = CongestionWindow/2
CongestionWindow = 1
perform slow start
Dynamic Window Sizing

When congestion occurs, senders need to give time for the congestion to clear before sending at a high rate

- Reset $cwnd = 1$
- Set $ssthresh = cwnd/2$
- Increase $cwnd$ exponentially until $ssthresh$ is reached
- For $cwnd > ssthresh$ increase linearly
Slow Start/Congestion Avoidance Example

- Assume that $ssthresh = 8$

**Roundtrip times**

- $cwnd = 1$
- $cwnd = 2$
- $cwnd = 4$
- $cwnd = 8$
- $cwnd = 9$
- $cwnd = 10$

$ssthresh$
Putting Everything Together

Initially:

cwnd = 1;
ssthresh = infinite;

New ack received:

if (cwnd < ssthresh)
  /* Slow Start*/
  cwnd = cwnd + 1;
else
  /* Congestion Avoidance */
  cwnd = cwnd + 1/cwnd;

Timeout:

/* Multiplicative decrease */
ssthresh = win/2;
cwnd = 1;

while (next < unack + win)
  transmit next packet;

where win = min(cwnd, advert_win);

seq #

unack

next

win
The Full Sequence

- **Slow Start**
- **Timeout**
- **Congestion Avoidance**

Diagram shows the evolution of cwnd over time with stages of Slow Start, Timeout, and Congestion Avoidance.
Enhancing Congestion Control ➔
Congestion Avoidance

- TCP Tahoe
  - Fast Retransmit
- TCP Reno
  - Fast Recovery
TCP Tahoe

- **Slow Start, Fast Retransmit**
- **Detects congestion using timeouts**
- **Initialization**
  - cwnd initialized to 1;
  - ssthresh initialized to 1/2 MaxWin
- **Upon timeout**
  - ssthresh = 1/2 cwnd, cwnd = 1
  - enter slow start
Original Problem with TCP CC - Packet Loss Detection

Wait for TCP Sender Retransmission Time Out (RTO)
- This is a problem because RTO is a performance killer

- In BSD TCP implementation, RTO is usually more than 1 second
  - the granularity of RTT estimate is 500 ms
  - retransmission timeout is at least two times RTT
- Solution: Don’t wait for RTO to expire
Fast Retransmit

• Recall that TCP may send delayed acknowledgements
• TCP is required to send an immediate acknowledgement for packets received out of order
• If a sender receives 3 consecutive out-of-order acknowledgements
  - it assumes that there must be packets that were lost
  - so it retransmits them without waiting for the RTO timer to expire
Fast Retransmit

- **Resend segment after 3 dupACKs**
  - note a duplicate ACK means that an out-of-sequence segment was received

- **Notes:**
  - duplicate ACKs due to packet reordering
  - if window is small don’t get duplicate ACKs

```
cwnd = 1
```

```
cwnd = 2
```

```
cwnd = 4
```

3 duplicate ACKs
Fast Retransmit

- Retransmit after 3 duplicated acks
  - prevent expensive timeouts
- No need to slow start again
- At steady state, \( cwnd \) oscillates around the optimal window size
TCP Reno

- Fast Retransmit, Fast Recovery
- Detects congestion loss using *timeouts* as well as *duplicate ACKs*
- On timeout, TCP Reno behaves same as TCP Tahoe
- On Duplicate ACKs
  - uses fast retransmit:
  - skips slow start and goes directly into congestion avoidance phase
  - ssthresh = 1/2 cwnd; cwnd = ssthresh
Fast Recovery

• When Fast Retransmit signals congestion (due to lost packet)
  → use the ACKs outstanding (still in the pipe) to clock the sending of packets

• Removes the slow start phase between Fast Retransmit start and AIMD phases
Fast Recovery

• If you needed to do a Fast Retransmit:
  - set $ssthresh = cwnd/2$ (at least 1 MSS)
  - Do not set $cwnd = 1$
  - retransmit the missing segment

• enter congestion avoidance phase
  - set $cwnd$ to $ssthresh + 3 \times \text{segment size}$
  - increment $cwnd$ linearly

• If RTO expires then set $cwnd = 1$
Congestion Avoidance
Congestion Avoidance

• We have seen TCP’s approach:
  - detect congestion after it happens
  - repeatedly increase load in an effort to find the point at
    which congestion occurs, and then back off

• Alternatively:
  - we can try to predict congestion and reduce rate before
    loss occurs
  - this is called congestion avoidance

• Two approaches for Congestion Avoidance:
  - Router based (router-centric) congestion avoidance
    • RED, DECBit
  - Source based (host-centric) congestion avoidance
    • TCP Vegas
Router congestion notification

- Router has unified view of queuing behavior
- Routers can distinguish between transient and persistent queuing delays
- Routers can inform source of congestion by explicit or implicit mechanisms
  - DECbit: *explicit* way by setting congestion bit on in packets from source
  - RED: *implicitly* by dropping packet. It is developed in conjunction with TCP
DECbit & End Hosts

- Each router monitors the load and explicitly notifies the end nodes when congestion is about to occur
  - If avg. queue length is $\geq 1$, then notify
- Notification is implemented by setting a binary congestion bit
- Router attempts to balance throughput against delay
DECbit & End Hosts

• The destination host then copies the bit into ACK

• Source adjusts its sending rate to avoid congestion
  
  - If less than 50% packets/ACKs have the bit set, increase congestion window by 1
  
  - If 50% or more packets have congestion bit set, decrease congestion window to 0.875 of previous value

• 50% was chosen based on modeling - corresponds to the peak of the power curve (throughput : delay ratio)

• Increase by 1 / decrease by 0.875 (AIMD) stabilizes system
Random Early Detection (RED)

- Similar to DECbit in that
  - each router monitors its own queue length
  - detects impending congestion & notifies source

- Differs from DECbit in that
  - Notification is implicit
    - just drops the packet (TCP will timeout / dupACK)
    - could make explicit by marking the packet
  - Decision method for \textit{when} to drop (signal) and \textit{which} packet to drop
RED

• **Observation**
  - Congestion is transient per RTT time
  - TCP detects congestion from loss - after queues are exhausted in Drop tail scheme.
  - Allowing queues to build up to the fullest extent increases end-to-end delay
  - Better to drop before congestion builds up and notify user to slow down
    - Better to drop a few now than many later
  - What the criteria for when / what to drop?
RED

• **Aim**
  - keep throughput high and delay low
  - do not reach queue limits and drop indiscriminately
    → that’s the drop tail mechanism
  - Keep to average queue length
  - accommodate bursts
    • do not drop for transient queue build-ups

• **Early random drop**
  - drop an arriving packet with some *drop probability*
    whenever the queue length exceeds some *drop level*
RED Details

- Maintain running average of queue length
- Compute average queue length thusly

\[
\text{AvgLen} = (1 - \text{Weight}) \times \text{AvgLen} + \\
\text{Weight} \times \text{SampleLen}
\]

\(0 < \text{Weight} < 1\) (usually 0.002)

\text{SampleLen} is queue length each time a packet arrives

![Diagram showing MaxThreshold, MinThreshold, and AvgLen]
RED Details (cont)

- Two queue length *thresholds*

  if $\text{AvgLen} \leq \text{MinThreshold}$ then
  queue the packet

  if $\text{MinThreshold} < \text{AvgLen} < \text{MaxThreshold}$ then
  calculate probability $P$
  drop arriving packet with probability $P$

  if $\text{MaxThreshold} \leq \text{AvgLen}$ then
  drop arriving packet
RED Details (cont’d)

- **Computing probability P**

  \[
  \text{TempP} = \frac{\text{MaxP} \times (\text{AvgLen} - \text{MinThreshold})}{(\text{MaxThreshold} - \text{MinThreshold})}
  \]

  \[
  P = \frac{\text{TempP}}{1 - \text{count} \times \text{TempP}}
  \]

- **Drop Probability Curve**

  # of newly arrived packets queued, but not dropped
Packet Dropping Probability

- Dropping probability is calculated in a way that depends on queue length and the number of packets not dropped since last packet drop
  - dropping probability based on queue length
  - \( P_b = \max_p \times (\text{avg} - \min_{\text{th}}) / (\max_{\text{th}} - \min_{\text{th}}) \)

- Just marking based on \( P_b \) can lead to clustered dropping. (can cause multiple drops in a single connection - due to bursts - which is unfair, similar to tail drop)
  - Dropping should be evenly distributed over timeline
  - RED likely drops packets from higher bandwidth-grabbing source

- Better to bias \( P_b \) by history of undropped packets
  - \( P_a = P_b / (1 - \text{count} \times P_b) \), where count the number of packets not dropped since the last drop
Notes re RED

• RED is not the same as Early Packet Discard (EPD) for ATM
  - EPD detects the initial cell of a packet
  - Congested ATM switch drops that cell and all cells belonging to the packet

• Partial Packet Discard (PPD)
  - Special case of dropping a packet due to congestion and then dropping all remaining packets for that segment
Source-based Avoidance

- TCP Vegas
TCP Vegas

- Idea: source watches for some sign that router’s queue is building up and congestion will happen; e.g.,
  - RTT grows
  - sending rate flattens
TCP Vegas

Increasing RTT
- We have seen that Timeout is considered as indication of congested state
- RTT is a good measure of load in network
- Increment in RTT is a indicator of increased load in bottleneck router
  - The router will soon go to a 'congested state'
- Sending packets in loaded state will probably only increase queue length in the bottleneck router
TCP Vegas

BaseRTT

- Define $BaseRTT$ to be the minimum of all RTTs when flow is not congested
- That is when all ACKs return within a round trip time
- Initially it is set to the RTT of first packet

We assume that we are not overflowing router queue---

- Compute $ExpectedRate = \frac{Current\ CongestionWindow}{BaseRTT}$
TCP Vegas Algorithm

- Let $BaseRTT$ be the minimum of all measured RTTs (commonly the RTT of the first packet)
- If not overflowing the connection, then
  \[
  \text{ExpectRate} = \frac{\text{CongestionWindow}}{BaseRTT}
  \]
- Source calculates sending rate ($ActualRate$) once per RTT
- Source compares $ActualRate$ with $ExpectRate$

\[
\text{Diff} = \text{ExpectedRate} - \text{ActualRate}
\]
if Diff < $\alpha$
  increase CongestionWindow linearly
else if Diff > $\beta$
  decrease CongestionWindow linearly
else
  leave CongestionWindow unchanged
Algorithm (cont)

- **Parameters**
  - $\alpha = 1$ packet
  - $\beta = 3$ packets

- **Even faster retransmit**
  - keep fine-grained timestamps for each packet
  - check for timeout on first duplicate ACK
Source- and Router-based Avoidance

• ECN
Active Queue Management (rfc2309)

• Simplest for of AQM is *drop tail*
• Lock-out problem
  - *drop-tail* allows a few flows to monopolize the queue space, locking out other flows
• Full queues problem:
  - *drop tail* maintains full or nearly-full queues during congestion
    • Increases delay
  - *queue length limits* should reflect the size of bursts to absorb, not steady-state queuing
• AQM involves both the end-hosts & routers
Explicit Congestion Notification in IP

- Two bits from the IP header will be used for Explicit Congestion Notification.
- Bits 6 and 7 in the TOS (Type of Service) octet are designated as the ECN field:
  - 0 0 - Not ECN-Capable Transport
  - 0 1 - ECN-Capable Transport (ECT(0))
  - 1 0 - ECN-Capable Transport (ECT(1))
  - 1 1 - Congestion Experienced (CE)
- The not-ECT code point ‘00’ indicates a packet that is not using ECN.
Explicit Congestion Notification in IP

- The ECN-Capable Transport code points ‘01’ and ‘10’ are set by the data sender to indicate that the endpoints of the transport protocol are ECN-capable.

- Routers treat the ECT(0) and ECT(1) as equivalent.

- The CE code point ‘11’ is set by a router to indicate congestion to the end nodes.
  - Routers that have a packet arriving at a full queue drop the packet, just as they do in the absence of ECN.
Congestion Notification - end nodes

- Upon the receipt by an ECN-Capable transport of a single CE packet
  - the congestion control algorithm must be essentially the same as the congestion control response to a single dropped packet.
  - E.g., ECN-capable TCP source halves its congestion window for either a packet drop or an ECN indication
Congestion Notification - end nodes

• The same congestion-control response accommodates the incremental deployment of ECN in both end-systems and in routers
  - If there were different congestion control responses to a CE code point than to a packet drop
    • this could result in unfair treatment for different flows

• An additional goal is that the end-systems should react to congestion at most once per RTT
  - to avoid reacting multiple times to multiple indications of congestion within a round-trip time
Congestion Notification - routers

- Routers should set the CE code point of an ECN-Capable packet
  - only if it would otherwise drop the packet as an indication of congestion to the end nodes
- When a CE packet (i.e., a packet that has the CE code point set) is received by a router
  - the CE code point is left unchanged
  - the packet is transmitted as usual
Congestion Notification - routers

- An environment where all end-nodes are ECN-Capable
  - could allow new criteria to be developed for setting the CE code point, and
  - new congestion control mechanisms could be developed for end-node reaction to CE packets

- Additional treatment should be done for fragmented and encapsulated packets
End of Class 19