TCP Congestion Control

Module Goals

At the conclusion of this module, students will be able to

- explain TCP's implementation of congestion control
- calculate the size of the TCP congestion window in various situations
- describe the difference between congestion control and congestion avoidance
- describe modern congestion avoidance schemes

Overview

- the previous lesson focused on general concepts of congestion and congestion control
- TCP's congestion control is the most common congestion control paradigm in use today
- basic idea: hosts send packets based on a window size and react to observable events in the network
  (e.g. dropped packets)
- the devil is in the details...

TCP Congestion Control

- TCP congestion control was indroduces into the Internet in the late 1980s by Vaan Jacobson
- roughly eight years after the TCP./IP protocol stack has become operational
- immediately preceding this time, the Internet was suffering from **congestion collapse**

Congestion Collapse

- hosts would... send packets into the Internet as fast as the advertised window would allow
- then... congestion would occur at some router(causing packets to be dropped)
- then... the hosts would time out and retransmit their packets, resulting in even more congestion
- lather, rinse, repeat

Self-Clocking

- each TCP source determines how much capacity is available in the network, so that it knows how many packets it can safely have in transit
- a key aspect of this is **self-clocking**!
- recall: if there is data in-flight, a TCP host doesn't send another packet until it receives the ACK for successfully received packet
- but how do we use this to determine the network capacity?

Additional State

- TCP maintains a new state variable for each connection, called $\mathrm{CongestionWindow}$
- used by the source to limit how much data it is allowed to have in transit at any given time

  (congestion control's counterpart to flow controls $\mathrm{AdvertisedWindow}$)
- TCP is modified such that the maximum number of bytes of unacknowledged data allowed is either
  - the size of the congestion window, or...
  - the size of teh advertised window,

  specifically, whichever is smaller

The New Effective Window

- TCP's effective window is revised as follows:

$$\begin{aligned}\text{EffectiveWindow} =& \text{MaxWindow} - \\ & (\text{LastByteSent} - \text{LastByteAckd})\end{aligned}$$

  where

$$\text{MaxWindow} = \min(\text{CongestionWindow}, \text{AdvertisedWindow})$$

- a TCP source is allowed to send no faster than the slowest component—the network, or the destination host—can accommodate

Still Haven't Answered the Question

- we still haven't really answered the question:

  how do we know how big the congestion window is, and therefore, the capacity of the network?

- unlike $\mathrm{AdvertisedWindow}$, which is sent by the receiving side of the connection, there is no one to send a suitable $\mathrm{CongestionWindow}$ to the sender
- instead, the TCP source sets $\mathrm{CongestionWindow}$ based on the level of congestion it perceives to exist in the network
- this involves decreasing the congestion window when the level of congestion goes up, and increasing the congestion window when the level of congestion goes down
- this leads us to **additive increase / multiplicative decrease**

Rules for Modifying the Congestion Window

- the main reason packets are not delivered in TCP is not because of errors or link failures
- the main reason is due to routers dropping packets somewhere
- a dropped packet doesn't get ACKed and therefore times out
- TCP interprets timeout as a sign of congestion and reduces the rate at which it is transmitting
- specifically, each time a timeout occurs, the source sets $\mathrm{CongestionWindow}$ to half of its previous value
  (multiplicative decrease

Rules for Modifying the Congestion Window

- although $\mathrm{CongestionWindow}$ is defined in terms of bytes, it's easiest to understand multiplicative decrease if we think in terms of whole packets
  - suppose $\mathrm{CongestionWindow}$ is currently set to 16 packets: if a loss is detected, $\mathrm{CongestionWindow}$ is set to 8 packets
  - additional losses cause $\mathrm{CongestionWindow}$ to be reduced to 4, 2, and finally 1 packet
- recall that the maximum segment size (MSS) is the largest unit of data that can be recieved as a single TCP segment
- as it doesn't make sense to force TCP to send less than a single TCP segment (due to congestion), $\mathrm{CongestionWindow}$ is not allowed to fall below the MSS

Rules for Modifying the Congestion Window

- now that we've discussed how to decrease the window size due to congestion, how to we open it back up when
  - new linke capacity is added, or
  - old link capacity becomes available again?
- each time the source successfully sends a $\mathrm{CongestionWindow}$ worth of packets—that is, each packet sent out during the last RTT has been ACKed— it adds 1 MSS to the window size (additive increase)

How it Really Works!

- TCP does not actually wait for an entire window's worth of ACKs to add 1 MSS to the window
- instead, the $\mathrm{CongestionWindow}$ is incremented a little for each ACK that arrives:

$$\mathrm{Increment} = \mathrm{MSS} \cdot {}^{\mathrm{MSS}}/_{\mathrm{CongestionWindow}}$$

Congestion Window Example

A Couple of Notes

- it has been shown tha AIMD is required in order for a congestion control mechanism to be stable
- why wouldn't AIAD or MIAD work (intuitively)?
- if congestion is detected via timeouts, it's important that our timeout calculations are accurate

Slow Start

- a bad name for a good idea
  (there is historical rationale behind the name)
- AIMD is the right approach to use when the source is operating close to the available capacity of the network, but...
- it takes too long to ramp up a connection when it is starting from scratch
- slow start is used to increase the congestion window rapidly from a cold start
- slow start will effectively increase the congestion window exponentially, rather than linearly

"Fast" Slow Start

- the source starts by setting $\mathrm{CongestionWindow}$ to one MSS
- when the ACK for this packet arrives, TCP adds 1 MSS to $\mathrm{CongestionWindow}$ and then sends two packets
- in general, each ACK causes the the congestion window to increase by 1 MSS
- the end result is that TCP effictively doubls the number of packets it has in transit every RTT

Stopping Slow Start

- when does slow start stop?
- when a congestion-related loss occurs, we store the value of $\frac{1}{2}\mathrm{CongestionWindow}$ into a variable called $\mathrm{ssthresh}$ and set the congestion window back to 1 MSS
- after that, receiving ACKs causes us to increase the window again
- but are we increasing in slow start mode or normal mode?

Increasing the Window

- if the congestion window is less than `ssthresh`, slow start is still used until the window gets back to `ssthresh`
- after that point, AIMD takes over
- this moves us quickly back to a level that hsould be close to a reasonable congestion window, then slows the rate of increase

Quick Start

- an alternative to slow start
- allows the initiator of a TCP connection to make a rate request in the SYN packet
- routers along the way determine if that is reasonable and either modify or pass along the rate
- if a router does not support quick start then the connection defaults back to the slow start mechanism

All Original

- the thresholds and slow start are all part of the original TCP congestion control standard
- however, TCP didn't use a very good clock, so the connection would go dead for some time while a packet was timing out
- to fix this, a new mechanism called **fast retransmit** was added to TCP
- the basic idea is to use a heuristic to determine when a packet might be dropped and retransmit it before the timeout actually occurs

Fast Retransmit

- the implementation is simple
- a packet arrives at the receiver and the receiver ACKs it
  (even if this sequence number has already been acknowledged!)
- when a packet arrives out of order, a TCP host will be resending the same acknowledgement it sent the last time

Fast Retransmit

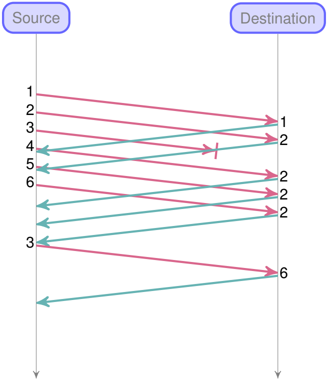
- this subsequent transmission of the same acknowledgement is called a **duplicate ACK**
- a duplicate ACK is a clue to the sender that a packet has been delivered out-of-order

  (though it might take some work to figure out which)
- the packet might just be delayed, so we don't want to retransmit right away
- we wait for multiple duplicate ACKs before retransmitting
- specifically, we wait for three duplicate ACKs

  (this eliminates about 50% of timeouts)

Fast Retransmit Example

- packets 1 and 2 are acknowledged properly, then packet 3 goes missing
- packets 4, 5, and 6 are all ACKed as packet 2
- after 3 of those duplicate ACKs, we resent packet 3
- then the receiver ACKs the laargest sequence number it has received



Fast Recovery

- because the timeout doesn't wait for the link to be completely dead, there arae ACKs that are still in the pipe to clock the sending of packets
- therefore we don't need to set the congestion window to 1 and slow start again
- this is called **fast recovery**

Congestion Avoidance

- it's important to understand that TCP only controls congestion once it happens
- in fact, TCP repeatedly increases the load it imposes on the network in an effort to find the congestion point, then backs off
- in other words, TCP creates congestion before it contains it
- an approach that is undergoing development is trying to predict when congestion is about to happen, then reduce the rates at which hosts send data just before packets start being discarded
- we will call this strategy **congestion avoidance**

Three Strategies

- DECbit: add a congestion bit to the header of a packet, which can be set by the router when congestion is about to occur
- Random Early Detection: implicitly notify a source of pending congestion by dropping a packet before congestion occurs
- Source-Based Congestion Avoidance: check RTTs and modify congestion window appropriately

DECbit

- designed by the Digital Equipment Corporation (DEC) for use on the Digital Network Architecture (DNA)
- the DNA is a connectionless network with a connection-oriented transport protocol
- therefore, the mechanism could also be applied to TCP and IP
- the idea is to more evenly split the responsibility for congestion control between the routers and the end nodes

DECbit

- each router monitors the load it is experiencing and explicitly notifies the end nodes when congestion is about to occur
- notification is implemented by setting a binary congestion bit in the packets that flow through the router
  (hence the name DECbit)
- the destination host then copies this congestion bit into the AACK it sends back to the source
- finally, the source adjusts its sending rate so as to avoid congestion

When is Congestion Pending?

- a router sets this bit in a packet if its average queue length is greater than or equal to 1 at the time the packet arrives
- this average queue length is measured over a time interval that spans the last busy+idle cycle, plus the current busy cycle
- using a queue length of 1 as the trigger is a trade-off between
  - significant queuing (higher throughput)
  - increased idle time (lower delay)

DECbit

- the source records how many of its packets resulted in some router setting the congestion bit
- the fraction of the last window's worth of packets resulted in the bit being set determines how the congestion window is altered
  - if less than 50%, then the source increases its congestion window by one MSS
  - if more than 50%, then the source decreases its congestion window to 0.875 times the previous value

DECbit Analysis

- the 50% threshold was chosen based on analysis that showed it to correspond to the peak of something called the **power curve**
- the "increase by 1, decrease by 0.875" rule was selected because AIMD seemed to make the mechanism stable
- the result of analysis, or simply tinkering until it worked?

Explicit Congestion Notification

- a proposal (RFC3168) existed for a long time to add explicit congestion notification to the network layer
- it takes two bits in the TOS octet of the IP header that are currently not used (long story)
- bits indicate congestion levels
  - `00`: I don't do ECN, thanks (default / do nothing)
  - `01`: I do ECN and nothing's wrong
  - `10`: I do ECN and nothing's wrong
  - `11`: I do ECN and there's congestion
- what defines "congestion" to a given router?

  (the RFC forgot to mention)
- receiving a packet with the ECN bit set should cause the host to do the same thing it would do if a packet was dropped; that is,

Explicit Congestion Notification

- unfortunately, ECN was poorly adopted as of 2011, and many devices “mangled” the ECN fields
- only $\approx 25\%$ of hosts negotiated with ECN as of 2012
- note that, while standardized in 2001, ECN didn’t even receive major OS support until 2007
- ECN does seem to be gaining more traction in IPv6, but IPv6 adoption is really slow

Random Early Detection

- **random early detection** (RED) was developed in the early 1990s
- another protocol that shares the load between routers and sources by having routers monitor their own load and inform sources when congestion is about to occur
- what are the differences between RED and explicit notification methods such as DECbit

Random Early Detection vs DECbit

- for on, RED was meant to work with a protocol that still exists (TCP)
- that means it doesn't necessarily need explicit notification like DECbit does
  - it implicitly notifies the source of congestion by dropping one of its packets
  - therefore, the source is notified by hte subsequent timeout or duplicate ACK and uses standard congestion rules
- RED also uses a more complicated way of determining when congestion is about to occur

When (and What) to Drop?

- we want to preempt congestion
  (why we call it collision **avoidance**)
- basic idea: let's drop a few packets now, instead of lots of packets later
- the easiest way to handle dropping packets is called **tail drop**, where a router's queue is full and everything that arrived afterwards was dropped
- doesn't preempt the problem however

When (and What) to Drop?

- RED makes a decision on whether or not to drop **each arriving packet** with some drop probability whenever the queue length exceeds some level
- this idea is called **random early drop**
- RED also define sthe details of how to monitor the queue length and when to drop a packet

RED Algorithm

- compute a weighted running average of the queue length, similar to the original TCP timeout computation:

  $$\mathrm{AvgLength} = \mathrm{Weight} \cdot \mathrm{SampleLength} + (1 - \mathrm{Weight}) \cdot \mathrm{AvgLength}$$

  where $0 < \mathrm{Weight} < 1$ and $\mathrm{SampleLength}$ is the length measurement of the queue
- in most software implementations, the queue length is measured every time a new packet arrives at the gateway
- in hardware implementations, it might be calculated at some fixed sampling interval

RED Algorithm

- RED keeps two queue length thresholds: $\mathrm{MinThreshold}$, $\mathrm{MaxThreshold}$
- when a packet arrives at the gateway, RED compares the current $\mathrm{AveLength}$ with the two thresholds according to the following rules:
  - $\mathrm{AveLength} \leq \mathrm{MinThreshold}$ (things are great!)

    queue the packet
  - $\mathrm{MinThreshold} < \mathrm{AveLength} < \mathrm{MaxThreshold}$ (getting scary!)

    drop the arriving packet with some probability p
  - $\mathrm{MaxThreshold} \leq \mathrm{AveLength}$ (time for drastic measures!)

    drop the arriving packet

RED Algorithm

- p is not a fixed probability
- rather, it is a function of $\mathtt{AvgLength}$ and how long it has been since the last packet was dropped:

$$p = \bar{p} / 1 - \mathrm{Count} \cdot \bar{p}$$

where

$$\bar{p} \ = \ \mathrm{MaxProb} \cdot \frac{\mathrm{AvgLength} - \mathrm{MinThreshold}}{\mathrm{MaxThreshold} - \mathrm{MinThreshold}}$$

- Count is the number of packets that have successfully been added to the queue
- $\mathrm{MaxProb}$ is the maximum probability a packet will be dropped
- this allows for packets to be more evenly dropped over time instead of in clusters

RED Algorithm

- a fair amount of analysis has gone into determining “good” parameters, but...
- the algorithm doesn’t seem especially sensitive to parameter changes
- some rules of thumb
  - Weight should be set so that changes in queue size over time periods less than 100 ms should be filtered out
  - $\mathrm{MaxThreshold}$ should be about twice $\mathrm{MinThreshold}$
- what if hosts ignore these RED’s signals? **unresponsive flow problem**

Source-Based Congestion Avoidance

- so far, evenything has been the responsibility of the routers
  (they know how long their queues are)
- in source-based avoidance, the hosts themselves watch for some sign from the network that queues are building up
- what kind of sign? RTT is an easy one!
- for example, the source might notices that as packet queues build up in the routers, there is a measureable increase in the RTT for each successive packet it sends

Using RTT for Avoidance

- Algorithm 1:
  - the congestion window normally increases as in TCP, but every two round-trip delays the algorithm checks to see if the current RTT is greater than the average of the minimum and maximum RTTs seen so far
  - if so, the algorithm decreases the congestion window by one-eighth

Using RTT for Avoidance

- Algorithm 2:
  - similar, but factors in window size
  - the window is adjusted once every two round-trip delays based on the product

$$(\mathrm{CurrWindow} - \mathrm{PrevWindow}) \cdot (\mathrm{CurrRTT} - \mathrm{PrevRTT})$$

  - if positive, the source decreases the window size by one-eighth
  - if negative, the srouce increases the window by 1 MSS
  - the window will change during every adjustment and oscillate around its optimal point

Vegas, Baby!

- the final option which we will mention (but not discuss in detail) is TCP Vegas
- basic idea: calculate the throughput of the link and adjust the congestion window accordingly
- the source attempts to keep the link busy, but not too busy, using some thresholds based on the difference between the expected rate of transfer and the actual rate
- this version of TCP competes with older versions Reno and Tahoe

Vegas, Baby!

- Vegas never garnered much support
- other versions of congestion avoidance have taken it's place
  - Hybla: changest in the congestion window in addition to RTT
  - Compound TCP: Microsoft's dual congestion windows (one AIMD-based, one delay-based)
  - CUBIC: used in Linux kernels from 2.6.19 until 3.2, then Linux switched to Proportional Rate Reduction
  - Bottleneck Bandwidth and Round Trip Propagation Time (BBR): Google's new mechanism not based on packet-loss