ECE/COMPSCI 356 Computer Network Architecture

Lecture 20: TCP Congestion Control

Neil Gong

neil.gong@duke.edu

Slides credit: Xiaowei Yang, PD

Overview

- Additive increase multiplicative decrease
- Slow start
- Fast retransmit and fast recovery

Congestion Control

- Different TCP connections compete for resources
  - Bandwidth of the links
  - Buffers at the routers and switches

- Packets contend at a router for the use of a link

- Contending packets are placed in a queue

Congestion Control

- When too many packets are contending for the same link
  - The queue overflows
  - Packets get dropped
    - Network is congested!

- Network should provide a congestion control mechanism to deal with such a situation

- TCP enables a sender to dynamically adjust its sending speed for congestion control

TCP congestion control

- Introduced in the late 1980s by Van Jacobson
  - Eight years after the TCP/IP had become operational.
- Before this, the Internet suffered from congestion collapse
  - hosts send as fast as the advertised window would allow
  - congestion would occur at some router, causing packets to be dropped
  - hosts time out and retransmit, resulting in even more congestion

TCP congestion control

- The idea: a sender determines network capacity and adjusts sending speed
  - How can a sender know it is safe to send more packets?
  - Uses the arrival of an ACK as a signal
  - By using ACKs to pace the transmission of packets, TCP is said to be *self-clocking*.

CongestionWindow

- A new state variable: *CongestionWindow*
  - limit how much data a sender can have in transit.
  - congestion control's counterpart to flow control's AdvertisedWindow.
- Maximum number of unacknowledged bytes is the minimum of the *CongestionWindow* and the AdvertisedWindow

New effective window

- TCP’s effective window is revised as follows:
  - MaxWindow = MIN(CongestionWindow, AdvertisedWindow)
  - EffectiveWindow = MaxWindow − (LastByteSent − LastByteAcked).
- MaxWindow replaces AdvertisedWindow when calculating EffectiveWindow.
- A sender sends no faster than the slowest component—the network or the destination host—can accommodate.

How to determine CongestionWindow size

- Unlike AdvertisedWindow
  - Sent by receiver
- Sender sets CongestionWindow based on the level of congestion it perceives to exist in the network.
- Additive increase/multiplicative decrease (AIMD)
  - Decreasing CongestionWindow when the level of congestion goes up
  - Increasing CongestionWindow when the level of congestion goes down.

When to decrease CongestionWindow?

- Main reason for timeout: congestion/dropped packets
- Therefore, timeout is a sign of congestion and CongestionWindow reduces
- Each time a timeout occurs
  - CongestionWindow reduces by half
    - “multiplicative decrease” part of AIMD

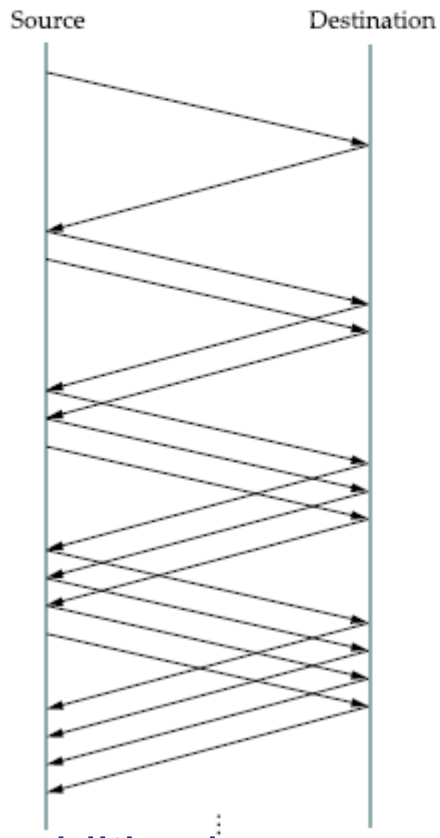
When to decrease CongestionWindow?

- CongestionWindow is defined in terms of bytes
- Easier to understand multiplicative decrease if we think in terms of whole packets.
  - E.g., suppose the CongestionWindow is currently set to 16 packets. If a timeout, CongestionWindow is set to 8.
  - Additional timeouts cause CongestionWindow to be reduced to 4, then 2, and finally to 1 packet.
  - CongestionWindow is not allowed to fall below the size of a single packet, or the *maximum segment size (MSS)*.

When to increase CongestionWindow

- Every time the sender successfully sends a CongestionWindow's worth of packets
  - i.e., each packet sent out during the last RTT has been ACKed
  - it adds the equivalent of 1 packet to CongestionWindow.
- "additive increase" part of AIMD

Illustration



Packets in transit during additive increase, with one packet being added each RTT.

When to increase CongestionWindow

- In practice, sender does not wait for an entire window's worth of ACKs to add 1 packet to CongestionWindow
- Increment CongestionWindow by a little for each ACK.
- CongestionWindow is incremented each time an ACK arrives:
  - Increment = MSS × (MSS/CongestionWindow)
  - CongestionWindow+= Increment
  - Rather than incrementing CongestionWindow by an entire MSS bytes each RTT, we increment it by a fraction of MSS every time an ACK is received.
  - Assuming that each ACK acknowledges the receipt of MSS bytes, then that fraction is MSS/CongestionWindow.

Problem of additive increase

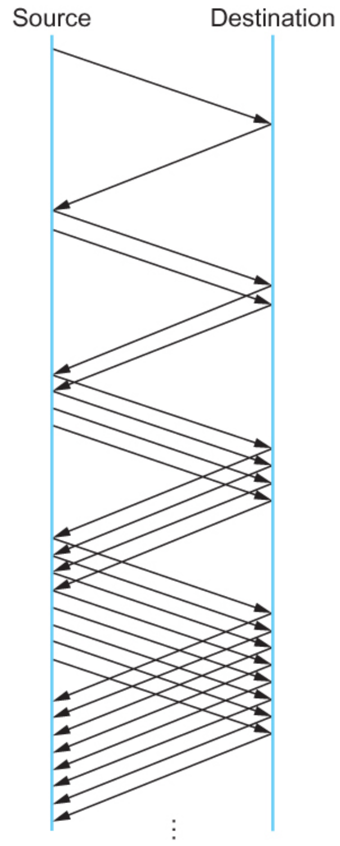
- It takes too long to ramp up a connection when starting from scratch
  – CongestionWindow is initialized as 1.

- Solution: *slow start*.
  – Increases CongestionWindow exponentially, rather than linearly.

Slow Start

- Sender initializes CongestionWindow as one packet.
- Upon receiving an ACK, increments CongestionWindow by 1.
- Sender doubles the number of packets every RTT.

Slow Start



Packets in transit during slow start.

When to use slow start – situation 1

- Very beginning of a connection.
  - slow start continues to increment CongestionWindow by 1 packet each ACK until a timeout occurs

When to use slow start – situation 2

- A timeout occurs
  - multiplicative decrease to divide CongestionWindow by 2.
    - Variable CongestionThreshold = CongestionWindow/2
  - CongestionWindow initializes as 1 packet
  - Increment by 1 packet every ACK until reaching CongestionThreshold
  - After that, additive increase

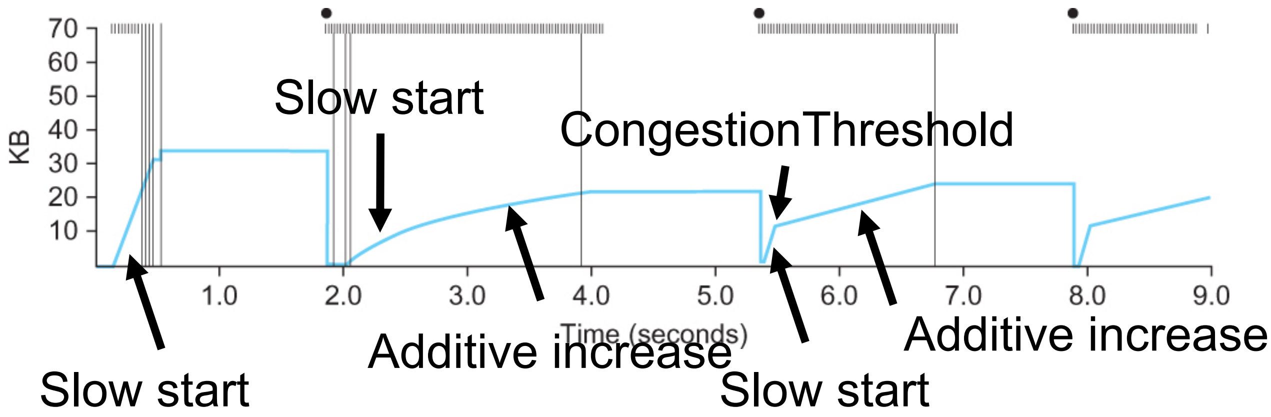
When to use slow start – situation 2

– CongestionWindow increases as follows upon receiving an ACK:

```
u_int cw = state->CongestionWindow;
u_int incr = state->maxseg;
if (cw > state->CongestionThreshold)
        incr = incr * incr / cw;
state->CongestionWindow = MIN(cw + incr,
TCP_MAXWIN);
```

- state represents the state of a TCP connection and TCP_MAXWIN is upper bound of CongestionWindow.

CongestionWindow over time



Behavior of TCP congestion control. Colored line = value of CongestionWindow over time; solid bullets at top of graph = timeouts; hash marks at top of graph = time when each packet is transmitted; vertical bars = time when a packet that was eventually retransmitted was first transmitted.

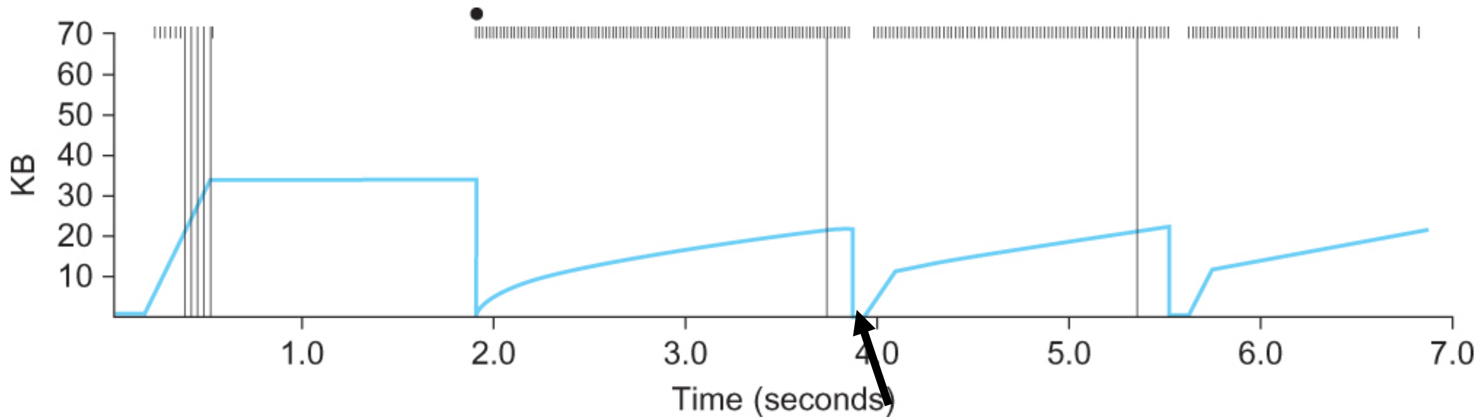
Fast Retransmit

- Problem: long waiting time before timeout.
- Solution: *fast retransmit* was added to TCP.
  - A heuristic that sometimes triggers retransmission sooner than timeout.

Fast Retransmit

- Receiver resends the same acknowledgment it sent the last time when receiving an out of order packet
  - Called *duplicate ACK*
  - Used together with selective ACK
- Sender knows earlier packet might have been lost when seeing a duplicate ACK.
- Sender waits until it has seen three duplicate ACKs before retransmitting the packet.
  - In case the packet is delayed instead of being dropped
  - Slow start or additive increase for CongestionWindow

Fast Retransmit



Fast retransmit is triggered
Slow start to increment CongestionWindow

Trace of TCP with fast retransmit. Colored line = CongestionWindow; solid bullet = timeout; hash marks = time when each packet is transmitted; vertical bars = time when a packet that was eventually retransmitted was first transmitted.

Fast Recovery

- Problem: when fast retransmit is triggered, congestion is not too bad (compared to timeout)
  - But slow start resets CongestionWindow to be 1 and increments it
  - Does not leverage network capacity
- Solution: *fast recovery* uses the following CongestionWindow
  - Initialized as CongestionThreshold
  - additive increase

Summary

- Additive increase multiplicative decrease
- Slow start
- Fast retransmit and fast recovery